

Programmer's Reference Manual Volume 2

© 1983-1993 The Santa Cruz Operation, Inc.
© 1980-1990 Microsoft Corporation.
© 1989-1990 UNIX System Laboratories, Inc.
All Rights Reserved.

No part of this publication may be reproduced, transmitted, stored in a retrieval system, nor translated into any human or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without the prior written permission of the copyright owner, The Santa Cruz Operation, Inc., 400 Encinal, Santa Cruz, California, 95060, U.S.A. Copyright infringement is a serious matter under the United States and foreign Copyright Laws.

The copyrighted software that accompanies this manual is licensed to the End User only for use in strict accordance with the End User License Agreement, which should be read carefully before commencing use of the software. Information in this document is subject to change without notice and does not represent a commitment on the part of The Santa Cruz Operation, Inc.

SCO OPEN DESKTOP Software is commercial computer software and, together with any related documentation, is subject to the restrictions on U.S. Government use as set forth below.

If this procurement is for a DOD agency, the following DFAR Restricted Rights Legend applies:

RESTRICTED RIGHTS LEGEND:

USE, DUPLICATION OR DISCLOSURE BY THE GOVERNMENT IS SUBJECT TO RESTRICTIONS AS SET FORTH IN SUBPARAGRAPH (c)(1)(ii) OF RIGHTS IN TECHNICAL DATA AND COMPUTER SOFTWARE CLAUSE AT DFARS 252.227-7013. CONTRACTOR/MANUFACTURER IS THE SANTA CRUZ OPERATION, INC., 400 ENCINAL STREET, SANTA CRUZ, CA 95060.

If this procurement is for a civilian government agency, the following FAR Restricted Rights Legend applies:

RESTRICTED RIGHTS LEGEND:

THIS COMPUTER SOFTWARE IS SUBMITTED WITH RESTRICTED RIGHTS UNDER GOVERNMENT CONTRACT NO. _____ (AND SUBCONTRACT NO. _____, IF APPROPRIATE). IT MAY NOT BE USED, REPRODUCED, OR DISCLOSED BY THE GOVERNMENT EXCEPT AS PROVIDED IN PARAGRAPH (g)(3)(i) OF FAR CLAUSE 52.227-14 OR AS OTHERWISE EXPRESSLY STATED IN THE CONTRACT. CONTRACTOR/MANUFACTURER IS THE SANTA CRUZ OPERATION, INC., 400 ENCINAL STREET, SANTA CRUZ, CA 95060.

SCO, Open Desktop, The Santa Cruz Operation, the Open Desktop logo, and the SCO logo are registered trademarks of The Santa Cruz Operation, Inc. in the USA and other countries.

SCO NFS was developed by Lachman Technology, Inc. based on **LACHMAN SYSTEM V NFS.5**. **LACHMAN** is a trademark of Lachman Technology, Inc. **NFS** is a registered trademark of Sun Microsystems, Inc.

SCO TCP/IP was developed by Lachman Technology, Inc. **SCO TCP/IP** is derived from **LACHMAN SYSTEM V STREAMS TCP**, a joint development of Lachman Technology, Inc. and Convergent Technologies. **LACHMAN** is a trademark of Lachman Technology, Inc.

All other brand and product names are or may be trademarks of, and are used to identify products or services of, their respective owners.

Date: 6 January 1993

Document Version: 3.2.4F

Preface 1

Documentation conventions	2
Development System documentation set	3
Reference manual sections	7
Alphabetized list	9

System services (S) 33

Intro(S)	35
Routines(S)	51
a64l(S)	83
abort(S)	84
abs(S)	85
accept_pw(S)	86
access(S)	88
acct(S)	91
alarm(S)	93
assert(S)	94
atexit(S)	96
atof(S)	98
audit(S)	100
authaudit(S)	101
authcap(S)	104
bessel(S)	107
brk(S)	109
brkctl(S)	111
bsearch(S)	115
catgets(S)	118
catopen(S)	120
cfspeed(S)	122
chdir(S)	124
chmod(S)	126
chown(S)	129
chroot(S)	131
chsize(S)	133
clock(S)	134
close(S)	135
creat(S)	137
create_file_securely(S)	140

creatsem(S)	143
crypt(S)	145
ctermid(S)	148
ctime(S)	149
ctype(S)	155
curses(S)	158
cuserid(S)	198
dblock(S)	200
dbm(S)	202
defopen(S)	206
dial(S)	208
difftime(S)	211
directory(S)	213
div(S)	216
drand48(S)	218
dup(S)	221
dup2(S)	223
ecvt(S)	224
end(S)	226
erf(S)	227
ev_block(S)	228
ev_close(S)	229
ev_count(S)	230
ev_flush(S)	231
ev_getdev(S)	232
ev_getemask(S)	234
ev_gindev(S)	235
ev_init(S)	236
ev_open(S)	238
ev_pop(S)	240
ev_read(S)	241
ev_resume(S)	242
ev_setemask(S)	243
ev_suspend(S)	245
exec(S)	246
execseg(S)	250
exit(S)	253
exp(S)	255
fclose(S)	257
fcntl(S)	259
ferror(S)	263

fgetpos(S)	265
field(S)	268
fields(S)	273
fieldtype(S)	275
floor(S)	278
fopen(S)	279
fork(S)	282
form(S)	285
fpgetround(S)	291
fread(S)	293
frexp(S)	295
fseek(S)	296
fsetpos(S)	298
fsync(S)	300
ftok(S)	301
ftw(S)	303
gamma(S)	305
getc(S)	307
getcwd(S)	310
getdents(S)	312
getdvagent(S)	314
getenv(S)	317
getgrent(S)	318
getgroups(S)	321
gethz(S)	323
getitimer(S)	324
getlogin(S)	328
getluid(S)	330
getmsg(S)	331
getopt(S)	334
getpass(S)	337
getpasswd(S)	339
getpid(S)	342
getprdfent(S)	343
getprfient(S)	346
getpriv(S)	349
getprpwent(S)	351
getprtcent(S)	357
getpw(S)	360
getpwent(S)	361
gets(S)	364

getspent(S)	366
getuid(S)	369
getut(S)	370
hsearch(S)	374
hypot(S)	378
identity(S)	379
ioctl(S)	382
isaddindex(S)	384
isbuild(S)	385
isclose(S)	387
isconv(S)	388
isdelcurr(S)	391
isdelete(S)	392
isdelindex(S)	393
isdelrec(S)	394
iserase(S)	395
isindexinfo(S)	396
islock(S)	398
isnan(S)	399
isopen(S)	401
isread(S)	403
isrelease(S)	405
isrename(S)	406
isrewcurr(S)	407
isrewrec(S)	408
isrewrite(S)	409
issetunique(S)	411
isstart(S)	412
isuniqueid(S)	415
isunlock(S)	416
iswrcurr(S)	417
iswrite(S)	419
item(S)	420
kill(S)	423
l3tol(S)	425
labs(S)	426
ldahread(S)	427
ldclose(S)	428
ldfhread(S)	429
ldgetname(S)	430
ldiv(S)	431

ldlread(S)	433
ldlseek(S)	435
ldohseek(S)	436
ldopen(S)	437
ldptr(S)	439
ldrseek(S)	441
ldshread(S)	442
ldsseek(S)	443
ldtbindex(S)	444
ldtbread(S)	445
ldtbseek(S)	446
libwindows(S)	447
link(S)	450
llog(S)	452
lock(S)	456
lockf(S)	457
locking(S)	460
localeconv(S)	463
logname(S)	464
lsearch(S)	465
lseek(S)	467
malloc(S)	469
malloc(S)	472
matherr(S)	476
mblen(S)	479
memmove(S)	481
memory(S)	483
menu(S)	486
mkdir(S)	494
mkfifo(S)	496
mknod(S)	498
mktemp(S)	501
mktime(S)	502
ml_send(S)	504
mmdf(S)	507
monitor(S)	514
mount(S)	516
msgctl(S)	518
msgget(S)	520
msgop(S)	522
nap(S)	526

nice(S)	527
nl_cxtime(S)	528
nl_init(S)	530
nl_langinfo(S)	532
nl_printf(S)	533
nl_scanf(S)	535
nl_strcmp(S)	536
nlist(S)	537
open(S)	539
opensem(S)	543
paccess(S)	545
panel(S)	550
passlen(S)	553
pathconf(S)	555
pause(S)	558
perror(S)	559
phs(S)	561
pipe(S)	563
plock(S)	564
plot(S)	566
poll(S)	568
popen(S)	571
printf(S)	573
procti(S)	577
profil(S)	580
ptrace(S)	581
putc(S)	585
putenv(S)	587
putmsg(S)	588
putpwent(S)	591
puts(S)	592
putspent(S)	594
pw_nametoid(S)	595
qsort(S)	597
raise(S)	599
rand(S)	600
randomword(S)	602
rdchk(S)	604
read(S)	605
readlink(S)	608
regcmp(S)	610

regexp(S)	613
remove(S)	617
rename(S)	618
rmdir(S)	621
sc_init(S)	623
sc_raw(S)	626
sc_readkb(S)	629
scanf(S)	634
__scoinfo(S)	638
sdenter(S)	640
sdget(S)	642
sdgetv(S)	644
seed(S)	645
select(S)	647
semctl(S)	649
semget(S)	652
semop(S)	654
setbuf(S)	657
setgroups(S)	659
setjmp(S)	661
setlocale(S)	664
setluid(S)	667
setpgid(S)	669
setpgrp(S)	671
setpriv(S)	672
setsid(S)	675
setuid(S)	676
shmctl(S)	678
shmget(S)	680
shmop(S)	682
shutdn(S)	684
sigaction(S)	686
sigsetjmp(S)	692
signal(S)	693
sigpending(S)	698
sigprocmask(S)	699
sigsem(S)	701
sigset(S)	703
sigsetv(S)	705
sigsuspend(S)	711
sinh(S)	713

sleep(S)	715
sputl(S)	716
ssignal(S)	717
stat(S)	719
statfs(S)	723
stdio(S)	726
stime(S)	728
stopio(S)	729
strcoll(S)	731
strerror(S)	734
strftime(S)	736
string(S)	738
strtod(S)	742
strtol(S)	743
strtoul(S)	745
subsystems(S)	746
swab(S)	749
symlink(S)	750
sync(S)	753
sysconf(S)	754
sysfs(S)	756
sysi86(S)	758
system(S)	762
t_accept(S)	764
t_alloc(S)	767
t_bind(S)	770
t_close(S)	773
t_connect(S)	775
t_error(S)	778
t_free(S)	780
t_getinfo(S)	782
t_getstate(S)	784
t_listen(S)	786
t_look(S)	789
t_open(S)	791
t_optmgmt(S)	793
t_rcv(S)	796
t_rcvconnect(S)	799
t_rcvudata(S)	802
t_rcvuderr(S)	804
t_rcvdis(S)	806

t_rcvrel(S)	808
t_snd(S)	810
t_snddis(S)	813
t_sndrel(S)	815
t_sndudata(S)	817
t_sync(S)	819
t_unbind(S)	821
tai(S)	823
tam(S)	826
tcattr(S)	838
tcflow(S)	840
tcpgrp(S)	843
termcap(S)	845
terminfo(S)	847
time(S)	854
times(S)	856
tmpfile(S)	858
tmpnam(S)	860
toascii(S)	862
trig(S)	865
tsearch(S)	868
ttynam(S)	872
ttyslot(S)	874
uadmin(S)	875
ulimit(S)	877
umask(S)	879
umount(S)	880
uname(S)	882
ungetc(S)	884
unlink(S)	886
ustat(S)	888
utime(S)	890
varargs(S)	892
vprintf(S)	897
wait(S)	899
waitsem(S)	903
write(S)	905
xlist(S)	909

Preface

The *Programmer's Reference Manual* is a two-volume set that includes manual pages for the SCO[®] UNIX[®] Development System, including sections (CP), (FP), (DOS), (S) and (XNX).

This volume contains a complete set of the section (S) manual pages. These manual pages contain comprehensive descriptions of the system services, library routines, functions and macros included with the Development System.

All of these manual pages are accessible online through the **man** command.

Documentation conventions

SCO documents use font changes and other typographic conventions to distinguish text elements. The following table shows these conventions:

Font conventions

Example	Entity
cc or cc(CP)	command. The "CP" indicates the manual page section in which the command is documented.
<i>libc.a</i>	filename
<i>action</i>	placeholder
⟨Esc⟩	keyname
open or open(S)	system calls, library routines, kernel functions, C keywords. The "S" indicates the manual page section in which the command is documented.
buf	structure
b.errno	structure member
\$HOME	environment or shell variable
SIGHUP	named constant
login	output
"clobber"	jargon
"adm3a"	data value
date	user input

Development System documentation set

The contents of the SCO® Open Desktop® Development System documentation set are illustrated here. In addition to these books, a copy of *Release and Installation Notes* is provided with the Development System.

	Reference	Guide
UNIX® Development System	• Encyclopedia† • Programmer's Reference Manual (2 Volumes)† • Permuted Index†	• Developer's Overview† • Developer's Topics† • Programming Tools Guidet • Debugging Tools Guidet • User Interfaces Guidet
Packaging	• Software Mastering Toolkit Guide	
Networking	• Network Programmer's Guide and Reference	
Graphical Environment	• X Window System Programmer's Reference • OSF/Motif Programmer's Reference	• Integrating Applications into Open Desktop
Optional	• Macro Assembler Writer's Guide • Device Driver Writer's Guide	

† These manuals are distributed with both the UNIX Development System software and the Open Desktop Development System software.

The books included with the Development System are described here.

Developer's Overview

introduces the Development System facilities and gives general information about developing software to run on SCO UNIX systems and the supported cross-development environments.

Developer's Topics

a collection of technical papers about topics of interest to users of the Development System. Many of these papers include examples illustrating features that are extensions provided by SCO.

Encyclopedia

contains articles that give background information about system internals, descriptions of facilities, and other general issues. Articles are arranged alphabetically.

Programmer's Reference Manual

a two-volume set that includes manual pages for the entire UNIX Development System, including sections (CP), (DOS), (FP), (S), and (XNX). See the "Manual pages" article in the *Encyclopedia* for a description of these manual page sections.

Permuted Index

a permuted index of all manual pages that make up the Open Desktop Development System.

Programming Tools Guide

provides generally useful information about programming tools and their use in developing software. It also provides implementation-specific details about the ANSI-conforming C compiler provided with the Development System.

Debugging Tools Guide

provides generally useful information about the debugging tools, and their use in tracking down and eliminating problems in C and assembly language programs. In addition, it includes information on using dbXtra™ to debug C and C++ programs in an OSF/Motif™-based windowing environment.

User Interfaces Guide

introduces the facilities available for developing user interfaces, and gives detailed instructions for using curses(S) and the Extended Terminal Interface (ETI), and for writing user interfaces that can be run on ASCII terminals. This book includes numerous examples of curses programs.

Software Mastering Toolkit Guide

provides information and examples for using the Software Mastering Toolkit® to develop a custom-installable product. Also included are the (SMT) manual pages.

Network Programmer's Guide and Reference

provides procedures and reference information for developing applications that use the SCO networking services: TCP/IP, NFS™, IPX/SPX, TLI/XTI, int5c, and the PC-Interface™ extended library. As well, the (SLIB), (SSC), (NC), (NS), and (PCI) man pages are provided in this volume.

X Window System Programmer's Reference

provides X Window System™ programming commands and functions: X Library (XS), X Toolkit (Xt), X Miscellaneous Utilities (Xmu), and X Extensions (Xext) reference manual pages for programmers.

OSF/Motif Programmer's Reference

consists of the OSF/Motif toolkit, window manager, and user interface language commands and functions: (Xm) reference manual pages.

Integrating Applications into Open Desktop

provides information about application programming interfaces that help in integrating applications into the Open Desktop

graphical environment: putting an icon on the desktop, adapting to display resolution at runtime, supporting the desktop color selector, and communicating with the session manager.

In addition, the following books are available on request:

Macro Assembler Writer's Guide

provides information on the assembly language for the Intel® 286, 386, and 486 processors. Developers who are writing device drivers or applications having tight performance requirements may want to write portions in assembly language. This book is sold separately.

Device Driver Writer's Guide

provides guide and reference information about writing device drivers for SCO UNIX systems. This book is sold separately.

Using the Documentation

We recommend the following approach to using this documentation set. In general, we suggest the following:

1. All users should read the *Developer's Overview* thoroughly, soon after installing the software. Experienced users may skim some sections, but this book gives a comprehensive overview of the Development System's capabilities.
2. Use the *Encyclopedia* to look up topics about which you need more information. The *Encyclopedia* gives both summary and background information on many topics, and includes cross-references to sources of more in-depth information.
3. The *Programmer's Reference Manual* contains complete reference information about all capabilities provided in the UNIX Development System. New users are warned, however, that the manual pages provide reference material that may be difficult to utilize if they do not understand basic concepts.

The *Programmer's Reference Manual* contains separate sections for Development System commands (CP), system services (S), Development System file formats (FP), and other topics. The pages in each section of the *Programmer's Reference Manual* are arranged alphabetically, although some related facilities are grouped together on a page. If you cannot locate the correct manual page, use the permuted index to find the one that contains the information you want. The "Manual pages" article in the *Encyclopedia* explains how to use the permuted index.

4. The *Programming Tools Guide* and *Debugging Tools Guide* are mainly for users who have not used one of these tools before, or who want to

reacquaint themselves with some aspect of that tool's operation. These books are good places to look when the *Encyclopedia* and *Programmer's Reference Manual* are giving you lots of detail, but not enough guidance. Nonetheless, even experienced programmers will discover helpful hints and practical tips on the tools discussed in these books.

Commercial books and articles

A number of fine books and articles are published commercially that discuss how to develop software on the UNIX Operating System. We have not attempted to replicate guide information on all of these topics; manual pages are provided, and many articles in the *Encyclopedia* list additional sources of information on their subject matters. In addition, the *Encyclopedia* includes a large "Bibliography" that will interest users of the Development System.

In particular, we recommend that all developers have the following standard textbooks on their shelves:

Brian Kernighan and Dennis Ritchie, *The C Programming Language*, 2nd Edition.

One of the following general books about programming on UNIX operating systems:

Donald Lewine, *POSIX Programmer's Guide*.

Marc Rochkind, *Advanced UNIX Programming*.

W. Richard Stevens, *Advanced Programming in the UNIX Environment*.

W. Richard Stevens, *UNIX Network Programming*. Useful for programmers who are writing distributed applications.

Maurice Bach, *Design of the UNIX Operating System*. Useful for programmers who want to thoroughly understand UNIX system internals.

See the "Bibliography" in the *Encyclopedia* for full citations if you are not familiar with the books listed above.

Reference manual sections

Reference material is distributed as individual reference sections in the various volumes of the Operating and Development Systems. The following table lists the section name, description, and location of each reference section.

Section	Description	Volume
ADM	administrative commands	<i>System Administrator's Reference</i>
C	commands	<i>User's Reference</i>
CP	programming commands	<i>Programmer's Reference Manual</i>
DOS	DOS and OS/2 [®] library routines	<i>Programmer's Reference Manual</i>
F	file formats	<i>System Administrator's Reference</i>
FP	programming file formats	<i>Programmer's Reference Manual</i>
HW	hardware devices	<i>System Administrator's Reference</i>
K	kernel functions used in device drivers	<i>Device Driver Writer's Guide</i>
M	miscellaneous	<i>User's Reference</i>
NC	RPC protocol compiler	<i>Network Programmer's Guide and Reference</i>
NS	network system calls	<i>Network Programmer's Guide and Reference</i>
PCI	PC-Interface extended library	<i>Network Programmer's Guide and Reference</i>
S	system calls and library routines	<i>Programmer's Reference Manual</i>
SLIB	socket library functions	<i>Network Programmer's Guide and Reference</i>
SMT	Software Mastering Toolkit utilities	<i>Software Mastering Toolkit Guide</i>
SSC	socket system calls	<i>Network Programmer's Guide and Reference</i>
X	X clients	online only
XNX	XENIX [®] cross development	<i>Programmer's Reference Manual</i>
XS	X library routines	<i>X Window System Programmer's Reference</i>
Xext	X Extensions library	<i>X Window System Programmer's Reference</i>
Xm	OSF/Motif commands and functions	<i>OSF/Motif Programmer's Reference</i>
Xmu	Xmu library	<i>X Window System Programmer's Reference</i>
Xt	X Toolkit Intrinsics library	<i>X Window System Programmer's Reference</i>

The alphabetized list that follows is a complete listing of all UNIX commands, system calls, library routines, and file formats: (ADM), (C), (CP), (DOS), (F), (FP), (HW), (M), (S), (XNX).

Alphabetized list

Commands, system calls, library routines, and file formats

[	test (C)	admin	admin (CP)
300	300 (C)	advance	regexp (S)
300s	300 (C)	agetcommand	authcap (S)
4014	4014 (C)	agetdefault	authcap (S)
450	450 (C)	agetfile	authcap (S)
80387	80387 (HW)	agetflag	authcap (S)
86rel	86rel (FP)	agetstr	authcap (S)
a64l	a64l (S)	agettty	authcap (S)
abort	abort (S)	agetuser	authcap (S)
abs	abs (S)	aio	aio (F)
accept	accept (ADM)	aio	aio (M)
acceptable_password	accept_pw (S)	aioinfo	aioinfo (ADM)
access	access (S)	aiolkinit	aiolkinit (ADM)
acct	acct (ADM)	aiomemlock	aiomemlock (F)
acct	acct (FP)	alarm	alarm (S)
acct	acct (S)	ale	ale (ADM)
acctcms	acctcms (ADM)	allocldptr	ldptr (S)
acctcom	acctcom (ADM)	altzone	ctime (S)
acctcon	acctcon (ADM)	a.out	a.out (FP)
acctcon1	acctcon (ADM)	ap	ap (ADM)
acctcon2	acctcon (ADM)	ar	ar (CP)
acctdisk	acct (ADM)	ar	ar (FP)
acctdusg	acct (ADM)	ar	ar (XNX)
acctmrg	acctmrg (ADM)	arc	plot (S)
accton	acct (ADM)	archive	archive (F)
accton	accton (ADM)	as	as (CP)
acctprc	acctprc (ADM)	asa	undocumented (M)
acctprc1	acctprc (ADM)	ascii	ascii (M)
acctprc2	acctprc (ADM)	asctime	ctime (S)
acctsh	acctsh (ADM)	asetdefaults	authcap (S)
acctwtmpt	acct (ADM)	asin	trig (S)
acos	trig (S)	asktime	asktime (ADM)
adb	adb (CP)	asktimer	asktime (ADM)
addch	curses (S)	asroot	asroot (ADM)
addch	tam (S)	assert	assert (S)
addch	terminfo (S)	assign	assign (C)
addkey	curses (S)	at	at (C)
addkey	terminfo (S)	atan	trig (S)
addstr	curses (S)	atan2	trig (S)
addstr	tam (S)	atcronsh	atcronsh (ADM)
addstr	terminfo (S)	atexit	atexit (S)
addxusers	addxusers (ADM)	atof	atof (S)
adf_gttok	tam (S)	atoi	atof (S)
adf_gtwrd	tam (S)	atol	atof (S)
adf_gtxcd	tam (S)	attroff	curses (S)

attroff tam(S)
attroff terminfo(S)
attron curses(S)
attron tam(S)
attron terminfo(S)
attrset curses(S)
attrset terminfo(S)
audit audit(HW)
audit_adjust_mask authaudit(S)
audit_auth_entry authaudit(S)
audit_close audit(S)
auditcmd auditcmd(ADM)
auditt auditt(ADM)
audit_lax_file authaudit(S)
audit_lock authaudit(S)
audit_login authaudit(S)
audit_no_resource authaudit(S)
audit_open audit(S)
audit_passwd authaudit(S)
audit_read audit(S)
audit_security_failure authaudit(S)
audits audits(ADM)
audit_subsystem authaudit(S)
authaudit authaudit(S)
authcap authcap(F)
authcap authcap(S)
authck authck(ADM)
authckrc tcbck(ADM)
authorize authorize(F)
authorized_user subsystems(S)
auths auths(C)
authsh authsh(ADM)
autoboot autoboot(ADM)
awk awk(C)
backup backup(ADM)
backupsh backupsh(ADM)
badtrk badtrk(ADM)
banner banner(C)
basename basename(C)
batch at(C)
baudrate curses(S)
baudrate tam(S)
baudrate terminfo(S)
bc bc(C)
BC termcap(S)
bcheckrc bcheckrc(ADM)
bdiff bdiff(C)
beep curses(S)
beep tam(S)
beep terminfo(S)
bessel bessel(S)

bfs bfs(C)
bigcrypt getpasswd(S)
bigcryptmax getpasswd(S)
boot boot(HW)
Bottom libwindows(S)
bottom_panel panel(S)
box curses(S)
box plot(S)
box terminfo(S)
brand undocumented(M)
brk brk(S)
brkctl brkctl(S)
bsearch bsearch(S)
btld btld(F)
btldinstall btldinstall(ADM)
cal cal(C)
calendar calendar(C)
calloc malloc(S)
cancel cancel(C)
can_change_color curses(S)
can_change_color terminfo(S)
captinfo captinfo(ADM)
cat cat(C)
catclose catopen(S)
catgets catgets(S)
catopen catopen(S)
cb cb(CP)
cbackup cbackup(ADM)
cbreak curses(S)
cbreak tam(S)
cbreak terminfo(S)
cc cc(CP)
cd cd(C)
cdc cdc(CP)
cdrom cdrom(HW)
ceil floor(S)
cfgetispeed cfspeed(S)
cfgetospeed cfspeed(S)
cflow cflow(CP)
cfree malloc(S)
cfsetispeed cfspeed(S)
cfsetospeed cfspeed(S)
cfspeed cfspeed(S)
chargefee acctsh(ADM)
chdir chdir(S)
checkaddr checkaddr(ADM)
check_auth_parameters identity(S)
checkeq undocumented(M)
checklist checklist(F)
checkmail checkmail(C)
checkque checkque(ADM)

checkup	checkup (ADM)	comm	comm (C)
chg_audit	chg_audit (ADM)	compile	regexp (S)
chgrp	chgrp (C)	compress	compress (C)
chkshlib	chkshlib (CP)	compver	compver (F)
chmod	chmod (C)	configure	configure (ADM)
chmod	chmod (S)	console	console (M)
chown	chown (C)	consoleprint	consoleprint (ADM)
chown	chown (S)	cont	plot (S)
chroot	chroot (ADM)	conv	conv (CP)
chroot	chroot (S)	convert	convert (CP)
chrtbl	chrtbl (M)	convkey	mapkey (M)
chsize	chsize (S)	copy	copy (C)
chtype	unretire (ADM)	copydvagent	getdvagent (S)
circf	regexp (S)	copyright	copyright (F)
circle	plot (S)	copywin	curses (S)
ckbupscd	undocumented (M)	copywin	terminfo (S)
ckpacct	acctsh (ADM)	core	core (FP)
cksum	cksum (C)	corex	corex (C)
cleanque	cleanque (ADM)	cos	trig (S)
cleantmp	cleantmp (ADM)	cosh	sinh (S)
cleanup	undocumented (M)	cp	cp (C)
clear	clear (C)	cpio	cpio (C)
clear	curses (S)	cpio	cpio (F)
clear	tam (S)	cpp	cpp (CP)
clear	terminfo (S)	cprs	cprs (CP)
clearerr	ferror (S)	cps	fixmog (ADM)
clearok	curses (S)	cpset	cpset (C)
clearok	tam (S)	crash	crash (ADM)
clearok	terminfo (S)	creat	creat (S)
clock	clock (F)	create_file_securely ..	create_file_securely (S)
clock	clock (S)	creatsem	creatsem (S)
clone	clone (M)	crmode	curses (S)
close	close (S)	crmode	tam (S)
closedir	directory (S)	crmode	terminfo (S)
closepl	plot (S)	cron	cron (C)
clri	clri (ADM)	crontab	crontab (C)
clrtobot	curses (S)	crypt	crypt (C)
clrtobot	tam (S)	crypt	crypt (S)
clrtobot	terminfo (S)	crypt_close	crypt (S)
clrtoeol	curses (S)	cryptopen	crypt (S)
clrtoeol	tam (S)	cscope	cscope (CP)
clrtoeol	terminfo (S)	csh	csh (C)
cmos	cmos (HW)	csplit	csplit (C)
cmp	cmp (C)	ct	ct (C)
cnvtmbox	cnvtmbox (ADM)	ctags	ctags (C)
codeview	codeview (CP)	ctermid	ctermid (S)
col	col (C)	ctime	ctime (S)
color_content	curses (S)	ctrace	ctrace (CP)
color_content	terminfo (S)	ctype	ctype (S)
coltbl	coltbl (M)	cu	cu (C)
comb	comb (CP)	curoff	curses (S)

curoff	terminfo(S)	delta	delta (CP)
curon	curses (S)	delwin	curses (S)
curon	terminfo(S)	delwin	terminfo (S)
Current	libwindows (S)	depend	depend (F)
current_field	form (S)	des_crypt	crypt (S)
current_item	menu (S)	des_encrypt	crypt (S)
curses	curses (S)	des_setkey	crypt (S)
curs_set	curses (S)	devassign	devassign (F)
curs_set	terminfo(S)	devices	devices (F)
curtbl	monttbl (M)	devnm	devnm (C)
cuserid	cuserid (S)	df	df (C)
custom	custom (ADM)	dfsck	fsck (ADM)
cut	cut (C)	dfspace	dfspace (C)
cv	codeview (CP)	dial	dial (ADM)
cxref	cxref (CP)	dial	dial (S)
dat	dat (HW)	dialcodes	dialcodes (F)
date	date (C)	dialers	dialers (F)
daylight	ctime (S)	diff	diff (C)
dblock	dblock (S)	diff3	diff3 (C)
dbm	dbm (S)	difftime	difftime (S)
dbmbuild	dbmbuild (ADM)	dir	dir (FP)
dbmedit	dbmedit (ADM)	dircmp	dircmp (C)
dbminit	dbm (S)	directory	directory (S)
dbXtra	dbXtra (CP)	dirent	dirent (FP)
dc	dc (C)	dirname	dirname (C)
dcopy	dcopy (ADM)	dis	dis (CP)
dd	dd (C)	disable	disable (C)
deassign	assign (C)	diskcmp	diskcp (C)
debrand	undocumented (M)	diskcp	diskcp (C)
default	default (F)	diskusg	diskusg (ADM)
defaults	defaults (F)	displaypkg	displaypkg (ADM)
defopen	defopen (S)	div	div (S)
def_prog_mode	curses (S)	divvy	divvy (ADM)
def_prog_mode	terminfo (S)	dkinit	dparam (ADM)
defread	defopen (S)	dlvr_audit	dlvr_audit (ADM)
def_shell_mode	curses (S)	dmesg	dmesg (ADM)
def_shell_mode	terminfo (S)	dmp_win	curses (S)
delay_output	curses (S)	dodisk	acctsh (ADM)
delay_output	terminfo (S)	doscat	doscmd (C)
delch	curses (S)	doscmd	doscmd (C)
delch	tam (S)	doscp	doscmd (C)
delch	terminfo (S)	dosdir	doscmd (C)
del_curterm	curses (S)	dosformat	doscmd (C)
del_curterm	terminfo (S)	dosld	dosld (CP)
delete	dbm (S)	dosls	doscmd (C)
Delete	libwindows (S)	dosmkdir	doscmd (C)
deleteln	curses (S)	dosrm	doscmd (C)
deleteln	tam (S)	dosrmdir	doscmd (C)
deleteln	terminfo (S)	doupdate	curses (S)
deliver	deliver (ADM)	doupdate	terminfo (S)
del_panel	panel (S)	dparam	dparam (ADM)

draino	 curses (S)	erase	 plot (S)
draino	 terminfo (S)	erase	 tam (S)
drand48	 drand48 (S)	erase	 terminfo (S)
dsconfig	 undocumented (M)	erasechar	 curses (S)
dtox	 dtox (C)	erasechar	 terminfo (S)
dtype	 dtype (C)	erf	 erf (S)
du	 du (C)	erfc	 erf (S)
dump	 dump (CP)	errno	 perror (S)
dumpmsg	 dumpmsg (CP)	error	 error (M)
dumpwin	 terminfo (S)	ERROR	 regexp (S)
dup	 dup (S)	etext	 end (S)
dup2	 dup2 (S)	ev_block	 ev_block (S)
dup_field	 field (S)	ev_close	 ev_close (S)
dupwin	 curses (S)	ev_count	 ev_count (S)
dupwin	 terminfo (S)	ev_flush	 ev_flush (S)
eaccess	 access (S)	ev_getdev	 ev_getdev (S)
ecc	 ecc (ADM)	ev_getemask	 ev_getemask (S)
eccd	 ecc (ADM)	ev_gindev	 ev_gindev (S)
echo	 curses (S)	ev_init	 ev_init (S)
echo	 echo (C)	ev_initf	 ev_init (S)
echo	 tam (S)	ev_open	 ev_open (S)
echo	 terminfo (S)	ev_pop	 ev_pop (S)
echochar	 curses (S)	ev_read	 ev_read (S)
echochar	 terminfo (S)	ev_resume	 ev_resume (S)
ecvt	 ecvt (S)	ev_setemask	 ev_setemask (S)
ed	 ed (C)	ev_suspend	 ev_suspend (S)
edata	 end (S)	ex	 ex (C)
edit	 ex (C)	exec	 exec (S)
egrep	 grep (C)	execl	 exec (S)
eisa	 eisa (ADM)	execle	 exec (S)
emactovi	 undocumented (M)	execlp	 exec (S)
enable	 enable (C)	execseg	 execseg (S)
encrypt	 crypt (S)	execv	 exec (S)
end	 end (S)	execve	 exec (S)
enddvagent	 getdvagent (S)	execvp	 exec (S)
endgrent	 getgrent (S)	exhelp	 tam (S)
endprdfent	 getprdfent (S)	exit	 exit (S)
endprfient	 getprfient (S)	Exit	 libwindows (S)
endprpwent	 getprpwent (S)	exit_quiet_zone	 dblock (S)
endprtcent	 getprtcent (S)	exp	 exp (S)
endpwent	 getpwent (S)	expand	 expand (C)
endspent	 getspent (S)	expr	 expr (C)
endutent	 getut (S)	fabs	 floor (S)
endwin	 curses (S)	factor	 factor (C)
endwin	 tam (S)	false	 false (C)
endwin	 terminfo (S)	fclose	 fclose (S)
enter_quiet_zone	 dblock (S)	fcntl	 fcntl (M)
env	 env (C)	fcntl	 fcntl (S)
environ	 environ (M)	fcvt	 ecvt (S)
erand48	 drand48 (S)	fd	 fd (HW)
erase	 curses (S)	FD_CLR	 select (S)

fdisk	fdisk (ADM)	fixhdr	fixhdr (C)
FD_ISSET	select (S)	fixmog	fixmog (ADM)
fdopen	fopen (S)	fixperm	fixperm (ADM)
FD_SET	select (S)	fixshlib	undocumented (M)
fdswap	fdswap (ADM)	fixterm	curses (S)
FD_ZERO	select (S)	fixterm	tam (S)
feof	ferror (S)	flash	curses (S)
ferror	ferror (S)	flash	tam (S)
fetch	dbm (S)	flash	terminfo (S)
ff	ff (ADM)	floor	floor (S)
fflush	fclose (S)	flushinp	curses (S)
fgetc	getc (S)	flushinp	tam (S)
fgetgrent	getgrent (S)	flushinp	terminfo (S)
fgetpasswd	getpasswd (S)	fmod	floor (S)
fgetpos	fgetpos (S)	fold	fold (C)
fgetpwent	getpwent (S)	fopen	fopen (S)
fgets	gets (S)	fork	fork (S)
fgetspent	getspent (S)	form	form (S)
fgrep	grep (C)	form	tam (S)
field	field (S)	format	format (C)
field_arg	field (S)	form_driver	form (S)
field_back	field (S)	form_fields	form (S)
field_buffer	field (S)	form_init	form (S)
field_count	form (S)	form_opts	form (S)
field_fore	field (S)	form_opts_off	form (S)
field_index	form (S)	form_opts_on	form (S)
field_info	field (S)	form_page	form (S)
field_init	form (S)	form_sub	form (S)
field_just	field (S)	form_term	form (S)
field_opts	field (S)	form_userptr	form (S)
field_opts_off	field (S)	form_win	form (S)
field_opts_on	field (S)	fpathconf	pathconf (S)
field_pad	field (S)	fpgetmask	fpgetround (S)
fields	fields (S)	fpgetround	fpgetround (S)
field_status	field (S)	fpgetsticky	fpgetround (S)
field_term	form (S)	fprintf	printf (S)
field_type	field (S)	fpsetmask	fpgetround (S)
fieldtype	fieldtype (S)	fpsetround	fpgetround (S)
field_userptr	field (S)	fpsetsticky	fpgetround (S)
file	file (C)	fputc	putc (S)
filehdr	filehdr (FP)	fputs	puts (S)
fileno	ferror (S)	fread	fread (S)
files	files (F)	frec	undocumented (M)
filesys	filesys (F)	free	malloc (S)
filesystem	filesystem (FP)	free_field	field (S)
filter	curses (S)	free_fieldtype	fieldtype (S)
filter	terminfo (S)	free_form	form (S)
find	find (C)	free_item	item (S)
findstr	findstr (CP)	freeldptr	ldptr (S)
finger	finger (C)	free_menu	menu (S)
firstkey	dbm (S)	freopen	fopen (S)

frexp	frexp (S)	geteuid	geteuid (S)
fsanck	undocumented (M)	getgid	getuid (S)
fsave	fsave (ADM)	getgrent	getgrent (S)
fsba	undocumented (M)	getgrgid	getgrent (S)
fscanf	scanf (S)	getgrnam	getgrent (S)
fsck	fsck (ADM)	getgroups	getgroups (S)
fsdb	fsdb (ADM)	gethz	gethz (S)
fseek	fseek (S)	getitimer	getitimer (S)
fsetpos	fsetpos (S)	getlogin	getlogin (S)
fsname	fsname (ADM)	getluid	getluid (S)
fspec	fspec (F)	getmaxyx	curses (S)
fsphoto	fsphoto (ADM)	getmaxyx	terminfo (S)
fsstat	fsstat (ADM)	getmsg	getmsg (S)
_fst	undocumented (M)	getopt	getopt (C)
fstat	stat (S)	getopt	getopt (S)
fstatfs	statfs (S)	getoptcv	getopts (C)
fstyp	fstyp (ADM)	getopts	getopts (C)
fsync	fsync (S)	getorg	curses (S)
ftell	fseek (S)	getorg	terminfo (S)
ftime	time (S)	getpass	getpass (S)
ftok	ftok (S)	getpasswd	getpasswd (S)
ftw	ftw (S)	getpgrp	getpid (S)
fuser	fuser (ADM)	getpid	getpid (S)
fwrite	fread (S)	getppid	getpid (S)
fwtmp	fwtmp (ADM)	getprdfent	getprdfent (S)
fxlist	xlist (S)	getprdfnam	getprdfent (S)
gamma	gamma (S)	getprfient	getprfient (S)
garbagedlines	curses (S)	getprfinam	getprfient (S)
garbagedlines	terminfo (S)	getpriv	getpriv (S)
gcvt	ecvt (S)	getprpwent	getprpwent (S)
gencat	gencat (CP)	getprpwnam	getprpwent (S)
gencc	gencc (CP)	getprpwuid	getprpwent (S)
get	get (CP)	getprtcent	getprtcent (S)
getbegyx	curses (S)	getprtcnam	getprtcent (S)
getbegyx	terminfo (S)	getpw	getpw (S)
getc	getc (S)	getpwent	getpwent (S)
GETC	regex (S)	getpwnam	getpwent (S)
getch	curses (S)	getpwuid	getpwent (S)
getch	tam (S)	gets	gets (C)
getch	terminfo (S)	gets	gets (S)
getchar	getc (S)	get_seed	seed (S)
getclk	getclk (M)	getsermo	getsermo (C)
getconf	getconf (C)	getspent	getspent (S)
getcwd	getcwd (S)	getspnam	getspent (S)
getdents	getdents (S)	getstr	curses (S)
getdim	curses (S)	getstr	terminfo (S)
getdim	terminfo (S)	getsyx	curses (S)
getdvagent	getdvagent (S)	getsyx	terminfo (S)
getdvagname	getdvagent (S)	gettmode	curses (S)
getegid	getuid (S)	getty	getty (M)
getenv	getenv (S)	gettydefs	gettydefs (F)

getuid **getuid** (S)
getut **getut** (S)
getutent **getut** (S)
getutid **getut** (S)
getutline **getut** (S)
getw **getc** (S)
getyx **curses** (S)
getyx **tam** (S)
getyx **terminfo** (S)
gmtime **ctime** (S)
goodpw **goodpw** (ADM)
graph **graph** (ADM)
greek **greek** (C)
grep **grep** (C)
gr_idtoname **pw_nametoid** (S)
gr_nametoid **pw_nametoid** (S)
group **group** (F)
grpck **grpck** (ADM)
gsignal **ssignal** (S)
halfdelay **curses** (S)
halfdelay **terminfo** (S)
haltsys **haltsys** (ADM)
has_colors **curses** (S)
has_colors **terminfo** (S)
hashcheck **spell** (C)
hashmake **spell** (C)
has_ic **curses** (S)
has_ic **terminfo** (S)
has_il **curses** (S)
has_il **terminfo** (S)
hcreate **hsearch** (S)
hd **hd** (C)
hd **hd** (HW)
hdestroy **hsearch** (S)
hdr **hdr** (XNX)
head **head** (C)
hello **hello** (C)
help **help** (CP)
hide_panel **panel** (S)
hp **hp** (C)
hs **hs** (F)
hsearch **hsearch** (S)
hwconfig **hwconfig** (C)
hypot **hypot** (S)
i286 **machid** (C)
i286emul **i286emul** (C)
i286emul **i286emul** (CP)
i386 **machid** (C)
i486 **machid** (C)
iAPX286 **machid** (C)
ibmloppt **undocumented** (M)

iconv **iconv** (CP)
id **id** (C)
idaddld **idaddld** (ADM)
idas **idas** (M)
idbuild **idbuild** (ADM)
idcheck **idcheck** (ADM)
idconfig **idbuild** (ADM)
identity **identity** (S)
idinstall **idinstall** (ADM)
idld **idld** (M)
idleout **idleout** (ADM)
idlok **curses** (S)
idlok **terminfo** (S)
idmkenv **idbuild** (ADM)
idmkinit **idmkinit** (ADM)
idmknod **idmknod** (ADM)
idmkunix **idbuild** (ADM)
idscsi **idbuild** (ADM)
idspace **idspace** (ADM)
idtune **idtune** (ADM)
idvidi **idbuild** (ADM)
inch **curses** (S)
inch **terminfo** (S)
infocmp **infocmp** (ADM)
inipcrm **undocumented** (M)
init **init** (M)
INIT **regex** (S)
init.base **inittab** (F)
init_color **curses** (S)
init_color **terminfo** (S)
initcond **initcond** (ADM)
init_pair **curses** (S)
init_pair **terminfo** (S)
initscr **curses** (S)
initscr **tam** (S)
initscr **terminfo** (S)
initscript **initscript** (ADM)
inittab **inittab** (F)
inode **inode** (FP)
insch **curses** (S)
insch **tam** (S)
insch **terminfo** (S)
insertln **curses** (S)
insertln **tam** (S)
insertln **terminfo** (S)
insertmsg **insertmsg** (CP)
install **install** (ADM)
installf **installf** (ADM)
installpkg **installpkg** (ADM)
integrity **integrity** (ADM)
intrflush **curses** (S)

intrflush	terminfo (S)	is_starting_euid	identity (S)
Intro	Intro (ADM)	is_starting_luid	identity (S)
Intro	Intro (C)	is_starting_rgid	identity (S)
Intro	Intro (CP)	is_starting_ruid	identity (S)
Intro	Intro (F)	issue	issue (F)
Intro	Intro (FP)	isuniqueid	isuniqueid (S)
Intro	Intro (HW)	isunlock	isunlock (S)
Intro	Intro (M)	isupper	ctype (S)
Intro	Intro (S)	isverify	isverify (M)
Intro	Intro (XNX)	iswind	tam (S)
ioctl	ioctl (S)	iswcurr	iswcurr (S)
ipcrm	ipcrm (ADM)	iswrite	iswrite (S)
ipcs	ipcs (ADM)	isxdigit	ctype (S)
isaddindex	isaddindex (S)	item	item (S)
isalnum	ctype (S)	item_count	item (S)
isalpha	ctype (S)	item_description	item (S)
isascii	ctype (S)	item_index	menu (S)
isatty	ttyname (S)	item_init	menu (S)
isbuild	isbuild (S)	item_name	item (S)
isclose	isclose (S)	item_opts	item (S)
iscntrl	ctype (S)	item_opts_off	item (S)
isconv	isconv (S)	item_opts_on	item (S)
isdelcurr	isdelcurr (S)	item_term	menu (S)
isdelete	isdelete (S)	item_userptr	item (S)
isdelindex	isdelindex (S)	item_value	item (S)
isdelrec	isdelrec (S)	item_visible	item (S)
isdigit	ctype (S)	itimer	getitimer (S)
isendwin	curses (S)	j0	bessel (S)
isendwin	terminfo (S)	j1	bessel (S)
iserase	iserase (S)	jagent	jagent (M)
isgraph	ctype (S)	jn	bessel (S)
isindexinfo	isindexinfo (S)	join	join (C)
islock	islock (S)	jrand48	drand48 (S)
islower	ctype (S)	jterm	jterm (C)
ismpx	ismpx (C)	jwin	jwin (C)
isnan	isnan (S)	kbmode	kbmode (ADM)
isnand	isnan (S)	kcodelmap	tam (S)
isnanf	isnan (S)	keyboard	keyboard (HW)
isopen	isopen (S)	keyname	curses (S)
isprint	ctype (S)	keyname	terminfo (S)
ispunct	ctype (S)	keypad	curses (S)
isread	isread (S)	keypad	tam (S)
isrelease	isrelease (S)	keypad	terminfo (S)
isrename	isrename (S)	kill	kill (C)
isrewcurr	isrewcurr (S)	kill	kill (S)
isrewrec	isrewrec (S)	killall	killall (ADM)
isrewrite	isrewrite (S)	killchar	curses (S)
issetunique	issetunique (S)	killchar	terminfo (S)
isspace	ctype (S)	kmem	mem (FP)
isstart	isstart (S)	ksh	ksh (C)
is_starting_egid	identity (S)	l	ls (C)

l3tol **l3tol** (S)
l64a **a64l** (S)
label **plot** (S)
labelit **labelit** (ADM)
labs **labs** (S)
langinfo **langinfo** (FP)
last **last** (C)
lastlogin **acctsh** (ADM)
layers **layers** (C)
layers **layers** (M)
lc **ls** (C)
lckpwdf **getspent** (S)
lcong48 **drand48** (S)
lconv **lconv** (FP)
ld **ld** (CP)
ld **ld** (XNX)
ldaclose **ldclose** (S)
ldahread **ldahread** (S)
ldaopen **ldopen** (S)
ldclose **ldclose** (S)
lddbl **isconv** (S)
ldexp **frexp** (S)
ldfcn **ldfcn** (FP)
ldfhread **ldfhread** (S)
ldfloat **isconv** (S)
ldgetname **ldgetname** (S)
ldint **isconv** (S)
ldiv **ldiv** (S)
ldlinit **ldlread** (S)
ldlitem **ldlread** (S)
ldlong **isconv** (S)
ldlread **ldlread** (S)
ldlseek **ldlseek** (S)
ldnlseek **ldlseek** (S)
ldnrseek **ldrseek** (S)
ldnshread **ldshread** (S)
ldnsseek **ldsseek** (S)
ldohseek **ldohseek** (S)
ldopen **ldopen** (S)
ldrseek **ldrseek** (S)
ldshread **ldshread** (S)
ldsseek **ldsseek** (S)
ldsysdump **ldsysdump** (ADM)
ldtbindx **ldtbindx** (S)
ldtbread **ldtbread** (S)
ldtbseek **ldtbseek** (S)
leaveok **curses** (S)
leaveok **tam** (S)
leaveok **terminfo** (S)
lex **lex** (CP)
lf **ls** (C)

lfind **lsearch** (S)
lgamma **gamma** (S)
libwindows **libwindows** (S)
limits **limits** (FP)
line **line** (C)
line **plot** (S)
linemod **plot** (S)
linenum **linenum** (FP)
link **link** (ADM)
link **link** (S)
link_field **field** (S)
link_fieldtype **field type** (S)
link_unix **link_unix** (ADM)
lint **lint** (CP)
list **list** (ADM)
list **list** (CP)
ll_close **llog** (S)
ll_err **llog** (S)
ll_hdinit **llog** (S)
ll_init **llog** (S)
ll_log **llog** (S)
llog **llog** (S)
ll_open **llog** (S)
ln **ln** (C)
__loc1 **regcmp** (S)
loc1 **regexp** (S)
loc2 **regexp** (S)
locale **locale** (C)
locale **locale** (M)
localeconv **localeconv** (S)
localedef **localedef** (C)
localtime **ctime** (S)
lock **lock** (C)
lock **lock** (S)
locked_out **fields** (S)
lockf **lockf** (S)
locking **locking** (S)
locs **regexp** (S)
log **exp** (S)
log **log** (HW)
log **log** (M)
log10 **exp** (S)
login **login** (M)
logname **logname** (C)
logname **logname** (S)
logs **logs** (F)
longjmp **setjmp** (S)
longname **curses** (S)
longname **terminfo** (S)
lorder **lorder** (CP)
lp **lp** (C)

lp	lp (HW)	mbtowc	mblen (S)
lp0	lp (HW)	mc68k	machid (C)
lp1	lp (HW)	mcart	mcart (C)
lp2	lp (HW)	mcconfig	mcconfig (F)
lpadmin	lpadmin (ADM)	mcdaemon	mcconfig (F)
lpfilter	lpfilter (ADM)	mcs	mcs (CP)
lpforms	lpforms (ADM)	mdevice	mdevice (F)
lpmove	lpmove (ADM)	mem	mem (FP)
lpr	lp (C)	memccpy	memory (S)
lprint	lprint (C)	memchr	memory (S)
lprof	lprof (CP)	memcmp	memory (S)
lpsched	lpsched (ADM)	memcpy	memory (S)
lpsh	lpsh (ADM)	memmove	memmove (S)
lpshut	lpsched (ADM)	memory	memory (S)
lpstat	lpstat (C)	memset	memory (S)
lpusers	lpusers (ADM)	memsize	memsize (ADM)
lr	ls (C)	menu	menu (S)
lrnd48	drand48 (S)	menu	tam (S)
ls	ls (C)	menu_add	undocumented (M)
lsearch	lsearch (S)	menu_back	menu (S)
lseek	lseek (S)	menu_del	undocumented (M)
lstat	stat (S)	menu_driver	menu (S)
ltol3	l3tol (S)	menu_fore	menu (S)
lx	ls (C)	menu_format	menu (S)
m4	m4 (CP)	menu_grey	menu (S)
machid	machid (C)	menu_init	menu (S)
mail	mail (C)	menu_items	menu (S)
maildelivery	maildelivery (F)	menu_mark	menu (S)
mailx	mail (C)	menumerge	menumerge (ADM)
majorinuse	majorinuse (ADM)	menu_opts	menu (S)
make	make (CP)	menu_opts_off	menu (S)
makekey	makekey (ADM)	menu_opts_on	menu (S)
make_transition_files	dblock (S)	menu_pad	menu (S)
mallinfo	mallinfo (FP)	menu_pattern	menu (S)
mallinfo	malloc (S)	menu_sub	menu (S)
malloc	malloc (S)	menu_term	menu (S)
mallopt	malloc (S)	menu_userptr	menu (S)
man	man (C)	menu_win	menu (S)
mapchan	mapchan (F)	mesg	mesg (C)
mapchan	mapchan (M)	message	tam (S)
mapkey	mapkey (M)	message	undocumented (M)
mapscrn	mapkey (M)	messages	messages (M)
mapstr	mapkey (M)	mestbl	mestbl (M)
mar	mar (CP)	meta	curses (S)
masm	masm (CP)	meta	terminfo (S)
math	math (M)	mfsys	mfsys (FP)
matherr	matherr (S)	mkdev	mkdev (ADM)
maxuuscheds	maxuuscheds (F)	mkdir	mkdir (C)
maxuuxqts	maxuuxqts (F)	mkdir	mkdir (S)
mblen	mblen (S)	mkfifo	mkfifo (C)
mbstowcs	mblen (S)	mkfifo	mkfifo (S)

mkfs mkfs (ADM)
mknod mknod (C)
mknod mknod (S)
mkshlib mkshlib (CP)
mkstr mkstr (CP)
mktemp mktemp (S)
mktime mktime (S)
ml_1adr ml_send (S)
ml_adr ml_send (S)
ml_aend ml_send (S)
ml_cc ml_send (S)
ml_end ml_send (S)
ml_file ml_send (S)
ml_init ml_send (S)
mlist undocumented (M)
ml_send ml_send (S)
ml_tinit ml_send (S)
ml_to ml_send (S)
ml_txt ml_send (S)
mmdf mmdf (ADM)
mmdf mmdf (S)
mmdfalias mmdfalias (ADM)
mmdftailor mmdftailor (F)
mm_end mmdf (S)
mm_init mmdf (S)
mm_pkend mmdf (S)
mm_pkinit mmdf (S)
mm_radr mmdf (S)
mm_rinit mmdf (S)
mm_rrec mmdf (S)
mm_rrply mmdf (S)
mm_rstm mmdf (S)
mm_rtxt mmdf (S)
mm_sbend mmdf (S)
mm_sbinit mmdf (S)
mm_wadr mmdf (S)
mm_waend mmdf (S)
mm_winit mmdf (S)
mm_wrec mmdf (S)
mm_wrply mmdf (S)
mm_wstm mmdf (S)
mm_wtend mmdf (S)
mm_wtxt mmdf (S)
mnlist mnlist (ADM)
mnt mnt (C)
mnttab mnttab (F)
modf frexp (S)
monacct acctsh (ADM)
monitor monitor (S)
montbl montbl (M)
more more (C)

mount mount (ADM)
mount mount (S)
mountall mountall (ADM)
mouse mouse (HW)
move curses (S)
Move libwindows (S)
move plot (S)
move tam (S)
move terminfo (S)
move_field field (S)
move_panel panel (S)
mrand48 drand48 (S)
mscreen mscreen (M)
mscsi mscsi (F)
msg msg (FP)
msgctl msgctl (S)
msgget msgget (S)
msgop msgop (S)
msgrcv msgop (S)
msgsnd msgop (S)
mt undocumented (M)
mtune mtune (F)
multiscreen multiscreen (M)
mv mv (C)
mvaddch curses (S)
mvaddch tam (S)
mvaddch terminfo (S)
mvaddstr curses (S)
mvaddstr tam (S)
mvaddstr terminfo (S)
mvcur curses (S)
mvcur terminfo (S)
mvdelch curses (S)
mvdelch terminfo (S)
mvdevice mvdevice (F)
mvdir mvdir (ADM)
mvgetch curses (S)
mvgetch terminfo (S)
mvgetstr curses (S)
mvgetstr terminfo (S)
mvinch curses (S)
mvinch tam (S)
mvinch terminfo (S)
mvinsch curses (S)
mvinsch terminfo (S)
mvprintw curses (S)
mvprintw terminfo (S)
mvscanw curses (S)
mvscanw terminfo (S)
mvwaddch curses (S)
mvwaddch terminfo (S)

mvwaddstr	curses (S)	nictable	nictable (ADM)
mvwaddstr	terminfo (S)	nl	curses(S)
mvwdelch	curses (S)	nl	nl (C)
mvwdelch	terminfo (S)	nl	tam (S)
mvwgetch	curses (S)	nl	terminfo (S)
mvwgetch	terminfo (S)	nl_ascxtime	nl_cxtime (S)
mvwgetstr	curses (S)	nl_cxtime	nl_cxtime (S)
mvwgetstr	terminfo (S)	nl_fprintf	nl_printf (S)
mvwin	curses (S)	nl_fscanf	nl_scanf (S)
mvwin	terminfo (S)	nl_init	nl_init (S)
mvwinch	curses (S)	nlist	nlist (S)
mvwinch	terminfo (S)	nl_langinfo	nl_langinfo (S)
mvwinsch	curses (S)	nl_printf	nl_printf (S)
mvwinsch	terminfo (S)	nlsadmin	nlsadmin (ADM)
mvwprintw	curses (S)	nl_scanf	nl_scanf (S)
mvwprintw	terminfo (S)	nl_sprintf	nl_printf (S)
mvwscanw	curses (S)	nl_sscanf	nl_scanf (S)
mvwscanw	terminfo (S)	nl_strcmp	nl_strcmp (S)
nap	nap (S)	nl_strerror	nl_strerror (S)
napms	curses (S)	nl_types	nl_types (FP)
napms	terminfo (S)	nm	nm (CP)
nawk	awk (C)	nm	nm (XNX)
nbra	regexp (S)	nocbreak	curses (S)
nbwaitsem	waitsem (S)	nocbreak	tam (S)
ncheck	ncheck (ADM)	nocbreak	terminfo (S)
netbuf	netbuf (FP)	nocrmode	curses (S)
netconfig	netconfig (ADM)	nocrmode	tam (S)
netutil	netutil (ADM)	nocrmode	terminfo (S)
New	libwindows (S)	nodelay	curses (S)
new_field	field (S)	nodelay	tam (S)
new_fieldtype	fieldtype (S)	nodelay	terminfo (S)
new_form	form (S)	noecho	curses (S)
newform	newform (C)	noecho	tam (S)
newgrp	newgrp (C)	noecho	terminfo (S)
new_item	item (S)	nohup	nohup (C)
Newlayer	libwindows (S)	nonl	curses (S)
newmail	undocumented (M)	nonl	tam (S)
new_menu	menu (S)	nonl	terminfo (S)
newpad	curses (S)	noraw	curses (S)
newpad	terminfo (S)	noraw	terminfo (S)
new_page	form (S)	notimeout	curses (S)
new_panel	panel (S)	notimeout	terminfo (S)
news	news (C)	nrand48	drand48 (S)
newterm	curses (S)	nssend	nssend (FP)
newterm	terminfo (S)	null	null (F)
newwin	curses (S)	null	null (FP)
newwin	terminfo (S)	nulladm	acctsh (ADM)
_nextchoice	fieldtype (S)	numtbl	numtbl (M)
nextkey	dbm (S)	oawk	awk (C)
nice	nice (C)	od	od (C)
nice	nice (S)	open	open (S)

openagent libwindows (S)
openchan libwindows (S)
opendir directory (S)
openpl plot (S)
opensem opensem (S)
optarg getopt (S)
opterr getopt (S)
optind getopt (S)
os2ld os2ld (CP)
ospeed termcap (S)
overlay curses (S)
overlay terminfo (S)
overwrite curses (S)
overwrite terminfo (S)
paccess paccess (S)
pack pack (C)
page more (C)
pair_content curses (S)
pair_content terminfo (S)
panel panel (S)
panel_above panel (S)
panel_below panel (S)
panel_hidden panel (S)
panel_userptr panel (S)
panel_window panel (S)
parallel parallel (HW)
passlen passlen (S)
passwd passwd (C)
passwd passwd (F)
passwd passwd (FP)
paste paste (C)
pathchk pathchk (C)
pathconf pathconf (S)
pause pause (S)
pax pax (C)
pb_check tam (S)
pb_empty tam (S)
pb_gbuf tam (S)
pb_gets tam (S)
pb_name tam (S)
pb_open tam (S)
pb_puts tam (S)
pb_seek tam (S)
pb_weof tam (S)
PC termcap (S)
pcat pack (C)
pclose popen (S)
pcpio pcpio (C)
pdp11 machid (C)
pechochar curses (S)
pechochar terminfo (S)

PEEKC regexp (S)
permissions permissions (F)
perms perms (F)
perror perror (S)
pg pg (C)
phs phs (S)
phs_get phs (S)
phs_msg phs (S)
phs_note phs (S)
pipe pipe (ADM)
pipe pipe (S)
pkgadd pkgadd (ADM)
pkgask pkgask (ADM)
pkgchk pkgchk (ADM)
pkginfo pkginfo (ADM)
pkginfo pkginfo (F)
pkgmap pkgmap (F)
pkgmk pkgmk (ADM)
pkgparam pkgparam (ADM)
pkgproto pkgproto (ADM)
pkgrm pkgrm (ADM)
pkgtrans pkgtrans (ADM)
plock plock (S)
plot plot (FP)
plot plot (S)
pnch pnch (FP)
pnoutrefresh curses (S)
pnoutrefresh terminfo (S)
point plot (S)
Poll poll (F)
poll poll (S)
Poll.day poll (F)
Poll.hour poll (F)
popen popen (S)
pos_form_cursor form (S)
pos_menu_cursor menu (S)
post_form form (S)
post_menu menu (S)
pow exp (S)
pr pr (C)
prctmp acctsh (ADM)
prdaily acctsh (ADM)
prefresh curses (S)
prefresh terminfo (S)
_prevchoice field type (S)
prf prf (HW)
prfdc profiler (ADM)
prfld profiler (ADM)
prfpr profiler (ADM)
prfsnap profiler (ADM)
prfstat profiler (ADM)

primary_auth	subsystems (S)	pw_nametoid	pw_nametoid (S)
primary_of_secondary_auth		pwunconv	pwconv (ADM)
.....	subsystems (S)	qsort	qsort (S)
printenv	env (C)	queue	queue (F)
printf	printf (S)	queuedefs	queuedefs (F)
printw	curses (S)	quot	quot (C)
printw	tam (S)	raise	raise (S)
printw	terminfo (S)	ramdisk	ramdisk (HW)
proc	proc (FP)	rand	rand (S)
proct1	proct1 (S)	random	random (C)
prof	prof (CP)	randomword	randomword (S)
prof	prof (M)	ranlib	ranlib (XNX)
prof	prof (XNX)	raw	curses (S)
profil	profil (S)	raw	terminfo (S)
profile	profile (M)	rc0	rc0 (ADM)
profiler	profiler (ADM)	rc2	rc2 (ADM)
proto	proto (F)	rcc	rcc (CP)
prototype	prototype (F)	rcflow	rcflow (CP)
prpw	prpw (F)	rcp	rcp (C)
prs	prs (CP)	rcvalert	rcvalert (C)
prtacct	acctsh (ADM)	rcvfile	rcvfile (C)
prwarn	prwarn (C)	rcvprint	rcvprint (C)
ps	ps (C)	rcvtrip	rcvtrip (C)
pstat	pstat (C)	rcxref	rcxref (CP)
ptar	ptar (C)	rdchk	rdchk (S)
ptmx	ptmx (M)	read	read (S)
ptrace	ptrace (S)	readdir	directory (S)
pts???	ptmx (M)	readlink	readlink (S)
purge	purge (C)	read_pw_fields	fields (S)
purge	purge (F)	read_tc_fields	fields (S)
putc	putc (S)	realloc	malloc (S)
putchar	putc (S)	reboot	haltsys (ADM)
putdvagname	getdvagent (S)	red	ed (C)
putenv	putenv (S)	reduce	reduce (ADM)
putmsg	putmsg (S)	refresh	curses (S)
putp	curses (S)	refresh	tam (S)
putp	terminfo (S)	refresh	terminfo (S)
putprdfnam	getprdfent (S)	regcmp	regcmp (CP)
putprfinam	getprfient (S)	regcmp	regcmp (S)
putprpwnam	getprpwent (S)	regex	regcmp (S)
putprtcnam	getprtcnt (S)	regexp	regexp (S)
putpwent	putpwent (S)	reject	accept (ADM)
puts	puts (S)	relax	relax (ADM)
putspent	putspent (S)	reloc	reloc (FP)
pututline	getut (S)	relogin	relogin (ADM)
putw	putc (S)	remote	remote (C)
pwck	pwck (ADM)	remove	remove (S)
pwconv	pwconv (ADM)	removef	removef (ADM)
pwd	pwd (C)	removepkg	removepkg (ADM)
pwdmenu	undocumented (M)	rename	rename (S)
pw_idtoname	pw_nametoid (S)	repackman	repackman (ADM)

replace_file dblock(S)
 replace_panel panel(S)
 resend resend(C)
 reset_prog_mode curses(S)
 reset_prog_mode terminfo(S)
 reset_shell_mode curses(S)
 reset_shell_mode terminfo(S)
 resetterm curses(S)
 resettty curses(S)
 resettty tam(S)
 resettty terminfo(S)
 resetty curses(S)
 resetty tam(S)
 resetty terminfo(S)
 Reshape libwindows(S)
 restartterm curses(S)
 restartterm terminfo(S)
 restore restore(ADM)
 RETURN regexp(S)
 rewind fseek(S)
 rewinddir directory(S)
 ripoffline curses(S)
 ripoffline terminfo(S)
 rksh ksh(C)
 rlint rlint(CP)
 rlpCmd rlpCmd(C)
 rlpstat lpstat(C)
 rm rm(C)
 rmail rmail(ADM)
 rmb rmb(M)
 rmdel rmdel(CP)
 rmdir rmdir(C)
 rmdir rmdir(S)
 rmgroup rmuser(ADM)
 rmpasswd rmuser(ADM)
 rmuser rmuser(ADM)
 Routines Routines(DOS)
 Routines Routines(S)
 rsh rsh(C)
 rstab undocumented(M)
 rtc rtc(HW)
 runacct acctsh(ADM)
 runacct runacct(ADM)
 run_crypt crypt(S)
 Runlayer libwindows(S)
 run_setkey crypt(S)
 sa1 sar(ADM)
 sa2 sar(ADM)
 sact sact(CP)
 sadc sar(ADM)
 sag sag(ADM)

sar sar(ADM)
 saveterm curses(S)
 savetty curses(S)
 savetty tam(S)
 savetty terminfo(S)
 sbrk brk(S)
 scale_form form(S)
 scale_menu menu(S)
 scandcode scandcode(HW)
 scanf scanf(S)
 scanoff scanon(M)
 scanon scanon(M)
 scanw curses(S)
 scanw terminfo(S)
 sc_copystate sc_raw(S)
 sccsdiff sccsdiff(CP)
 sccsfile sccsfile(FP)
 sc_exit sc_init(S)
 sc_getfkeystr sc_init(S)
 sc_getinfo sc_raw(S)
 sc_getkbmap sc_init(S)
 sc_getkeymap sc_init(S)
 sc_getled sc_init(S)
 sc_getscreenswitch sc_raw(S)
 schedule schedule(ADM)
 sc_init sc_init(S)
 sc_kb2mapcode sc_readkb(S)
 sc_mapcode2kb sc_readkb(S)
 sc_mapcode2str sc_readkb(S)
 sc_mapin sc_readkb(S)
 sc_mapinit sc_init(S)
 sc_mapout sc_readkb(S)
 scnhdr scnhdr(FP)
 __scoinfo __scoinfo(S)
 scosh scosh(C)
 sc_raw sc_raw(S)
 scr_dump curses(S)
 scr_dump scr_dump(FP)
 scr_dump terminfo(S)
 sc_readkb sc_readkb(S)
 sc_readmapcode sc_readkb(S)
 sc_readstr sc_readkb(S)
 sc_receive_kb sc_init(S)
 screen screen(HW)
 scr_init curses(S)
 scr_init terminfo(S)
 scroll curses(S)
 scroll terminfo(S)
 scrollok curses(S)
 scrollok terminfo(S)
 scr_restore curses(S)

scr_restore	terminfo (S)	set_field_pad	field (S)
sc_setfkeystr	sc_init (S)	set_field_status	field (S)
sc_setinfo	sc_raw (S)	set_field_term	form (S)
sc_setkeymap	sc_init (S)	set_field_type	field (S)
sc_setled	sc_init (S)	set_fieldtype_arg	fieldtype (S)
sc_setscreenswitch	sc_raw (S)	set_fieldtype_choice	fieldtype (S)
scsi	scsi (HW)	set_field_userptr	field (S)
scsibadblk	scsibadblk (ADM)	set_form_fields	form (S)
sc_str2kb	sc_readkb (S)	set_form_init	form (S)
sc_unraw	sc_raw (S)	set_form_opts	form (S)
sd	sd (ADM)	set_form_page	form (S)
sdb	sdb (CP)	set_form_sub	form (S)
sdd	sd (ADM)	set_form_term	form (S)
sddate	sddate (C)	set_form_userptr	form (S)
sdenter	sdenter (S)	set_form_win	form (S)
sdevice	sdevice (F)	setgid	setuid (S)
sdfree	sdget (S)	setgrent	getgrnt (S)
sdget	sdget (S)	setgroups	setgroups (S)
sdgetv	sdgetv (S)	set_item_init	menu (S)
sdiff	sdiff (C)	set_item_opts	item (S)
sdleave	sdenter (S)	set_item_term	menu (S)
sdwaitv	sdgetv (S)	set_item_userptr	item (S)
secondary_auth	subsystems (S)	set_item_value	item (S)
sed	regexp (S)	setitimer	getitimer (S)
sed	sed (C)	setjmp	setjmp (S)
seed	seed (S)	setkey	crypt (S)
seed48	drand48 (S)	setkey	setkey (C)
seekdir	directory (S)	setlocale	setlocale (S)
select	select (S)	setluid	setluid (S)
sem	sem (FP)	set_menu_back	menu (S)
semctl	semctl (S)	set_menu_fore	menu (S)
semget	semget (S)	set_menu_format	menu (S)
semop	semop (S)	set_menu_grey	menu (S)
serial	serial (HW)	set_menu_init	menu (S)
set_auth_parameters	identity (S)	set_menu_items	menu (S)
setbuf	setbuf (S)	set_menu_mark	menu (S)
setclk	undocumented (M)	set_menu_opts	menu (S)
setclock	setclock (ADM)	set_menu_pad	menu (S)
setcolor	setcolor (C)	set_menu_pattern	menu (S)
setcolour	setcolor (C)	set_menu_sub	menu (S)
set_current_field	form (S)	set_menu_term	menu (S)
set_current_item	menu (S)	set_menu_userptr	menu (S)
set_curterm	curses (S)	set_menu_win	menu (S)
set_curterm	terminfo (S)	setmnt	setmnt (ADM)
setdvagent	getdvagent (S)	set_new_page	form (S)
set_field_back	field (S)	set_panel_userptr	panel (S)
set_field_buffer	field (S)	setpgid	setpgid (S)
set_field_fore	field (S)	setpgrp	setpgrp (S)
set_field_init	form (S)	setprdfent	getprdfent (S)
set_field_just	field (S)	setprfient	getprfient (S)
set_field_opts	field (S)	setpriv	setpriv (S)

setprpwent getprpwent (S)
 setprtcent getprtcent (S)
 setpwent getpwent (S)
 setscrrcg curses (S)
 setscrreg terminfo (S)
 set_seed seed (S)
 setsid setsid (S)
 setspent getspent (S)
 setsyx curses (S)
 setsyx terminfo (S)
 set_term curses (S)
 set_term terminfo (S)
 settime settime (ADM)
 set_top_row menu (S)
 set_tty curses (S)
 set_tty terminfo (S)
 setuid setuid (S)
 setupterm curses (S)
 setupterm terminfo (S)
 setutent getut (S)
 setvbuf setbuf (S)
 sfmt sfmt (ADM)
 sfsys sfsys (FP)
 sg sg (C)
 sgetl sputl (S)
 sh sh (C)
 shadow shadow (F)
 shl shl (C)
 shm shm (FP)
 shmat shmop (S)
 shmctl shmctl (S)
 shmdt shmop (S)
 shmget shmget (S)
 shmop shmop (S)
 show_panel panel (S)
 shutacct acctsh (ADM)
 shutdown shutdown (S)
 shutdown shutdown (ADM)
 sigaction sigaction (S)
 sigaddset sigset (S)
 sigdelset sigset (S)
 sigemptyset sigset (S)
 sigfillset sigset (S)
 sighold sigsetv (S)
 sigignore sigsetv (S)
 sigismember sigset (S)
 siglongjmp sigsetjmp (S)
 signal signal (S)
 siggam gamma (S)
 sigpause sigsetv (S)
 sigpending sigpending (S)

sigprocmask sigprocmask (S)
 sigrelse sigsetv (S)
 sigsem sigsem (S)
 sigset sigset (S)
 sigset sigsetv (S)
 sigsetjmp sigsetjmp (S)
 sigsuspend sigsuspend (S)
 sin trig (S)
 sinh sinh (S)
 S_ISBLK stat (S)
 S_ISCHR stat (S)
 S_ISDIR stat (S)
 S_ISFIFO stat (S)
 S_ISNAM stat (S)
 S_ISREG stat (S)
 size size (CP)
 size size (XNX)
 sleep sleep (C)
 sleep sleep (S)
 slk_clear curses (S)
 slk_clear terminfo (S)
 slk_init curses (S)
 slk_init terminfo (S)
 slk_label curses (S)
 slk_label terminfo (S)
 slk_noutrefresh curses (S)
 slk_noutrefresh terminfo (S)
 slk_refresh curses (S)
 slk_refresh terminfo (S)
 slk_restore curses (S)
 slk_restore terminfo (S)
 slk_set curses (S)
 slk_set terminfo (S)
 slk_touch curses (S)
 slk_touch terminfo (S)
 slot slot (C)
 smmck tcbck (ADM)
 sort sort (C)
 space plot (S)
 space space (F)
 spell spell (C)
 spellin spell (C)
 spline spline (C)
 split split (C)
 sprintf printf (S)
 sputl sputl (S)
 sqrt exp (S)
 srand rand (S)
 srand48 drand48 (S)
 sscanf scanf (S)
 ssignal ssignal (S)

standend	curses (S)	strncat	string (S)
standend	terminfo (S)	strncmp	string (S)
standout	curses (S)	strncoll	strcoll (S)
standout	terminfo (S)	strncpy	string (S)
start_color	curses (S)	strnxfm	strcoll (S)
start_color	terminfo (S)	strpbrk	string (S)
starting_egid	identity (S)	strchr	string (S)
starting_euid	identity (S)	strspn	string (S)
starting_luid	identity (S)	strstr	string (S)
starting_rgid	identity (S)	strtod	strtod (S)
starting_ruid	identity (S)	strtok	string (S)
startup	acctsh (ADM)	strtol	strtol (S)
stat	stat (FP)	strtoul	strtoul (S)
stat	stat (S)	strxfm	strcoll (S)
statfs	statfs (S)	STTY	stty (C)
statlstat	stat (S)	stune	stune (F)
stdarg	varargs (S)	su	su (C)
stdbl	isconv (S)	submit	submit (ADM)
stderr	stdio (S)	subpad	curses (S)
stdin	stdio (S)	subpad	terminfo (S)
stdio	stdio (S)	subsystem	subsystem (M)
stdout	stdio (S)	subsystems	subsystems (S)
step	regex (S)	subwin	curses (S)
stfloat	isconv (S)	subwin	terminfo (S)
stime	stime (S)	sulogin	sulogin (ADM)
stint	isconv (S)	sum	sum (C)
stlong	isconv (S)	swab	swab (S)
stopio	stopio (S)	swap	swap (ADM)
store	dbm (S)	swconfig	swconfig (C)
store_pw_fields	fields (S)	sxt	sxt (M)
store_tc_fields	fields (S)	symlink	symlink (S)
strace	strace (ADM)	syms	syms (FP)
strcat	string (S)	sync	sync (ADM)
strchr	string (S)	sync	sync (S)
strclean	strclean (ADM)	sysadmcolor	sysadmcolor (F)
strcmp	string (S)	sysadmmenu	sysadmmenu (F)
strcoll	strcoll (S)	sysadmsh	sysadmsh (ADM)
strcpy	string (S)	sysconf	sysconf (S)
strcspn	string (S)	sysdef	sysdef (ADM)
strdup	string (S)	sys_errlist	perror (S)
streamio	streamio (M)	sysfiles	sysfiles (F)
strerr	strerr (ADM)	sysfs	sysfs (S)
strerror	strerror (S)	sysi86	sysi86 (S)
strftime	ctime (S)	sys_nerr	perror (S)
strftime	strftime (S)	system	system (S)
string	string (M)	systemid	systemid (F)
string	string (S)	systems	systems (F)
strings	strings (C)	sys tty	sys tty (M)
strip	strip (CP)	tables	tables (F)
strip	strip (XNX)	tabs	tabs (C)
strlen	string (S)	t_accept	t_accept (S)

tai_end	tai (S)	tgetent	terminfo (S)
tai_get	tai (S)	tgetflag	curses (S)
tai_init	tai (S)	tgetflag	termcap (S)
tail	tail (C)	tgetflag	terminfo (S)
t_alloc	t_alloc (S)	t_getinfo	t_getinfo (S)
tam	tam (S)	tgetnum	curses (S)
tan	trig (S)	tgetnum	termcap (S)
tanh	sinh (S)	tgetnum	terminfo (S)
tape	tape (C)	t_getstate	t_getstate (S)
tape	tape (HW)	tgetstr	curses (S)
tapecntl	tapecntl (C)	tgetstr	termcap (S)
tapedump	tapedump (C)	tgetstr	terminfo (S)
tar	tar (C)	tgoto	curses (S)
tar	tar (F)	tgoto	termcap (S)
t_bind	t_bind (S)	tgoto	terminfo (S)
tcback	tcback (ADM)	tic	tic (C)
tcdrain	tcflow (S)	tigetflag	curses (S)
tcflow	tcflow (S)	tigetflag	terminfo (S)
tcflush	tcflow (S)	tigetnum	curses (S)
tcgetattr	tcattr (S)	tigetnum	terminfo (S)
tcgetpgrp	tcpggrp (S)	tigetstr	curses (S)
tcl	tcl (C)	tigetstr	terminfo (S)
t_close	t_close (S)	time	time (C)
t_connect	t_connect (S)	time	time (S)
tcsendbreak	tcflow (S)	times	times (S)
tcsetattr	tcattr (S)	timex	timex (ADM)
tcsetpgrp	tcpggrp (S)	timezone	ctime (S)
tdelete	tsearch (S)	timezone	timezone (F)
tee	tee (C)	timod	timod (M)
telinit	init (M)	timtbl	timtbl (M)
telldir	directory (S)	t_info	t_info (FP)
tempnam	tmpnam (S)	tirdwr	tirdwr (M)
term	term (M)	t_listen	t_listen (S)
termcap	termcap (F)	t_look	t_look (S)
termcap	termcap (S)	tmpfile	tmpfile (S)
terminal	terminal (HW)	tmpnam	tmpnam (S)
terminals	terminals (M)	t_nerr	t_error (S)
terminfo	terminfo (F)	toascii	ctype (S)
terminfo	terminfo (M)	toascii	toascii (S)
terminfo	terminfo (S)	todigit	toascii (S)
termio	termio (M)	toint	toascii (S)
termios	termios (M)	tolower	ctype (S)
termupd	ttyupd (ADM)	_tolower	toascii (S)
t_errlist	t_error (S)	Top	libwindows (S)
t_erno	t_error (S)	top	top (F)
t_error	t_error (S)	t_open	t_open (S)
test	test (C)	top.next	top (F)
tfind	tsearch (S)	top_panel	panel (S)
t_free	t_free (S)	top_row	menu (S)
tgetent	curses (S)	t_optmgmt	t_optmgmt (S)
tgetent	termcap (S)	total_auths	subsystems (S)

touch	touch (C)	typeahead	terminfo (S)
touchline	curses (S)	types	types (FP)
touchline	terminfo (S)	tz	tz (M)
touchwin	curses (S)	tzname	ctime (S)
touchwin	terminfo (S)	tzset	ctime (S)
_toupper	ctype (S)	u370	machid (C)
_toupper	toascii (S)	u3b	machid (C)
tparm	curses (S)	u3b15	machid (C)
tparm	terminfo (S)	u3b2	machid (C)
tplot	tplot (ADM)	u3b5	machid (C)
tput	tput (C)	uadmin	uadmin (ADM)
tputs	curses (S)	uadmin	uadmin (S)
tputs	termcap (S)	ulckpddf	getspent (S)
tputs	terminfo (S)	ulimit	ulimit (S)
tr	tr (C)	umask	umask (C)
traceoff	curses (S)	umask	umask (S)
traceoff	terminfo (S)	umnt	mnt (C)
traceon	curses (S)	umount	mount (ADM)
traceon	terminfo (S)	umount	mount (S)
track	tam (S)	umountall	mountall (ADM)
translate	translate (C)	uname	uname (C)
trchan	trchan (M)	uname	uname (S)
t_rcv	t_rcv (S)	uncompress	compress (C)
t_rcvconnect	t_rcvconnect (S)	unctrl	curses (S)
t_rcvdis	t_rcvdis (S)	unctrl	terminfo (S)
t_rcvrel	t_rcvrel (S)	undial	dial (S)
t_rcvudata	t_rcvudata (S)	undocumented	undocumented (M)
t_rcvuderr	t_rcvuderr (S)	unexecseg	execseg (S)
trig	trig (S)	unexpand	expand (C)
true	true (C)	unget	unget (CP)
tsearch	tsearch (S)	UNGETC	regexp (S)
tset	tset (C)	ungetc	ungetc (S)
t_snd	t_snd (S)	ungetch	curses (S)
t_snddis	t_snddis (S)	ungetch	terminfo (S)
t_sndrel	t_sndrel (S)	uniq	uniq (C)
t_sndudata	t_sndudata (S)	unistd	unistd (FP)
tsort	tsort (CP)	units	units (C)
t_sync	t_sync (S)	unlink	link (ADM)
tty	tty (C)	unlink	unlink (S)
tty	tty (M)	unpack	pack (C)
tty1[a-h]	serial (HW)	unpost_form	form (S)
tty2[a-h]	serial (HW)	unpost_menu	menu (S)
ttyname	ttyname (S)	unretire	unretire (ADM)
ttys	ttys (F)	UP	termcap (S)
ttyslot	ttyslot (F)	update	undocumented (M)
ttytype	ttytype (F)	update_panels	panel (S)
ttyupd	ttyupd (ADM)	uptime	uptime (C)
t_unbind	t_unbind (S)	usemouse	usemouse (C)
turnacct	acctsh (ADM)	ustat	ustat (S)
twalk	tsearch (S)	utime	utime (S)
typeahead	curses (S)	utmp	utmp (F)

utmp_getty undocumented (M)
utmpname getut (S)
uuchat dial (ADM)
uucheck uucheck (ADM)
uucico uucico (ADM)
uuclean uuclean (ADM)
uucp uucp (C)
uudecode uuencode (C)
uudemon uudemon (ADM)
uudemon.admin uudemon (ADM)
uudemon.clean uudemon (ADM)
uudemon.hour uudemon (ADM)
uudemon.poll uudemon (ADM)
uudemon.poll2 uudemon (ADM)
uuencode uuencode (C)
uugetty getty (M)
uuinstall uuinstall (ADM)
uulist uulist (ADM)
uulog uucp (C)
uuname uucp (C)
uupick uuto (C)
uusched uusched (ADM)
uustat uustat (C)
uuto uuto (C)
uutry uutry (ADM)
uux uux (C)
uuxqt uuxqt (ADM)
va_alist varargs (S)
va_arg varargs (S)
va_dcl varargs (S)
va_end varargs (S)
val val (CP)
va_list varargs (S)
values values (M)
varargs varargs (S)
va_start varargs (S)
vax machid (C)
vc vc (CP)
vectorsinuse vectorsinuse (ADM)
vedit vi (C)
vfprintf vprintf (S)
vi vi (C)
vidattr curses (S)
vidattr terminfo (S)
vidi vidi (C)
vidputs curses (S)
vidputs terminfo (S)
view vi (C)
vldldptr ldptr (S)
vmstat vmstat (C)
volcopy volcopy (ADM)

vprintf vprintf (S)
vsprintf vprintf (S)
vwprintw curses (S)
vwprintw terminfo (S)
vwscanw curses (S)
vwscanw terminfo (S)
w w (C)
waddch curses (S)
waddch terminfo (S)
waddstr curses (S)
waddstr terminfo (S)
wait wait (C)
wait wait (S)
waitpid wait (S)
waitsem waitsem (S)
wall wall (ADM)
wattroff curses (S)
wattroff terminfo (S)
wattron curses (S)
wattron terminfo (S)
wattrset curses (S)
wattrset terminfo (S)
wc wc (C)
wclear curses (S)
wclear terminfo (S)
wclrtoebot curses (S)
wclrtoebot terminfo (S)
wclrtoeol curses (S)
wclrtoeol terminfo (S)
wcmd tam (S)
wcreate tam (S)
wcstombs mblen (S)
wctomb mblen (S)
wdelch curses (S)
wdelch terminfo (S)
wdelete tam (S)
wdeleteln curses (S)
wdeleteln terminfo (S)
wechochar curses (S)
wechochar terminfo (S)
werase curses (S)
werase terminfo (S)
wexit tam (S)
wgetc tam (S)
wgetch curses (S)
wgetch terminfo (S)
wgetmouse tam (S)
wgetpos tam (S)
wgetsel tam (S)
wgetstat tam (S)
wgetstr curses (S)

wgetstr	terminfo (S)	wstandout	curses (S)
wgoto	tam (S)	wstandout	terminfo (S)
what	what (C)	wtinit	wtinit (ADM)
what	what (CP)	wtmp	utmp (F)
who	who (C)	wtmpfix	fwtmp (ADM)
whodo	whodo (C)	wuser	tam (S)
wicoff	tam (S)	x286emul	x286emul (C)
wicon	tam (S)	x286emul	x286emul (CP)
widest_auth	subsystems (S)	xargs	xargs (C)
winch	curses (S)	xbackup	xbackup (ADM)
winch	terminfo (S)	xbackup	xbackup (F)
wind	tam (S)	xdump	xbackup (ADM)
winit	tam (S)	xdumpdir	xdumpdir (ADM)
winsch	curses (S)	xinstall	xinstall (ADM)
winsch	terminfo (S)	xlist	xlist (S)
winserln	curses (S)	x.out	x.out (FP)
winserln	terminfo (S)	xrestor	xrestore (ADM)
wlabel	tam (S)	xrestore	xrestore (ADM)
wmove	curses (S)	xstr	xstr (CP)
wmove	terminfo (S)	xt	xt (HW)
wndelay	tam (S)	xtd	xtd (ADM)
wnl	tam (S)	xtil	xtil (CP)
wnoutrefresh	curses (S)	xtod	xtod (C)
wnoutrefresh	terminfo (S)	xtproto	xtproto (M)
wpostwait	tam (S)	xtract	xtract (C)
wprexec	tam (S)	xts	xts (ADM)
wprintf	tam (S)	xtt	xtt (ADM)
wprintw	curses (S)	y0	bessel (S)
wprintw	terminfo (S)	y1	bessel (S)
wprompt	tam (S)	yacc	yacc (CP)
wputc	tam (S)	yes	yes (C)
wputs	tam (S)	yn	bessel (S)
wrastop	tam (S)	zcat	compress (C)
wreadmouse	tam (S)		
wrefresh	curses (S)		
wrefresh	tam (S)		
wrefresh	terminfo (S)		
write	write (C)		
write	write (S)		
write_authorizations	subsystems (S)		
wscanw	curses (S)		
wscanw	terminfo (S)		
wselect	tam (S)		
wserver	wserver (C)		
wsetmouse	tam (S)		
wsetscrreg	curses (S)		
wsetscrreg	terminfo (S)		
wsetstat	tam (S)		
wslk	tam (S)		
wstandend	curses (S)		
wstandend	terminfo (S)		

System services (S)

(S)

Intro

introduce system services, library routines, and error numbers

Description

This manual page describes all system services and error values. In addition, this manual page provides listings of obsolete and withdrawn system services.

System services include all library routines or system calls that are available in the operating system kernel.

Standards conformance

Many of the system services listed in section (S) conform to one or more of the following standards:

Standard	Latest Revision
NIST FIPS	National Institute of Standards and Technology (NIST), Federal Information Processing Standard, 151-1
POSIX	IEEE 1003.1 and ISO/IEC 9945-1, First Edition 1990-12-07
SVID	Issue 2, Select Code 307-127
XPG3	X/Open Portability Guide, XSI System Interface and Headers, December 1988
Intel	Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2)

XPG3 withdrawn routines

The X/Open Portability Guide, Issue 3 (XPG3), has been revised so that previously supported system services have now been withdrawn. To maintain portability with future issues of these standards, use of these services is not recommended. Specifying the `cc ... -xpg3` command causes the withdrawn routines to not be included in the output object.

The following system services have been withdrawn from XPG3:

<code>acct</code>	<code>getutent</code>	<code>nl_cxtime</code>	<code>setpgrp</code>
<code>brk</code>	<code>getutid</code>	<code>nl_fprintf</code>	<code>setpwent</code>
<code>catgetmsg</code>	<code>getutline</code>	<code>nl_fscanf</code>	<code>setutent</code>
<code>ecvt</code>	<code>gsignal</code>	<code>nl_init</code>	<code>ssignal</code>
<code>edata</code>	<code>ioctl</code>	<code>nl_printf</code>	<code>stime</code>
<code>end</code>	<code>l3tol</code>	<code>nl_scanf</code>	<code>sync</code>
<code>endgrent</code>	<code>lockf</code>	<code>nl_sprintf</code>	<code>sys_errlist</code>
<code>endpwent</code>	<code>logname</code>	<code>nl_sscanf</code>	<code>sys_nerr</code>
<code>endutent</code>	<code>l3tol</code>	<code>nl_strcmp</code>	<code>ttyslot</code>
<code>etext</code>	<code>mallinfo</code>	<code>nl_strncmp</code>	<code>umount</code>
<code>fcvt</code>	<code>mallopt</code>	<code>plock</code>	<code>ustat</code>
<code>fgetgrent</code>	<code>matherr</code>	<code>profil</code>	<code>utmpname</code>
<code>fgetpwent</code>	<code>mknod</code>	<code>ptrace</code>	
<code>gcvt</code>	<code>mktemp</code>	<code>putpwent</code>	
<code>getgrent</code>	<code>monitor</code>	<code>pututline</code>	
<code>getpw</code>	<code>mount</code>	<code>sbrk</code>	
<code>getpwent</code>	<code>nl_asctime</code>	<code>setgrent</code>	

In addition, the following header files are also withdrawn:

<code>malloc.h</code>	<code>sys/acct.h</code>	<code>termio.h</code>
<code>memory.h</code>	<code>sys/dirent.h</code>	<code>ustat.h</code>
<code>mon.h</code>	<code>sys/lock.h</code>	<code>utmp.h</code>

Obsolete routines

The following routines are no longer described in section (S). In all cases, the routines were documented, but did not have equivalent source code in the Development System. The routines are:

Routine	Manual Page
agetnum	authcap(S)
ascftime	ctime(S)
cftime	ctime(S)
check_basic_data_structures	check_basic_data_structures(S)
db_lock	dblock(S)
db_unlock	dblock(S)
discr_check_file	discr(S)
ensure_db_lock	dblock(S)
ensure_discr_file	discr(S)
ensure_e_ids	discr(S)
file_lock	dblock(S)
file_unlock	dblock(S)
get_discr_file	discr(S)
getprcment	getprcment(S)
getprcmnam	getprcment(S)
putprcmnam	getprcment(S)
read_cm_fields	fields(S)
running_secure	fields(S)
setchrclass	ctype(S)
setprcment	getprcment(S)
store_cm_fields	fields(S)

Because of the removal of these routines, the `check_basic_data_structures(S)`, `dblock(S)`, `discr(S)`, and `getprcment(S)` manual pages have been entirely removed.

Error values

Most services that are part of the operating system kernel have one or more error returns. An error condition is indicated by a value returned by each function, for example, -1 for functions returning an integer. The individual manual pages specify details. An error number is also made available in the external variable `errno`. `errno` is not cleared on successful calls, so it should be tested only after an error has been indicated.

Each of the tables in this section list errors numerically. After the tables, alphabetic information is provided about each error.

The following table lists error values defined in `/usr/include/sys/errno.h` (any error number that is not listed in this table is reserved):

1 EPERM	31 EMLINK	66 EREMOTE
2 ENOENT	32 EPIPE	67 ENOLINK
3 ESRCH	33 EDOM	68 EADV
4 EINTR	34 ERANGE	69 ESRMNT
5 EIO	35 ENOMSG	70 ECOMM
6 ENXIO	36 EIDRM	71 EPROTO
7 E2BIG	37 ECHRNG	74 EMULTIHOP
8 ENOEXEC	38 EL2NSYNC	75 ELBIN
9 EBADF	39 EL3HLT	76 EDOTDOT
10 ECHILD	40 EL3RST	77 EBADMSG
11 EAGAIN	41 ELNRNG	78 ENAMETOOLONG
12 ENOMEM	42 EUNATCH	80 ENOTUNIQ
13 EACCES	43 ENOCSS	81 EBADFD
14 EFAULT	44 EL2HLT	82 EREMCHG
15 ENOTBLK	45 EDEADLK	83 ELIBACC
16 EBUSY	46 ENOLCK	84 ELIBBAD
17 EEXIST	50 EBADE	85 ELIBSCN
18 EXDEV	51 EBADR	86 ELIBMAX
19 ENODEV	52 EXFULL	87 ELIBEXEC
20 ENOTDIR	53 ENOANO	89 ENOSYS
21 EISDIR	54 EBADRQC	90 TCPERR
22 EINVAL	55 EBADSLT	135 EUCLEAN
23 ENFILE	56 EDEADLOCK	137 ENOTNAM
24 EMFILE	57 EBFONT	138 ENAVAIL
25 ENOTTY	60 ENOSTR	139 EISNAM
26 ETXTBSY	61 ENODATA	140 EREMOTEIO
27 EFBIG	62 ETIME	141 EINIT
28 ENOSPC	63 ENOSR	142 EREMDEV
29 ESPIPE	64 ENONET	145 ENOTEMPTY
30 EROFS	65 ENOPKG	150 ELOOP

ISAM errors

The following table lists ISAM error values found in the `/usr/include/isam.h` file:

100 EDUPL	108 EKEXISTS	116 EBADMEM
101 ENOTOPEN	109 EPRIMKEY	117 EBADCOLL
102 EBADARG	110 EENDFILE	118 EPKDUP
103 EBADKEY	111 ENOREC	119 EBADTX
104 ETOOMANY	112 ENOCURR	120 ETXLCK
105 EBADFILE	113 EFLOCKED	121 EJOURNAL
106 ENOTEXCL	114 EFNAME	122 ENOLOCK
107 ELOCKED	115 ENOLOK	

STREAMS TCP errors

The following table lists STREAMS TCP errors defined in */usr/include/sys/errno.h*:

63 ENOBUFS	99 EOPNOTSUPP	110 EISCONN
90 EWOULDBLOCK	100 EPFNOSUPPORT	111 ENOTCONN
91 EINPROGRESS	101 EAFNOSUPPORT	112 ESHUTDOWN
92 EALREADY	102 EADDRINUSE	113 ETOOMANYREFS
93 ENOTSOCK	103 EADDRNOTAVAIL	114 ETIMEDOUT
94 EDESTADDRREQ	104 ENETDOWN	115 ECONNREFUSED
95 EMSGSIZE	105 ENETUNREACH	116 EHOSTDOWN
96 EPROTOTYPE	106 ENETRESET	117 EHOSTUNREACH
97 EPROTONOSUPPORT	107 ECONNABORTED	118 ENOPROTOOPT
98 ESOCKTNOSUPPORT	108 ECONNRESET	

Message catalog errors

The following table lists message catalog error values that are defined in */usr/include/nls/i_errno.h*:

-1 I_ERROR	7 I_EICNV	14 I_EIBNV
1 I_EBACC	8 I_EISTI	15 I_EREAD
2 I_EBADF	9 I_EISTR	16 I_ENMSG
3 I_ENMEM	10 I_EIDOC	17 I_EBADC
4 I_EINTL	11 I_EICTM	18 I_ENOOP
5 I_EIPRP	12 I_EICPR	
6 I_EICOL	13 I_EICSC	

TLI and XTI errors

The functions in the network services library return error values to the `t_errno` variable rather than `errno`. Both TLI and XTI have error codes with the same names and functions, except XTI has additional error codes- `TNOSTRUCTYPE`, `TBADNAME`, `TBADQLEN` and `TADDRBUSY`. The TLI error codes are defined in */usr/include/sys/tiuser.h*; the XTI error codes are defined in */usr/include/xti.h*. The error code values are listed in the following table and their meanings are described in the alphabetic listing in the next section:

1 TBADADDR	9 TLOOK	17 TNOREL
2 TBADOPT	10 TBADDATA	18 TNOTSUPPORT
3 TACCES	11 TBUFOVFLW	19 TSTATECHNG
4 TBADF	12 TFLOW	20 TNOSTRUCTYPE (XTI only)
5 TNOADDR	13 TNODATA	21 TBADNAME (XTI only)
6 TOUTSTATE	14 TNODIS	22 TBADQLEN (XTI only)
7 TBADSEQ	15 TNOUDERR	23 TADDRBUSY (XTI only)
8 TSYSERR	16 TBADFLAG	

Error descriptions

The following alphabetically sorted list provides more information on each value defined in the previous error value tables:

E2BIG	Arg list too long: An argument list longer than 5,120 bytes is presented to a member of the exec family.
E2NSYNC	Level 2 not synchronized
EACCES	Permission denied: An attempt was made to access a file in a way forbidden by the protection system.
EADDRINUSE	Address already in use
EADDRNOTAVAIL	Cannot assign requested address
EADV	Advertise error: This error is RFS-specific. It occurs when users try to advertise a resource that has been advertised already, or try to stop the RFS while there are resources still advertised, or try to force unmount a resource when it is still advertised.
EAFNOSUPPORT	Address family not supported by protocol family
EAGAIN	No more processes: A fork failed because the system's process table is full or the user is not allowed to create any more processes.
EALREADY	Operation already in progress
EBADARG	Bad argument: One of the ISAM call's arguments is outside the range of allowable values for that argument. The ISAM variable <code>isstat1</code> is set to 9 and <code>isstat2</code> is set to 0.
EBADCOLL	Bad ISAM collating
EBADE	Bad exchange descriptor
EBADF	Bad file number: A file descriptor refers to no open file, a read request is made to a file that is open only for writing, or a write request is made to a file only open for reading.
EBADFD	File descriptor in bad state
EBADFILE	Bad file: The format of an ISAM file has been damaged. The ISAM variable <code>isstat1</code> is set to 9 and <code>isstat2</code> is set to 0.
EBADKEY	Bad key: At least one of the elements of an ISAM index key description is not within the range of allowable values for that element. The ISAM variable <code>isstat1</code> is set to 9 and <code>isstat2</code> is set to 0.

EBADMEM	Bad memory: The ISAM call was unable to allocate sufficient memory to continue.
EBADMSG	Not a data message: During a <code>read(S)</code> , <code>getmsg(S)</code> , or <code>ioctl(S) I_RECVFD</code> system call to a <code>STREAMS</code> device, something has come to the head of the queue that cannot be processed. That something depends on the system call: <code>read(S)</code> – control information or a passed file descriptor <code>getmsg(S)</code> – passed file descriptor <code>ioctl(S)</code> – control or data information
EBADR	Bad request descriptor
EBADRQC	Bad request code
EBADSLT	Invalid slot
EBFONT	Bad font file <code>fnt</code>
EBUSY	Device busy: An attempt was made to mount a device that was already mounted or an attempt was made to dismount a device on which there is an active file (open file, current directory, mounted-on file, active text segment). It also occurs if an attempt is made to enable accounting when it is already enabled.
ECHNRG	Channel number out of range
ECHILD	No child processes: A <code>wait</code> was executed by a process that had no existing or unwaited-for child processes.
ECOMM	Communication error on send: This error is <code>RFS</code> -specific. It occurs when trying to send messages to remote machines but no virtual circuit can be found.
ECONNABORTED	Software caused connection abort
ECONNREFUSED	Connection refused
ECONNRESET	Connection reset by peer
EDEADLK	Deadlock situation detected and avoided: A deadlock situation was detected and avoided. This error pertains to file and record locking.
EDEADLOCK	File locking deadlock
EDESTADDRREQ	Destination address required
EDOM	Math arg out of domain of func: The argument of a function in the math package is out of the domain of the function.

EDOTDOT	Undefined
EDUPL	Duplicate: The call tried to add a duplicate value to an ISAM index where duplicates are not permitted. This error results from a call to isaddindex(S) , isrewcurr(S) , isrewrite(S) , or iswrite(S) . The ISAM variable isstat1 is set to 2 and isstat2 is set to 2.
EENDFILE	End of file: The beginning or end of an ISAM file was encountered. The ISAM variable isstat1 is set to 1 and isstat2 is set to 0.
EEXIST	File exists: An existing file was mentioned in an inappropriate context, for example, link .
EFAULT	Bad address: The system encountered a hardware fault in attempting to use an argument of a system call.
EFBIG	File too large: The size of a file exceeded the maximum file size (1,082,201,088 bytes) or ULIMIT .
EFLOCKED	File locked: The ISAM file is exclusively locked by another process. The ISAM variable isstat1 is set to 9 and isstat2 is set to 0.
EFNAME	Filename too long: The specified ISAM file is too long. The ISAM variable isstat1 is set to 9 and isstat2 is set to 0.
EHOSTDOWN	Host is down
EHOSTUNREACH	No route to host
EIDRM	Identifier removed: This error is returned to a process that resumes execution due to the removal of an identifier from the file system's name space; see msgctl(S) , semctl(S) , and shmctl(S) .
EINPROGRESS	Operation now in progress
EINTR	Interrupted system call: An asynchronous signal (such as interrupt or quit) that the user has elected to catch occurred during a system call. If execution is resumed after processing the signal, it appears as if the interrupted system call returned this error condition.
EIO	I/O error: Some physical I/O error. This error may in some cases occur on a call following the one to which it actually applies.
EINIT	Reserved: This error number is reserved for future use.

EINVAL	Invalid argument: An invalid argument (for example, dismounting a non-mounted device; mentioning an undefined signal in <code>signal</code> or <code>kill</code> ; reading or writing a file for which <code>lseek</code> has generated a negative pointer) was specified. Also set by the math functions described in the (S) entries of this manual.
EISCONN	Socket is already connected
EISDIR	Is a directory: An attempt was made to write on a directory.
EISNAM	Is a name file: A name file (semaphore, shared data, and so on) was specified when not expected.
EKEYXISTS	Key exists: The ISAM call tried to add an index that already exists. The ISAM variable <code>isstat1</code> is set to 9 and <code>isstat2</code> is set to 0.
EL2HLT	Level 2 halted
EL3HLT	Level 3 halted
EL3RST	Level 3 reset
ELBIN	Undefined
ELIBACC	Cannot access a needed shared library: Trying to <code>exec(S)</code> an <i>a.out</i> that requires a shared library (to be linked in) and the shared library does not exist or the user does not have permission to use it.
ELIBBAD	Accessing a corrupted shared lib: Trying to <code>exec(S)</code> an <i>a.out</i> that requires a shared library (to be linked in), and <code>exec(S)</code> could not load the shared library. The shared library is probably corrupted.
ELIBEXEC	Cannot exec a shared library directly: Trying to exec a shared library directly. This is not allowed.
ELIBMAX	Attempting to link in more shared libraries than system limit: Trying to <code>exec(S)</code> an <i>a.out</i> that requires more shared libraries (to be linked in), than is allowed on the current configuration of the system. See the <i>System Administrator's Guide</i>
ELIBSCN	<i>.lib</i> section in <i>a.out</i> corrupted: Trying to <code>exec(S)</code> an <i>a.out</i> that requires a shared library (to be linked in) and there was erroneous data in the <i>.lib</i> section of the <i>a.out</i> . The <i>.lib</i> section tells <code>exec(S)</code> what shared libraries are needed. The <i>a.out</i> is probably corrupted.
ELNRNG	Link number out of range
ELOCKED	Locked record: The ISAM call requested a record that is locked for use by another process. The ISAM variable <code>isstat1</code> is set to 9 and <code>isstat2</code> is set to 0.

ELOOP	Directory not empty, or too many symbolic links in path
EMFILE	Too many open files: No process may have more file descriptors open at a time than the number set in the NOFILES tunable kernel parameter.
EMLINK	Too many links: An attempt was made to make more than the maximum number of links (1000) to a file.
EMSGSIZE	Message too long
EMULTIHOP	Multihop attempted: This error is RFS -specific. It occurs when users try to access remote resources that are not directly accessible.
ENAMETOOLONG	Filename too long: The filename given was longer than is allowed.
ENANO	Anode table overflow
ENAVAIL	Not available: An opensem(S) , waitsem(S) , or sigsem(S) was issued to a semaphore that has not been initialized by a call to creatsem(S) . A sigsem was issued to a semaphore out of sequence; that is, before the process has issued the corresponding waitsem to the semaphore. An nbwaitsem was issued to a semaphore guarding a resource that is currently in use by another process. The semaphore on which a process was waiting has been left in an inconsistent state when the process controlling the semaphore exits without relinquishing control properly; that is, without issuing a waitsem on the semaphore.
ENETDOWN	Network is down
ENETRESET	Network dropped connection on reset
ENETUNREACH	Network is unreachable
ENFILE	File table overflow: An attempt to open a file occurred while the system table of open files was full. While the table is full, no opens are accepted.
ENOBUFS	No buffer space available
ENOCSS	No CSI structure available
ENOCURR	No current record: The ISAM call has been requested to operate on the current record when <i>not-current-record</i> has been defined. The ISAM variable isstat1 is set to 2 and isstat2 is set to 1.
ENODATA	No data

ENODEV	No such device: An attempt was made to apply an inappropriate system call to a device; for example, reading from a write-only device.
ENOENT	No such file or directory: A specified filename or directory did not exist.
ENOEXEC	Exec format error: A request was made to execute a file, which, although it had the appropriate permissions, did not start with a valid magic number.
ENOLCK	No record locks available: An attempt was made to lock or unlock a file that would have created an additional lock table entry while the system lock table was full.
ENOLINK	The link has been severed: An attempt was made to connect to a remote machine with RFS and the link (virtual circuit) was not available.
ENOLOK	No lock: The requested ISAM lock manager cannot be established. Check the DBKEY environment variable.
ENOMEM	Not enough space: During an <code>exec</code> , or <code>sbrk</code> , a program requested more space than the system was able to supply. This is not a temporary condition; the maximum space size is a system parameter. The error may also occur if the arrangement of text, data, and stack segments requires too many segmentation registers, or if there is not enough swap space during a fork.
ENOMSG	No message of desired type: An attempt was made to receive a message of a type that does not exist on the specified message queue; see <code>msgop(S)</code> .
ENONET	Machine is not on the network: An attempt was made to advertise, unadvertise, mount, or unmount remote RFS resources when a machine has not done the proper start-up to connect to the network.
ENOTNAM	Not a name file: A <code>creatsem(S)</code> , <code>opensem(S)</code> , <code>waitsem(S)</code> , or <code>sigsem(S)</code> was issued using an invalid semaphore identifier.
ENOPKG	Package not installed: An attempt was made to call a routine in an uninstalled package.
ENOPROTOOPT	Protocol not available
ENOREC	No record: The ISAM file contains no record meeting the search conditions. The ISAM variable <code>isstat1</code> is set to 2 and <code>isstat2</code> is set to 3.

ENOSPC	No space left on device: During a write to an ordinary file, no free space was found on the device.
ENOSR	Out of streams resources: During a STREAMS open(S) , either no STREAMS queues or no STREAMS head data structures were available.
ENOSTR	Device not a stream: A putmsg(S) or getmsg(S) system call was attempted on a file descriptor that is not a STREAMS device.
ENOSYS	Function not implemented
ENOTBLK	Block device required: A nonblock file was specified where a block device was required, for example, in a call to mount(S) .
ENOTCONN	Socket is not connected
ENOTDIR	Not a directory: A directory was specified that cannot be found such as in a path prefix or as an argument to chdir(S) .
ENOTEMPTY	Directory not empty
ENOTEXCL	Not exclusive: The ISAM call tried to add or delete an index but the file was not opened in exclusive mode. The ISAM variable isstat1 is set to 9 and isstat2 is set to 0.
ENOTOPEN	Not open: The call tried to operate on an ISAM file that was not yet opened or tried to perform an operation not specified by the isopen(S) read/write mode value in effect. The ISAM variable isstat1 is set to 9 and isstat2 is set to 0.
ENOTSOCK	Socket operation on non-socket
ENOTTY	Not a character device: The device requested could not be opened for character I/O.
ENOTUNIQ	Name not unique on network: An attempt was made to access a network and duplicate network names were found. Each system on the network must have a unique name.
ENXIO	No such device or address: I/O was attempted on a non-existing subdevice, or addressing is performed outside the limits of a device. This error may occur when a tape drive is not on line or a disk pack is not loaded on a drive.
EOPNOTSUPP	Operation not supported on socket

EPERM	Not owner: Typically, this error indicates an attempt to modify a file was made in some way forbidden except to its owner or to the super user. This error value is also returned for attempts by ordinary users to do things allowed only to the super user.
EPFNOSUPPORT	Protocol family not supported
EPIPE	Broken pipe: A write on a pipe was made for which a process was not available to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.
EPKDUP	Primary ISAM key duplicate
EPRIMEKEY	Primary key: The ISAM call tried to delete the primary key index. The primary key may not be deleted with <code>isdelindex(S)</code> . The ISAM variable <code>isstat1</code> is set to 9 and <code>isstat2</code> is set to 0.
EPROTO	Protocol error: A protocol error occurred. This error is device-specific, but is generally not related to a hardware failure.
EPROTONOSUPPORT	Protocol not supported
EPROTOTYPE	Protocol wrong type for socket
ERANGE	Math result not representable: The value of a function in the math package cannot be represented within machine precision.
EREMCHG	Remote address changed
EREMDEV	Reserved: This error number is reserved for future use.
EREMOTE	The object is remote: This error is RFS-specific. It occurs when users try to advertise a resource that is not on the local machine, or try to mount/unmount a device (or pathname) that is on a remote machine.
EREMOTEIO	Remote I/O error: There was an I/O error on a remote device.
EROFS	Read-only filesystem: An attempt to modify a file or directory was made on a device mounted read-only.
ESPIPE	Illegal seek: A call to <code>lseek(S)</code> was issued to a pipe.
ESHUTDOWN	Cannot send after socket shutdown
ESOCKTNOSUPPORT	Socket type not supported

ESRCH	No such process: No process can be found corresponding to that specified by <i>pid</i> in <i>kill</i> or <i>ptrace</i> .
ESRMNT	srmount error: This error is RFS-specific. It occurs when users try to stop RFS while resources are still mounted by remote machines.
ETIME	Timer expired: The timer set for a STREAMS <i>ioctl(S)</i> call has expired. The cause of this error is device specific and indicates either a hardware or software failure, or perhaps a timeout value that is too short for the specific operation. The status of the <i>ioctl(S)</i> operation is indeterminate.
ETIMEDOUT	Connection timed out
ETOOMANY	Too many files open: The ISAM call attempted to exceed the maximum number of files that can be open at one time. The ISAM variable <i>isstat1</i> is set to 9 and <i>isstat2</i> is set to 0.
ETOOMANYREFS	Too many references: cannot splice
ETXTBSY	Text file busy: An attempt was made to execute a pure-procedure program that is currently open for writing (or reading). Also an attempt to open for writing a pure-procedure program that is being executed.
EUCLEAN	Filesystem needs cleaning: An attempt was made to mount a filesystem using <i>mount(S)</i> , but the super block is not flagged as clean.
EUNATCH	Protocol driver not attached
EWouldBLOCK	Operation would block
EXDEV	Cross-device link: A link to a file on another device was attempted.
EXFULL	Exchange table full
I_EBACC	Inaccessible database
I_EBADC	Bad call of function
I_EBADF	Corrupted database/msgcat
I_EIBNV	Bad environment
I_EICNV	Illegal conversion table
I_EICOD	Illegal code in the database
I_EICOL	Illegal collation table
I_EICPR	Illegal conversion in doprnt

I_EICSC	Illegal conversion in doscan
I_EICTM	Illegal format in i_ctime
I_EINTL	Illegal database pointer
I_EIPRP	Illegal property table
I_EISTI	Illegal string index table
I_EISTR	Illegal string name
I_ENMEM	Not enough memory
I_ENMSG	No message available
I_ENOOP	Cannot open message catalog
I_EREAD	Read error in message catalog
I_ERROR	General error return used
TACCES	Incorrect permissions
TADDRBUSY	The specified address is already in use.
TBADADDR	Incorrect address format
TBADDATA	Illegal amount of data
TBADF	Illegal transport file descriptor
TBADFLAG	Bad flags
TBADNAME	Invalid transport provider name
TBADOPT	Incorrect option format
TBADQLEN	The qlen member of the t_bind structure has a value of zero.
TBADSEQ	Bad call sequence number
TBUFOVFLW	Buffer not large enough
TNOADDR	Could not allocate address
TFLOW	Flow control
TLOOK	Event requires attention
TNODATA	No data units are available
TNODIS	No disconnect request found in the queue
TNOREL	No orderly release request found in the queue
TNOSTRUCTYPE	Unsupported structure type requested
TNOTSUPPORT	Primitive not supported
TNOUDERR	Unit data error not found
TOUTSTATE	Out of state
TSTATECHNG	State is in process of changing
TSYSERR	System error

Notes

The following tables provide more information for ISAM errors using the `isstat1` and `isstat2` variables:

<code>isstat1</code>	Description
0	Successful execution of the call
1	End of file encountered
2	Invalid key
3	System error
9	User-defined error

The `isstat2` values are not defined.

See also

Development System Encyclopedia for definitions and concepts

Routines

List of all system service (S) section routines

Description

The **Routines** manual page lists all the routines and prominent constants in section (S).

Each routine appears with its appropriate manual page, the operating system with which it works, the type of routine, and the cc(CP) library link argument.

The operating system column codes have the following meanings:

Code	Meaning
U	UNIX system only; this routine only is in a COFF library
X	XENIX only; this routine only is in a OMF library
UX	UNIX system and XENIX; this routine is in both a COFF and an OMF library

The third column also identifies the type of routine with one of the following meanings:

Code	Meaning
C	indicates a system call
F	indicates a function defined in a header file
G	indicates that it is generated by the C compiler
L	indicates a library routine
M	indicates a macro
V	indicates a variable defined in a header file

Entries with a -l library call usage are system calls. Since system calls are linked automatically, the -l option need not be specified explicitly; it is included in this table for information purposes only. System calls derived from XENIX retain the -lx as an indication of their heritage.

Routine name	Manual page	OS/Type	cc library option
a64l	a64l(S)	UX/L	-lc
abort	abort(S)	UX/L	-lc
abs	abs(S)	UX/L	-lc
acceptable_password	accept_pw(S)	U/L	-lprot
access	access(S)	UX/L	-lc
acct	acct(S)	UX/L	-lc
acos	trig(S)	UX/L	-lm
addch	curses(S)	UX/L	-lcurses
addch	tam(S)	U/L	-ltam, -lcurses
addch	terminfo(S)	UX/L	-lcurses
addkey	curses(S)	U/L	-ltcap
addstr	curses(S)	UX/L	-lcurses
addstr	tam(S)	U/L	-ltam, -lcurses
addstr	terminfo(S)	UX/L	-lcurses
adf_gttok	tam(S)	U/L	-ltam, -lcurses
adf_gtwrd	tam(S)	U/L	-ltam, -lcurses
adf_gtxcd	tam(S)	U/L	-ltam, -lcurses
advance	regex(S)	U/F	function - regex.h
agetdefault	authcap(S)	U/L	-lprot, -lx
agetfile	authcap(S)	U/L	-lprot, -lx
agetflag	authcap(S)	U/L	-lprot, -lx
agetstr	authcap(S)	U/L	-lprot, -lx
agettty	authcap(S)	U/L	-lprot, -lx
agetuser	authcap(S)	U/L	-lprot, -lx
alarm	alarm(S)	UX/C	
alloclpdr	alloclpdr(S)	U/L	-lld
altzone	ctime(S)	UX/V	variable - time.h
arc	plot(S)	U/L	-lplot
asctime	ctime(S)	UX/L	-lc
asetdefaults	authcap(S)	U/L	-lprot, -lx
asin	trig(S)	UX/L	-lm
assert	assert(S)	UX/M	macro - assert.h
atan	trig(S)	UX/L	-lm
atan2	trig(S)	UX/L	-lm
atexit	atexit(S)	U/L	-lc
atof	atof(S)	UX/L	-lc
atoi	atof(S)	UX/L	-lc
atol	atof(S)	UX/L	-lc
attroff	tam(S)	U/L	-ltam, -lcurses
attroff	terminfo(S)	UX/L	-lcurses
attron	curses(S)	U/L	-lcurses
attron	tam(S)	U/L	-ltam, -lcurses
attron	terminfo(S)	UX/L	-lcurses
attrset	curses(S)	U/L	-lcurses
attrset	terminfo(S)	UX/L	-lcurses

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
audit_adjust_mask	authaudit(S)	U/L	-lprot
audit_auth_entry	authaudit(S)	U/L	-lprot
audit_close	audit(S)	U/L	-laudit
audit_lax_file	authaudit(S)	U/L	-lprot
audit_lock	authaudit(S)	U/L	-lprot
audit_login	authaudit(S)	U/L	-lprot
audit_no_resource	authaudit(S)	U/L	-lprot
audit_open	audit(S)	U/L	-laudit
audit_passwd	authaudit(S)	U/L	-lprot
audit_read	audit(S)	U/L	-laudit
audit_security_failure	authaudit(S)	U/L	-lprot
audit_subsystem	authaudit(S)	U/L	-lprot
authorized_user	subsystems(S)	U/L	-lprot, -lx
baudrate	curses(S)	U/L	-lcurses
baudrate	tam(S)	U/L	-ltam, -lcurses
baudrate	terminfo(S)	UX/L	-lcurses
beep	curses(S)	U/L	-lcurses
beep	tam(S)	U/L	-ltam, -lcurses
beep	terminfo(S)	UX/L	-lcurses
bigcrypt	getpasswd(S)	U/L	-lprot, -lcrypt_i, -lx
bigcryptmax	getpasswd(S)	U/L	-lprot, -lcrypt_i, -lx
Bottom	libwindows(S)	U/L	-lwindows
bottom_panel	panel(S)	U/L	-lpanel, -lcurses
box	curses(S)	UX/L	-lcurses
box	plot(S)	U/L	-lplot
box	terminfo(S)	UX/L	-lcurses
brk	brk(S)	UX/L	-lc
brkctl	brkctl(S)	X/L	-lbrkctl
bsearch	bsearch(S)	UX/L	-lc
calloc	malloc(S)	UX/L	-lc
calloc	malloc(S)	UX/L	-lmalloc
can_change_color	curses(S)	U/L	-lcurses
catclose	catopen(S)	U/L	-lintl
catgets	catgets(S)	U/L	-lintl
catopen	catopen(S)	U/L	-lintl
cbreak	curses(S)	U/L	-lcurses
cbreak	tam(S)	U/L	-ltam, -lcurses
cbreak	terminfo(S)	UX/L	-lcurses
ceil	floor(S)	UX/L	-lm
cfgetispeed	cfspeed(S)	U/L	-lc
cfgetospeed	cfspeed(S)	U/L	-lc
cfree	malloc(S)	U/L	-lc
cfsetispeed	cfspeed(S)	U/L	-lc
cfsetospeed	cfspeed(S)	U/L	-lc

(Continued on next page)

(S)

(Continued)

Routine name	Manual page	OS/Type	cc library option
chdir	chdir(S)	UX/C	
check_auth_parameters	identity(S)	U/L	-lprot
chmod	chmod(S)	UX/C	
chown	chown(S)	UX/C	
chroot	chroot(S)	UX/C	
chsize	chsize(S)	UX/C	-lx
cirf	regexp(S)	U/V	variable - regexp.h
circle	plot(S)	U/L	-lplot
clear	curses(S)	UX/L	-lcurses
clear	tam(S)	U/L	-ltam, -lcurses
clear	terminfo(S)	UX/L	-lcurses
clearerr	ferror(S)	UX/L	-lc
clearok	curses(S)	UX/L	-lcurses
clearok	tam(S)	U/L	-ltam, -lcurses
clearok	terminfo(S)	UX/L	-lcurses
clock	clock(S)	UX/L	-lc
close	close(S)	UX/C	
closedir	directory(S)	UX/L	-lc
closepl	plot(S)	U/L	-lplot
clrtobot	curses(S)	UX/L	-lcurses
clrtobot	tam(S)	U/L	-ltam, -lcurses
clrtobot	terminfo(S)	UX/L	-lcurses
clrtoeol	curses(S)	UX/L	-lcurses
clrtoeol	tam(S)	U/L	-ltam, -lcurses
clrtoeol	terminfo(S)	UX/L	-lcurses
color_content	curses(S)	U/L	-lcurses
compile	regexp(S)	UX/F	function - regexp.h
cont	plot(S)	U/L	-lplot
copydvagent	getdvagent(S)	U/L	-lprot, -lx
copywin	curses(S)	U/L	-lcurses
cos	trig(S)	UX/L	-lm
cosh	sinh(S)	UX/L	-lm
creat	creat(S)	UX/C	
create_file_securely	create_file_securely(S)	U/L	-lprot, -lx
creatsem	creatsem(S)	UX/C	-lx
crmode	curses(S)	UX/L	-lcurses
crypt	crypt(S)	U/L	-lcrypt
crypt_close	crypt(S)	U/L	-lcrypt
ctermid	ctermid(S)	UX/L	-lc
ctime	ctime(S)	UX/L	-lc
curoff	curses(S)	U/L	-ltcap
curon	curses(S)	U/L	-ltcap
Current	libwindows(S)	U/L	-lwindows
current_field	form(S)	U/L	-lform, -lcurses

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
current_item	menu(S)	U/L	-lmenu, -lcurses
curs_set	curses(S)	U/L	-lcurses
cuserid	cuserid(S)	UX/L	-lc
daylight	ctime(S)	UX/V	variable - time.h
dbmopen	dbm(S)	UX/L	-ldb
defopen	defopen(S)	UX/L	-lc
def_prog_mode	curses(S)	U/L	-lcurses
defread	defopen(S)	UX/L	-lc
delay_output	curses(S)	U/L	-lcurses
delay_output	terminfo(S)	UX/L	-lcurses
delch	curses(S)	UX/L	-lcurses
delch	tam(S)	U/L	-ltam, -lcurses
delch	terminfo(S)	UX/L	-lcurses
del_curterm	curses(S)	U/L	-lcurses
delete	dbm(S)	UX/L	-ldb
Delete	libwindows(S)	U/L	-lwindows
deleteln	tam(S)	U/L	-ltam, -lcurses
deleteln	terminfo(S)	UX/L	-lcurses
del_panel	panel(S)	U/L	-lpanel, -lcurses
delwin	curses(S)	UX/L	-lcurses
delwin	terminfo(S)	UX/L	-lcurses
des_crypt	crypt(S)	U/L	-lcrypt
des_encrypt	crypt(S)	U/L	-lcrypt
des_setkey	crypt(S)	U/L	-lcrypt
dial	dial(S)	UX/L	-lc
difftime	difftime(S)	U/L	-lc
div	div(S)	U/L	-lc
doupdate	curses(S)	U/L	-lcurses
doupdate	terminfo(S)	UX/L	-lcurses
draino	curses(S)	U/L	-lcurses
drand48	drand48(S)	UX/L	-lc
dup	dup(S)	UX/C	
dup2	dup2(S)	UX/L	-lc
dup_field	field(S)	U/L	-lform, -lcurses
eaccess	access(S)	U/C	
echo	curses(S)	UX/L	-lcurses
echo	tam(S)	U/L	-ltam, -lcurses
echo	terminfo(S)	UX/L	-lcurses
echochar	curses(S)	U/L	-lcurses
ecvt	ecvt(S)	UX/L	-lc
edata	end(S)	UX/G	C compiler generated
encrypt	crypt(S)	U/L	-lcrypt
end	end(S)	UX/G	C compiler generated
enddvagent	getdvagent(S)	U/L	-lprot, -lx

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
endgrent	getgrent(S)	UX/L	-lc
endprdfent	getprdfent(S)	U/L	-lprot, -lx
endprfient	getprfient(S)	U/L	-lprot, -lx
endprpwent	getprpwent(S)	U/L	-lprot, -lx
endprtcent	getprtcent(S)	U/L	-lprot, -lx
endpwent	getpwent(S)	UX/L	-lc
endutent	getut(S)	UX/L	-lc
endwin	curses(S)	UX/L	-lcurses
endwin	tam(S)	U/L	-ltam, -lcurses
endwin	terminfo(S)	UX/L	-lcurses
enter_quiet_zone	dblock(S)	U/L	-lprot, -lx
erand48	drand48(S)	UX/L	-lc
erase	curses(S)	UX/L	-lcurses
erase	plot(S)	U/L	-lplot
erase	tam(S)	U/L	-lplot
erase	terminfo(S)	UX/L	-lcurses
erasechar	terminfo(S)	UX/L	-lcurses
erf	erf(S)	UX/L	-lm
erfc	erf(S)	UX/L	-lm
errno	perror(S)	UX/V	variable - errno.h
ERROR	regexp(S)	UX/M	macro - regexp.h
etext	end(S)	UX/G	C compiler generated
ev_block	ev_block(S)	UX/L	-levent
ev_close	ev_close(S)	UX/L	-levent
ev_count	ev_count(S)	UX/L	-levent
ev_flush	ev_flush(S)	UX/L	-levent
ev_getdev	ev_getdev(S)	UX/L	-levent
ev_getemask	ev_getemask(S)	UX/L	-levent
ev_gindev	ev_gindev(S)	UX/L	-levent
ev_init	ev_init(S)	UX/L	-levent
ev_open	ev_open(S)	UX/L	-levent
ev_pop	ev_pop(S)	UX/L	-levent
ev_read	ev_read(S)	UX/L	-levent
ev_resume	ev_resume(S)	UX/L	-levent
ev_setemask	ev_setemask(S)	UX/L	-levent
ev_suspend	ev_suspend(S)	UX/L	-levent
execl	exec(S)	UX/L	-lc
execle	exec(S)	UX/L	-lc
execvp	exec(S)	UX/L	-lc
execseg	execseg(S)	UX/C	-lx
execv	exec(S)	UX/L	-lc
execve	exec(S)	UX/L	-lc
execvp	exec(S)	UX/L	-lc
exhelp	tam(S)	U/L	-ltam, -lcurses

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
_exit	exit(S)	UX/L	-lc
exit	exit(S)	UX/L	-lc
exit_quiet_zone	dblock(S)	U/L	-lprot, -lx
Exit	libwindows(S)	U/L	-lwindows
exp	exp(S)	UX/L	-lm
fabs	floor(S)	UX/L	-lm
fclose	fclose(S)	UX/L	-lc
fcntl	fcntl(S)	UX/C	
fcvt	ecvt(S)	UX/L	-lc
FD_CLR	select(S)	UX/M	macro - sys/types.h
FD_ISSET	select(S)	UX/M	macro - sys/types.h
fdopen	fopen(S)	UX/L	-lc
FD_SET	select(S)	UX/M	macro - sys/types.h
FD_ZERO	select(S)	UX/M	macro - sys/types.h
feof	ferror(S)	UX/M	macro - stdio.h
ferror	ferror(S)	UX/M	macro - stdio.h
fetch	dbm(S)	UX/L	-ldb
fflush	fclose(S)	UX/L	-lc
fgetc	getc(S)	UX/L	-lc
fgetgrent	getgrent(S)	U/L	-lc
fgetpasswd	getpasswd(S)	U/L	-lprot, -lcrypt_i, -lx
fgetpos	fgetpos(S)	U/L	-lc
fgetpwent	getpwent(S)	U/L	-lc
fgets	gets(S)	UX/L	-lc
field_arg	field(S)	U/L	-lform, -lcurses
field_back	field(S)	U/L	-lform, -lcurses
field_buffer	field(S)	U/L	-lform, -lcurses
field_count	form(S)	U/L	-lform, -lcurses
field_fore	field(S)	U/L	-lform, -lcurses
field_index	form(S)	U/L	-lform, -lcurses
field_info	field(S)	U/L	-lform, -lcurses
field_init	form(S)	U/L	-lform, -lcurses
field_just	field(S)	U/L	-lform, -lcurses
field_opts	field(S)	U/L	-lform, -lcurses
field_opts_off	field(S)	U/L	-lform, -lcurses
field_opts_on	field(S)	U/L	-lform, -lcurses
field_pad	field(S)	U/L	-lform, -lcurses
field_status	field(S)	U/L	-lform, -lcurses
field_term	form(S)	U/L	-lform, -lcurses
field_type	field(S)	U/L	-lform, -lcurses
field_userptr	field(S)	U/L	-lform, -lcurses
fileno	ferror(S)	UX/M	macro - stdio.h
filter	curses(S)	U/L	-lcurses
firstkey	dbm(S)	UX/L	-ldb

(Continued on next page)

(S)

(Continued)

Routine name	Manual page	OS/Type	cc library option
fixterm	tam(S)	U/L	-ltam, -lcurses
fixterm	terminfo(S)	UX/L	-lcurses
flash	curses(S)	U/L	-lcurses
flash	tam(S)	U/L	-ltam, -lcurses
flash	terminfo(S)	UX/L	-lcurses
floor	floor(S)	UX/L	-lm
flushinp	curses(S)	U/L	-lcurses
flushinp	tam(S)	U/L	-ltam, -lcurses
flushinp	terminfo(S)	UX/L	-lcurses
fmod	floor(S)	UX/L	-lm
fopen	fopen(S)	UX/L	-lc
fork	fork(S)	UX/C	
form	tam(S)	U/L	-ltam, -lcurses
form_driver	form(S)	U/L	-lform, -lcurses
form_fields	form(S)	U/L	-lform, -lcurses
form_init	form(S)	U/L	-lform, -lcurses
form_opts	form(S)	U/L	-lform, -lcurses
form_opts_off	form(S)	U/L	-lform, -lcurses
form_opts_on	form(S)	U/L	-lform, -lcurses
form_page	form(S)	U/L	-lform, -lcurses
form_sub	form(S)	U/L	-lform, -lcurses
form_term	form(S)	U/L	-lform, -lcurses
form_userptr	form(S)	U/L	-lform, -lcurses
form_win	form(S)	U/L	-lform, -lcurses
fpathconf	pathconf(S)	U/C	
fpgetmask	fpgetround(S)	U/L	-lc
fpgetround	fpgetround(S)	U/L	-lc
fpgetsticky	fpgetround(S)	U/L	-lc
fprintf	printf(S)	UX/L	-lc
fpsetmask	fpgetround(S)	U/L	-lc
fpsetround	fpgetround(S)	U/L	-lc
fpsetsticky	fpgetround(S)	U/L	-lc
fputc	putc(S)	UX/L	-lc
fputs	puts(S)	UX/L	-lc
fread	fread(S)	UX/L	-lc
free	malloc(S)	UX/L	-lc
free	malloc(S)	UX/L	-lmalloc
free_field	field(S)	U/L	-lform, -lcurses
free_fieldtype	fieldtype(S)	U/L	-lform, -lcurses
free_form	form(S)	U/L	-lform, -lcurses
freeldptr	alloclp(S)	U/L	-lld
free_menu	menu(S)	U/L	-lmenu, -lcurses
freopen	fopen(S)	UX/L	-lc
frexp	frexp(S)	UX/L	-lm

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
fscanf	scanf(S)	UX/L	-lc
fseek	fseek(S)	UX/L	-lc
fsetpos	fsetpos(S)	U/L	-lc
fstat	stat(S)	UX/C	
fstatfs	statfs(S)	UX/C	
fsync	fsync(S)	U/L	-lc
ftell	fseek(S)	UX/L	-lc
ftime	time(S)	UX/C	
ftok	ftok(S)	UX/L	-lc
ftw	ftw(S)	UX/L	-lc
fwrite	fread(S)	UX/L	-lc
fxlist	xlist(S)	UX/M	macro - x.out.h
gamma	gamma(S)	UX/L	-lm
garbagedlines	curses(S)	U/L	-lcurses
gcvt	ecvt(S)	UX/L	-lc
getbegyx	curses(S)	U/M	macro - tinfo.h
getc	getc(S)	UX/M	macro - stdio.h
GETC	regexp(S)	UX/M	macro - regexp.h
getch	tam(S)	U/L	-ltam, -lcurses
getch	terminfo(S)	UX/L	-lcurses
getchar	getc(S)	UX/L	-lc
getcwd	getcwd(S)	UX/L	-lc
getdents	getdents(S)	UX/C	
getdim	curses(S)	U/M	macro - tcap.h
getdvagent	getdvagent(S)	U/L	-lprot, -lx
getdvagname	getdvagent(S)	U/L	-lprot, -lx
getegid	getuid(S)	UX/L	-lc
getenv	getenv(S)	UX/L	-lc
geteuid	getuid(S)	UX/L	-lc
getgid	getuid(S)	UX/C	
getgrent	getgrent(S)	UX/L	-lc
getgrgid	getgrent(S)	UX/L	-lc
getgrnam	getgrent(S)	UX/L	-lc
getgroups	getgroups(S)	U/C	
gethz	gethz(S)	U/L	-lc
getlogin	getlogin(S)	UX/L	-lc
getluid	getluid(S)	U/L	-lprot
getmaxyx	curses(S)	U/M	macro - tinfo.h
getmsg	getmsg(S)	U/C	
getopt	getopt(S)	UX/L	-lc
getorg	curses(S)	U/M	macro - tcap.h
getpass	getpass(S)	UX/L	-lc
getpasswd	getpasswd(S)	U/L	-lprot, -lcrypt_i, -lx
getpgrp	getpid(S)	UX/L	-lc

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
getpid	getpid(S)	UX/C	
getppid	getpid(S)	UX/L	-lc
getprdfent	getprdfent(S)	U/L	-lprot, -lx
getprdfnam	getprdfent(S)	U/L	-lprot, -lx
getprfient	getprfient(S)	U/L	-lprot, -lx
getprfinam	getprfient(S)	U/L	-lprot, -lx
getpriv	getpriv(S)	U/L	-lprot
getprpwent	getprpwent(S)	U/L	-lprot, -lx
getprpwnam	getprpwent(S)	U/L	-lprot, -lx
getprpwuid	getprpwent(S)	U/L	-lprot, -lx
getprtcent	getprtcent(S)	U/L	-lprot, -lx
getprtcnam	getprtcent(S)	U/L	-lprot, -lx
getpw	getpw(S)	UX/L	-lc
getpwent	getpwent(S)	UX/L	-lc
getpwnam	getpwent(S)	UX/L	-lc
getpwuid	getpwent(S)	UX/L	-lc
gets	gets(S)	UX/L	-lc
get_seed	seed(S)	U/L	-lprot
getstr	curses(S)	UX/L	-lcurses
getsyx	curses(S)	U/M	macro - tinfo.h
getstr	terminfo(S)	UX/L	-lcurses
gettmode	terminfo(S)	UX/L	-lcurses
getuid	getuid(S)	UX/C	
getutent	getut(S)	UX/L	-lc
getutid	getut(S)	UX/L	-lc
getutline	getut(S)	UX/L	-lc
getw	getc(S)	UX/L	-lc
getyx	curses(S)	UX/M	macro - tcap.h
getyx	tam(S)	U/M	macro - tcap.h
getyx	terminfo(S)	UX/L	-lcurses
gmtime	ctime(S)	UX/L	-lc
gr_idtoname	pw_nametoid(S)	U/L	-lprot
gr_nametoid	pw_nametoid(S)	U/L	-lprot
gsignal	ssignal(S)	UX/L	-lc
halfdelay	curses(S)	U/L	-lcurses
has_colors	curses(S)	U/L	-lcurses
has_ic	curses(S)	U/L	-lcurses
has_ic	terminfo(S)	UX/L	-lcurses
has_il	curses(S)	U/L	-lcurses
has_il	terminfo(S)	UX/L	-lcurses
hcreate	hsearch(S)	UX/L	-lc
hdestroy	hsearch(S)	UX/L	-lc
hide_panel	panel(S)	U/L	-lpanel, -lcurses
hsearch	hsearch(S)	UX/L	-lc

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
hypot	hypot(S)	UX/L	-lm
idlok	curses(S)	U/L	-lcurses
idlok	terminfo(S)	UX/L	-lcurses
inch	curses(S)	UX/L	-lcurses
inch	terminfo(S)	UX/L	-lcurses
INIT	regexp(S)	UX/V	variable - regexp.h
init_color	curses(S)	U/L	-lcurses
init_pair	curses(S)	U/L	-lcurses
initscr	curses(S)	UX/L	-lcurses
initscr	tam(S)	U/L	-ltam, -lcurses
initscr	terminfo(S)	UX/L	-lcurses
insch	curses(S)	UX/L	-lcurses
insch	tam(S)	U/L	-ltam, -lcurses
insch	terminfo(S)	UX/L	-lcurses
insertln	curses(S)	UX/L	-lcurses
insertln	tam(S)	U/L	-ltam, -lcurses
insertln	terminfo(S)	UX/L	-lcurses
intrflush	curses(S)	U/L	-lcurses
intrflush	terminfo(S)	UX/L	-lcurses
ioctl	ioctl(S)	UX/C	
isaddindex	isaddindex(S)	U/L	-lisam
isalnum	ctype(S)	UX/M	macro - ctype.h
isalpha	ctype(S)	UX/M	macro - ctype.h
isascii	ctype(S)	UX/M	macro - ctype.h
isatty	ttyname(S)	UX/L	-lc
isbuild	isbuild(S)	U/L	-lisam
isclose	isclose(S)	U/L	-lisam
iscntrl	ctype(S)	UX/M	macro - ctype.h
isdelcurr	isdelcurr(S)	U/L	-lisam
isdelete	isdelete(S)	U/L	-lisam
isdelindex	isdelindex(S)	U/L	-lisam
isdelrc	isdelrc(S)	U/L	-lisam
isdigit	ctype(S)	UX/M	macro - ctype.h
isendwin	curses(S)	U/L	-lcurses
iserase	iserase(S)	U/L	-lisam
isgraph	ctype(S)	UX/L	-lc
isindexinfo	isindexinfo(S)	U/L	-lisam
islock	islock(S)	U/L	-lisam
islower	ctype(S)	UX/M	macro - ctype.h
isnan	isnan(S)	U/L	-lm
isnand	isnan(S)	U/L	-lc
isnanf	isnan(S)	U/M	macro - ieeefp.h
isopen	isopen(S)	U/L	-lisam
isprint	ctype(S)	UX/M	macro - ctype.h

(Continued on next page)

(S)

(Continued)

Routine name	Manual page	OS/Type	cc library option
ispunct	ctype(S)	UX/M	macro - ctype.h
isread	isread(S)	U/L	-lisam
isrelease	isrelease(S)	U/L	-lisam
isrename	isrename(S)	U/L	-lisam
isrewcurr	isrewcurr(S)	U/L	-lisam
isrewrec	isrewrec(S)	U/L	-lisam
isrewrite	isrewrite(S)	U/L	-lisam
issetunique	issetunique(S)	U/L	-lisam
isspace	ctype(S)	UX/M	macro - ctype.h
isstart	isstart(S)	U/L	-lisam
is_starting_egid	identity(S)	U/L	-lprot
is_starting_euid	identity(S)	U/L	-lprot
is_starting_luid	identity(S)	U/L	-lprot
is_starting_rgid	identity(S)	U/L	-lprot
is_starting_ruid	identity(S)	U/L	-lprot
isuniqueid	isuniqueid(S)	U/L	-lisam
isunlock	isunlock(S)	U/L	-lisam
isupper	ctype(S)	UX/M	macro - ctype.h
iswind	tam(S)	U/L	-ltam
iswrcurr	iswrcurr(S)	U/L	-lisam
isxdigit	ctype(S)	UX/M	macro - ctype.h
iswrite	iswrite(S)	U/L	-lisam
item_count	item(S)	U/L	-lmenu, -lcurses
item_description	item(S)	U/L	-lmenu, -lcurses
item_index	menu(S)	U/L	-lmenu, -lcurses
item_init	menu(S)	U/L	-lmenu, -lcurses
item_name	item(S)	U/L	-lmenu, -lcurses
item_opts	item(S)	U/L	-lmenu, -lcurses
item_opts_on	item(S)	U/L	-lmenu, -lcurses
item_term	menu(S)	U/L	-lmenu, -lcurses
item_userptr	item(S)	U/L	-lmenu, -lcurses
item_value	item(S)	U/L	-lmenu, -lcurses
item_visible	item(S)	U/L	-lmenu, -lcurses
j0	bessel(S)	UX/L	-lm
j1	bessel(S)	UX/L	-lm
jn	bessel(S)	UX/L	-lm
jrand48	drand48(S)	UX/L	-lc
kcodemap	tam(S)	U/M	macro - tam.h
keyname	curses(S)	U/L	-lcurses
keypad	curses(S)	U/L	-lcurses
keypad	tam(S)	U/L	-ltam, -lcurses
keypad	terminfo(S)	UX/L	-lcurses
kill	kill(S)	UX/C	
killchar	curses(S)	U/L	-lcurses

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
killchar	terminfo(S)	UX/L	-lcurses
l3tol	l3tol(S)	UX/L	-lc
l64a	a64l(S)	UX/L	-lc
label	plot(S)	U/L	-lplot
labs	labs(S)	U/L	-lc
lcong48	drand48(S)	UX/L	-lc
ldaclose	ldclose(S)	U/L	-lld
ldahread	ldahread(S)	U/L	-lld
ldaopen	ldopen(S)	U/L	-lld
ldclose	ldclose(S)	U/L	-lld
lddbl	isconv(S)	U/L	-lisam
ldexp	frexp(S)	UX/L	-lm
ldfhread	ldfhread(S)	U/L	-lld
ldfloat	isconv(S)	U/L	-lisam
ldgetname	ldgetname(S)	U/L	-lld
ldint	isconv(S)	U/M	macro - isam.h
ldiv	ldiv(S)	U/L	-lc
ldlinit	ldlread(S)	U/L	-lld
ldlitem	ldlread(S)	U/L	-lld
ldlong	isconv(S)	U/L	-lisam
ldlread	ldlread(S)	U/L	-lld
ldlseek	ldlseek(S)	U/L	-lld
ldnlseek	ldlseek(S)	U/L	-lld
ldnrseek	ldrseek(S)	U/L	-lld
ldnshread	ldshread(S)	U/L	-lld
ldnsseek	ldsseek(S)	U/L	-lld
ldohseek	ldohseek(S)	U/L	-lld
ldopen	ldopen(S)	U/L	-lld
ldrseek	ldrseek(S)	U/L	-lld
ldshread	ldshread(S)	U/L	-lld
ldsseek	ldsseek(S)	U/L	-lld
ldtbindex	ldtbindex(S)	U/L	-lld
ldtbread	ldtbread(S)	U/L	-lld
ldtbseek	ldtbseek(S)	U/L	-lld
leaveok	curses(S)	UX/L	-lcurses
leaveok	tam(S)	U/L	-ltam, -lcurses
leaveok	terminfo(S)	UX/L	-lcurses
lfind	lsearch(S)	UX/L	-lc
lgamma	gamma(S)	U/L	-lm
line	plot(S)	U/L	-lplot
linemod	plot(S)	U/L	-lplot
link	link(S)	UX/C	
link_field	field(S)	U/L	-lform, -lcurses
link_fieldtype	fieldtype(S)	U/L	-lform, -lcurses

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
llog_close	llog(S)	U/L	-lmmdf
llog_err	llog(S)	U/L	-lmmdf
llog_hdinit	llog(S)	U/L	-lmmdf
llog_init	llog(S)	U/L	-lmmdf
llog_log	llog(S)	U/L	-lmmdf
llog_open	llog(S)	U/L	-lmmdf
loc1	regex(S)	U/V	variable - regex.h
__loc1	regcmp(S)	UX/L	-lintl
loc2	regex(S)	U/V	variable - regex.h
localeconv	localeconv(S)	U/L	-lc
localtime	ctime(S)	UX/L	-lc
lock	lock(S)	UX/C	-lx
locked_out	fields(S)	U/L	-lprot
lockf	lockf(S)	UX/L	-lc
locking	locking(S)	UX/C	-lx
locs	regex(S)	U/V	variable - regex.h
log	exp(S)	UX/L	-lm
log10	exp(S)	UX/L	-lm
logname	logname(S)	UW/L	-lPW
longjmp	setjmp(S)	UX/L	-lc
longname	curses(S)	UX/L	-lcurses
longname	terminfo(S)	UX/L	-lcurses
lrnd48	drand48(S)	UX/L	-lc
lsearch	lsearch(S)	UX/L	-lc
lseek	lseek(S)	UX/L	-lc
lstat	stat(S)	U/C	
ltol3	l3tol(S)	UX/L	-lc
make_transition_files	dblock(S)	U/L	-lprot, -lx
mallinfo	malloc(S)	UX/L	-lmalloc
malloc	malloc(S)	UX/L	-lc
malloc	malloc(S)	UX/L	-lmalloc
mallopt	malloc(S)	UX/L	-lmalloc
matherr	matherr(S)	UX/L	-lm
mblen	mblen(S)	U/L	-lc
mbstowcs	mblen(S)	U/L	-lc
mbtowc	mblen(S)	U/L	-lc
memccpy	memory(S)	UX/L	-lc
memchr	memory(S)	UX/L	-lc
memcmp	memory(S)	UX/L	-lc
memcpy	memory(S)	UX/L	-lc
memmove	memmove(S)	U/L	-lc
memset	memory(S)	UX/L	-lc
menu	tam(S)	U/L	-ltam, -lcurses
menu_back	menu(S)	U/L	-lmenu, -lcurses

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
menu_driver	menu(S)	U/L	-lmenu, -lcurses
menu_fore	menu(S)	U/L	-lmenu, -lcurses
menu_grey	menu(S)	U/L	-lmenu, -lcurses
menu_init	menu(S)	U/L	-lmenu, -lcurses
menu_items	menu(S)	U/L	-lmenu, -lcurses
menu_mark	menu(S)	U/L	-lmenu, -lcurses
menu_opts	menu(S)	U/L	-lmenu, -lcurses
menu_opts_off	menu(S)	U/L	-lmenu, -lcurses
menu_opts_on	menu(S)	U/L	-lmenu, -lcurses
menu_pad	menu(S)	U/L	-lmenu, -lcurses
menu_pattern	menu(S)	U/L	-lmenu, -lcurses
menu_sub	menu(S)	U/L	-lmenu, -lcurses
menu_term	menu(S)	U/L	-lmenu, -lcurses
menu_userptr	menu(S)	U/L	-lmenu, -lcurses
menu_win	menu(S)	U/L	-lmenu, -lcurses
message	tam(S)	U/L	-ltam, -lcurses
meta	curses(S)	U/L	-lcurses
meta	terminfo(S)	UX/L	-lcurses
mkdir	mkdir(S)	U/C	
mkfifo	mkfifo(S)	U/L	-lc
mknod	mknod(S)	UX/C	
mktemp	mktemp(S)	UX/L	-lc
mktime	mktime(S)	U/L	-lc
ml_1adr	ml_send(S)	U/L	-lmmdf
ml_adr	ml_send(S)	U/L	-lmmdf
ml_aend	ml_send(S)	U/L	-lmmdf
ml_cc	ml_send(S)	U/L	-lmmdf
ml_end	ml_send(S)	U/L	-lmmdf
ml_file	ml_send(S)	U/L	-lmmdf
ml_init	ml_send(S)	U/L	-lmmdf
ml_tinit	ml_send(S)	U/L	-lmmdf
ml_to	ml_send(S)	U/L	-lmmdf
ml_txt	ml_send(S)	U/L	-lmmdf
mm_end	mmdf(S)	U/L	-lmmdf
mm_init	mmdf(S)	U/L	-lmmdf
mm_pkinit	mmdf(S)	U/L	-lmmdf
mm_radr	mmdf(S)	U/L	-lmmdf
mm_rinit	mmdf(S)	U/L	-lmmdf
mm_rrec	mmdf(S)	U/L	-lmmdf
mm_rply	mmdf(S)	U/L	-lmmdf
mm_rstm	mmdf(S)	U/L	-lmmdf
mm_rtxt	mmdf(S)	U/L	-lmmdf
mm_sbend	mmdf(S)	U/L	-lmmdf
mm_sbin	mmdf(S)	U/L	-lmmdf

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
mm_wadr	mmddf(S)	U/L	-lmmddf
mm_wend	mmddf(S)	U/L	-lmmddf
mm_winit	mmddf(S)	U/L	-lmmddf
mm_wrec	mmddf(S)	U/L	-lmmddf
mm_wrply	mmddf(S)	U/L	-lmmddf
mm_wstm	mmddf(S)	U/L	-lmmddf
mm_wtend	mmddf(S)	U/L	-lmmddf
mm_wtxt	mmddf(S)	U/L	-lmmddf
modf	frexp(S)	UX/L	-lm
monitor	monitor(S)	UX/L	-lc
mount	mount(S)	UX/L	-lc
Move	libwindows(S)	U/L	-lwindows
move	curses(S)	UX/L	-lcurses
move	plot(S)	U/L	-lplot
move	tam(S)	U/L	-ltam, -lplot
move	terminfo(S)	UX/L	-lcurses
move_field	field(S)	U/L	-lform, -lcurses
move_panel	panel(S)	U/L	-lpanel, -lcurses
mrnd48	drand48(S)	UX/L	-lc
msgctl	msgctl(S)	UX/L	-lc
msgget	msgget(S)	UX/L	-lc
msgrcv	msgop(S)	UX/L	-lc
msgsnd	msgop(S)	UX/L	-lc
mvaddch	curses(S)	UX/L	-lcurses
mvaddch	tam(S)	U/L	-ltam, -lcurses
mvaddch	terminfo(S)	UX/L	-lcurses
mvaddstr	curses(S)	UX/L	-lcurses
mvaddstr	tam(S)	U/L	-ltam, -lcurses
mvaddstr	terminfo(S)	UX/L	-lcurses
mvcur	curses(S)	UX/L	-lcurses
mvcur	terminfo(S)	UX/L	-lcurses
mvdelch	curses(S)	UX/L	-lcurses
mvdelch	terminfo(S)	UX/L	-lcurses
mvgetch	curses(S)	UX/L	-lcurses
mvgetch	terminfo(S)	UX/L	-lcurses
mvgetstr	curses(S)	UX/L	-lcurses
mvgetstr	terminfo(S)	UX/L	-lcurses
mvinch	curses(S)	UX/L	-lcurses
mvinch	tam(S)	U/L	-ltam, -lcurses
mvinch	terminfo(S)	UX/L	-lcurses
mvinsch	curses(S)	UX/L	-lcurses
mvinsch	terminfo(S)	UX/L	-lcurses
mvprintw	curses(S)	U/L	-lcurses
mvprintw	terminfo(S)	UX/L	-lcurses

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
mvscanw	curses(S)	U/L	-lcurses
mvscanw	terminfo(S)	UX/L	-lcurses
mvwaddch	curses(S)	UX/L	-lcurses
mvwaddch	terminfo(S)	UX/L	-lcurses
mvwaddstr	curses(S)	UX/L	-lcurses
mvwaddstr	terminfo(S)	UX/L	-lcurses
mvwdelch	curses(S)	UX/L	-lcurses
mvwdelch	terminfo(S)	UX/L	-lcurses
mvwgetch	curses(S)	UX/L	-lcurses
mvwgetch	terminfo(S)	UX/L	-lcurses
mvwgetstr	curses(S)	UX/L	-lcurses
mvwgetstr	terminfo(S)	UX/L	-lcurses
mvwin	curses(S)	UX/L	-lcurses
mvwin	terminfo(S)	UX/L	-lcurses
mvwinch	curses(S)	UX/L	-lcurses
mvwinch	terminfo(S)	UX/L	-lcurses
mvwinsch	curses(S)	UX/L	-lcurses
mvwinsch	terminfo(S)	UX/L	-lcurses
mvwprintw	curses(S)	U/L	-lcurses
mvwprintw	terminfo(S)	UX/L	-lcurses
mvwscanw	curses(S)	U/L	-lcurses
mvwscanw	terminfo(S)	UX/L	-lcurses
nap	nap(S)	UX/C	-lx
napms	curses(S)	U/L	-lcurses
nbra	regexp(S)	U/V	variable - regexp.h
nbwaitsem	waitsem(S)	UX/C	-lx
New	libwindows(S)	U/L	-lwindows
new_field	field(S)	U/L	-lform, -lcurses
new_fieldtype	fieldtype(S)	U/L	-lform, -lcurses
new_form	form(S)	U/L	-lform, -lcurses
new_item	item(S)	U/L	-lmenu, -lcurses
Newlayer	libwindows(S)	U/L	-lwindows
new_menu	menu(S)	U/L	-lmenu, -lcurses
newpad	curses(S)	U/L	-lcurses
newpad	terminfo(S)	UX/L	-lcurses
new_page	form(S)	U/L	-lform, -lcurses
new_panel	panel(S)	U/L	-lpanel, -lcurses
newterm	curses(S)	U/L	-lcurses
newterm	terminfo(S)	UX/L	-lcurses
newwin	curses(S)	UX/L	-lcurses
newwin	terminfo(S)	UX/L	-lcurses
_next_choice	fieldtype(S)	U/L	-lform
nextkey	dbm(S)	UX/L	-ldb
nice	nice(S)	UX/C	

(Continued on next page)

(S)

(Continued)

Routine name	Manual page	OS/Type	cc library option
nl	curses(S)	UX/L	-lcurses
nl	tam(S)	U/L	-ltam, -lcurses
nl	terminfo(S)	UX/L	-lcurses
nl_asctime	nl_ctime(S)	U/L	-lintl
nl_ctime	nl_ctime(S)	U/L	-lintl
nl_printf	nl_printf(S)	U/L	-lintl
nl_fscanf	nl_scanf(S)	U/L	-lintl
nl_init	nl_init(S)	U/L	-lintl
nlist	nlist(S)	UX/L	-lc
nl_langinfo	nl_langinfo(S)	U/L	-lintl
nl_printf	nl_printf(S)	U/L	-lintl
nl_scanf	nl_scanf(S)	U/L	-lintl
nl_sprintf	nl_printf(S)	U/L	-lintl
nl_sscanf	nl_scanf(S)	U/L	-lintl
nl_strcmp	nl_strcmp(S)	U/L	-lintl
nl_strncmp	nl_strcmp(S)	U/L	-lintl
nocbreak	curses(S)	U/L	-lcurses
nocbreak	tam(S)	U/L	-ltam, -lcurses
nocbreak	terminfo(S)	UX/L	-lcurses
nocrmode	curses(S)	UX/L	-lcurses
nodelay	curses(S)	U/L	-lcurses
nodelay	tam(S)	U/L	-ltam, -lcurses
nodelay	terminfo(S)	UX/L	-lcurses
noecho	curses(S)	UX/L	-lcurses
noecho	tam(S)	U/L	-ltam, -lcurses
noecho	terminfo(S)	UX/L	-lcurses
nonl	curses(S)	UX/L	-lcurses
nonl	terminfo(S)	UX/L	-lcurses
nonl	tam(S)	U/L	-ltam, -lcurses
noraw	curses(S)	UX/L	-lcurses
noraw	terminfo(S)	UX/L	-lcurses
notimeout	curses(S)	U/L	-lcurses
nrand48	drand48(S)	UX/L	-lc
open	open(S)	UX/C	
openagent	libwindows(S)	U/L	-lwindows
openchan	libwindows(S)	U/L	-lwindows
opendir	directory(S)	UX/L	-lc
openpl	plot(S)	U/L	-lplot
opensem	opensem(S)	UX/C	-lx
optarg	getopt(S)	UX/L	-lc
opterr	getopt(S)	UX/L	-lc
optind	getopt(S)	UX/L	-lc
overlay	curses(S)	UX/L	-lcurses
overlay	terminfo(S)	UX/L	-lcurses

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
overwrite	curses(S)	UX/L	-lcurses
overwrite	terminfo(S)	UX/L	-lcurses
paccess	paccess(S)	U/C	
pair_content	curses(S)	U/L	-lcurses
panel_above	panel(S)	U/L	-lpanel, -lcurses
panel_below	panel(S)	U/L	-lpanel, -lcurses
panel_hidden	panel(S)	U/L	-lpanel, -lcurses
panel_userptr	panel(S)	U/L	-lpanel, -lcurses
panel_window	panel(S)	U/L	-lpanel, -lcurses
passlen	passlen(S)	U/L	-lprot, -lm
pathconf	pathconf(S)	U/C	
pause	pause(S)	UX/C	
pb_check	tam(S)	U/L	-ltam, -lcurses
pb_empty	tam(S)	U/L	-ltam, -lcurses
pb_gbuf	tam(S)	U/L	-ltam, -lcurses
pb_gets	tam(S)	U/L	-ltam, -lcurses
pb_name	tam(S)	U/L	-ltam, -lcurses
pb_open	tam(S)	U/L	-ltam, -lcurses
pb_puts	tam(S)	U/L	-ltam, -lcurses
pb_seek	tam(S)	U/L	-ltam, -lcurses
pb_weof	tam(S)	U/L	-ltam, -lcurses
pclose	popen(S)	UX/L	-lc
pechochar	curses(S)	U/L	-lcurses
PEEKC	regexp(S)	UX/M	macro - regexp.h
perror	perror(S)	UX/L	-lc
phs_get	phs(S)	U/L	-lmmdf
phs_msg	phs(S)	U/L	-lmmdf
phs_note	phs(S)	U/L	-lmmdf
pipe	pipe(S)	UX/C	
plock	plock(S)	UX/L	-lc
pnoutrefresh	curses(S)	U/L	-lcurses
pnoutrefresh	terminfo(S)	UX/L	-lcurses
point	plot(S)	U/L	-lplot
poll	poll(S)	U/C	
popen	popen(S)	UX/L	-lc
pos_form_cursor	form(S)	U/L	-lform, -lcurses
pos_menu_cursor	menu(S)	U/L	-lmenu, -lcurses
post_form	form(S)	U/L	-lform, -lcurses
post_menu	menu(S)	U/L	-lmenu, -lcurses
pow	exp(S)	UX/L	-lm
prefresh	curses(S)	U/L	-lcurses
_prev_choice	fieldtype(S)	U/L	-lform
primary_auth	subsystems(S)	U/L	-lprot, -lx
primary_of_secondary_auth	subsystems(S)	U/L	-lprot, -lx

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
printf	printf(S)	UX/L	-lc
printw	curses(S)	UX/L	-lcurses
printw	tam(S)	U/L	-ltam, -lcurses
printw	terminfo(S)	UX/L	-lcurses
proct1	proct1(S)	UX/C	-lx
profil	profil(S)	UX/C	
ptrace	ptrace(S)	UX/C	
putc	putc(S)	UX/M	macro - stdio.h
putchar	putc(S)	UX/L	-lc
putdvagname	getdvagent(S)	U/L	-lprot, -lx
putenv	putenv(S)	UX/L	-lc
putmsg	putmsg(S)	U/C	
putp	curses(S)	U/L	-lcurses
putp	terminfo(S)	UX/L	-lcurses
putprdfname	getprdfent(S)	U/L	-lprot, -lx
putprfname	getprfient(S)	U/L	-lprot, -lx
putprpwnam	getprpwent(S)	U/L	-lprot, -lx
putprtname	getprtcent(S)	U/L	-lprot, -lx
putpwent	putpwent(S)	UX/L	-lc
puts	puts(S)	UX/L	-lc
pututline	getut(S)	UX/L	-lc
putw	putc(S)	UX/L	-lc
pw_idtoname	pw_nametoid(S)	U/L	-lprot
pw_nametoid	pw_nametoid(S)	U/L	-lprot
qsort	qsort(S)	UX/L	-lc
raise	raise(S)	U/L	-lc
rand	rand(S)	UX/L	-lc
randomword	randomword(S)	U/L	-lprot
raw	curses(S)	UX/L	-lcurses
raw	terminfo(S)	UX/L	-lcurses
rdchk	rdchk(S)	UX/C	-lx
read	read(S)	UX/C	
readdir	directory(S)	UX/L	-lc
readlink	readlink(S)	U/C	
read_pw_fields	fields(S)	U/L	-lprot, -lx
read_tc_fields	fields(S)	U/L	-lprot, -lx
realloc	malloc(S)	UX/L	-lc
realloc	malloc(S)	UX/L	-lmalloc
refresh	curses(S)	UX/L	-lcurses
refresh	tam(S)	U/L	-ltam, -lcurses
refresh	terminfo(S)	UX/L	-lcurses
regcmp	regcmp(S)	UX/L	-lintl
regex	regcmp(S)	UX/L	-lintl
remove	remove(S)	U/L	-lc

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
rename	rename(S)	U/C	
replace_file	dblock(S)	U/L	-lprot, -lx
replace_panel	panel(S)	U/L	-lpanel, -lcurses
reset_prog_mode	curses(S)	U/L	-lcurses
reset_shell_mode	curses(S)	U/L	-lcurses
resetterm	tam(S)	U/L	-ltam, -lcurses
resetterm	terminfo(S)	UX/L	-lcurses
reset_tty	curses(S)	U/L	-ltcap
resetty	curses(S)	UX/L	-lcurses
resetty	tam(S)	U/L	-ltam, -lcurses
resetty	terminfo(S)	UX/L	-lcurses
Reshape	libwindows(S)	U/L	-lwindows
restartterm	curses(S)	U/L	-lcurses
RETURN	regexp(S)	UX/M	macro - regexp.h
rewind	fseek(S)	UX/L	-lc
rewinddir	directory(S)	UX/M	macro - dirent.h
ripline	curses(S)	U/L	-lcurses
rmdir	rmdir(S)	U/C	
run_crypt	crypt(S)	U/L	-lcrypt
Runlayer	libwindows(S)	U/L	-lwindows
run_setkey	crypt(S)	U/L	-lcrypt
saveterm	terminfo(S)	UX/L	-lcurses
savetty	curses(S)	UX/L	-lcurses
savetty	tam(S)	U/L	-ltam, -lcurses
savetty	terminfo(S)	UX/L	-lcurses
sbrk	brk(S)	UX/L	-lc
sc_copyscstate	sc_raw(S)	U/L	-lsc_s
sc_exit	sc_init(S)	U/L	-lsc_s
sc_getfkeystr	sc_init(S)	U/L	-lsc_s
sc_getinfo	sc_raw(S)	U/L	-lsc_s
sc_getkbmap	sc_init(S)	U/L	-lsc_s
sc_getkeymap	sc_init(S)	U/L	-lsc_s
sc_getled	sc_init(S)	U/L	-lsc_s
sc_getscreenstate	sc_raw(S)	U/L	-lsc_s
sc_init	sc_init(S)	U/L	-lsc_s
sc_kb2mapcode	sc_readkb(S)	U/L	-lsc_s
sc_mapcode2kb	sc_readkb(S)	U/L	-lsc_s
sc_mapcode2str	sc_readkb(S)	U/L	-lsc_s
sc_mapin	sc_readkb(S)	U/L	-lsc_s
sc_mapinit	sc_init(S)	U/L	-lsc_s
sc_mapout	sc_readkb(S)	U/L	-lsc_s
sc_raw	sc_raw(S)	U/L	-lsc_s
sc_readkb	sc_readkb(S)	U/L	-lsc_s
sc_readmapcode	sc_readkb(S)	U/L	-lsc_s

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
sc_readstr	sc_readkb(S)	U/L	-lsc_s
sc_receive_kb	sc_init(S)	U/L	-lsc_s
sc_setfkeystr	sc_init(S)	U/L	-lsc_s
sc_setinfo	sc_raw(S)	U/L	-lsc_s
sc_setkeymap	sc_init(S)	U/L	-lsc_s
sc_setled	sc_init(S)	U/L	-lsc_s
sc_setscreenswitch	sc_raw(S)	U/L	-lsc_s
sc_str2kb	sc_readkb(S)	U/L	-lsc_s
sc_unraw	sc_raw(S)	U/L	-lsc_s
scale_form	form(S)	U/L	-lform, -lcurses
scale_menu	menu(S)	U/L	-lmenu, -lcurses
scanf	scanf(S)	UX/L	-lc
scanw	curses(S)	UX/L	-lcurses
scanw	terminfo(S)	UX/L	-lcurses
scoinfo	scoinfo(S)	U/C	
scr_dump	curses(S)	U/L	-lcurses
scr_init	curses(S)	U/L	-lcurses
scroll	curses(S)	UX/L	-lcurses
scroll	terminfo(S)	UX/L	-lcurses
scrollok	curses(S)	UX/L	-lcurses
scrollok	terminfo(S)	UX/L	-lcurses
scr_restore	curses(S)	U/L	-lcurses
sdenter	sdenter(S)	UX/C	-lx
sdfree	sdget(S)	UX/L	-lx
sdget	sdget(S)	UX/C	-lx
sdgetv	sdgetv(S)	UX/C	-lx
sdleave	sdenter(S)	UX/C	-lx
sdwaitv	sdgetv(S)	UX/C	-lx
secondary_auth	subsystems(S)	U/L	-lprot, -lx
sed	regexp(S)	U/V	variable - regexp.h
seed48	drand48(S)	UX/L	-lc
seekdir	directory(S)	UX/L	-lc
select	select(S)	UX/C	
semctl	semctl(S)	UX/L	-lc
semget	semget(S)	UX/L	-lc
semop	semop(S)	UX/L	-lc
set_auth_parameters	identity(S)	U/L	-lprot
setbuf	setbuf(S)	UX/L	-lc
set_current_field	form(S)	U/L	-lform, -lcurses
set_current_item	menu(S)	U/L	-lmenu, -lcurses
set_curterm	curses(S)	U/L	-lcurses
setdvagant	getdvagant(S)	U/L	-lprot, -lx
set_field_back	field(S)	U/L	-lform, -lcurses
set_field_buffer	field(S)	U/L	-lform, -lcurses

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
set_field_fore	field(S)	U/L	-lform, -lcurses
set_field_init	form(S)	U/L	-lform, -lcurses
set_field_just	field(S)	U/L	-lform, -lcurses
set_field_opts	field(S)	U/L	-lform, -lcurses
set_field_pad	field(S)	U/L	-lform, -lcurses
set_field_status	field(S)	U/L	-lform, -lcurses
set_field_term	form(S)	U/L	-lform, -lcurses
set_field_type	field(S)	U/L	-lform, -lcurses
set_fieldtype_arg	fieldtype(S)	U/L	-lform, -lcurses
set_fieldtype_choice	fieldtype(S)	U/L	-lform, -lcurses
set_field_userptr	field(S)	U/L	-lform, -lcurses
set_form_fields	form(S)	U/L	-lform, -lcurses
set_form_init	form(S)	U/L	-lform, -lcurses
set_form_opts	form(S)	U/L	-lform, -lcurses
set_form_page	form(S)	U/L	-lform, -lcurses
set_form_sub	form(S)	U/L	-lform, -lcurses
set_form_term	form(S)	U/L	-lform, -lcurses
set_form_userptr	form(S)	U/L	-lform, -lcurses
set_form_win	form(S)	U/L	-lform, -lcurses
setgid	setuid(S)	UX/C	
setgrent	getgrent(S)	UX/L	-lc
setgroups	setgroups(S)	U/C	
set_item_init	menu(S)	U/L	-lmenu, -lcurses
set_item_opts	item(S)	U/L	-lmenu, -lcurses
set_item_term	menu(S)	U/L	-lmenu, -lcurses
set_item_userptr	item(S)	U/L	-lmenu, -lcurses
set_item_value	item(S)	U/L	-lmenu, -lcurses
setjmp	setjmp(S)	UX/L	-lc
setkey	crypt(S)	U/L	-lcrypt
setlocale	setlocale(S)	U/L	-lc
setluid	setluid(S)	U/L	-lprot
set_menu_back	menu(S)	U/L	-lmenu, -lcurses
set_menu_fore	menu(S)	U/L	-lmenu, -lcurses
set_menu_format	menu(S)	U/L	-lmenu, -lcurses
set_menu_grey	menu(S)	U/L	-lmenu, -lcurses
set_menu_init	menu(S)	U/L	-lmenu, -lcurses
set_menu_items	menu(S)	U/L	-lmenu, -lcurses
set_menu_mark	menu(S)	U/L	-lmenu, -lcurses
set_menu_opts	menu(S)	U/L	-lmenu, -lcurses
set_menu_pad	menu(S)	U/L	-lmenu, -lcurses
set_menu_pattern	menu(S)	U/L	-lmenu, -lcurses
set_menu_sub	menu(S)	U/L	-lmenu, -lcurses
set_menu_term	menu(S)	U/L	-lmenu, -lcurses
set_menu_userptr	menu(S)	U/L	-lmenu, -lcurses

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
set_menu_win	menu(S)	U/L	-lmenu, -lcurses
set_new_page	form(S)	U/L	-lform, -lcurses
set_panel_userptr	panel(S)	U/L	-lpanel, -lcurses
setpgid	setpgid(S)	U/L	-lc
setpgrp	setpgrp(S)	UX/C	
setprdfent	getprdfent(S)	U/L	-lprot, -lx
setprfient	getprfient(S)	U/L	-lprot, -lx
setpriv	setpriv(S)	U/L	-lprot
setprpwent	getprpwent(S)	U/L	-lprot, -lx
setprtcent	getprtcent(S)	U/L	-lprot, -lx
setpwent	getpwent(S)	UX/L	-lc
setscrreg	curses(S)	U/L	-lcurses
setscrreg	terminfo(S)	UX/L	-lcurses
set_seed	seed(S)	U/L	-lprot
setsid	setsid(S)	U/L	-lc
setsyx	curses(S)	U/L	-lcurses
setterm	curses(S)	UX/L	-lcurses
setterm	terminfo(S)	UX/L	-lcurses
set_term	curses(S)	U/L	-lcurses
set_term	terminfo(S)	UX/L	-lcurses
set_top_row	menu(S)	U/L	-lmenu, -lcurses
set_tty	curses(S)	U/L	-lncap
setuid	setuid(S)	UX/C	
setupterm	curses(S)	U/L	-lcurses
setupterm	terminfo(S)	UX/L	-lcurses
setupterm	terminfo(S)	UX/L	-lcurses
setutent	getut(S)	UX/L	-lc
setvbuf	setbuf(S)	UX/L	-lc
sgetl	sputl(S)	UX/L	-lld
shmat	shmop(S)	UX/L	-lc
shmctl	shmctl(S)	UX/L	-lc
shmdt	shmop(S)	UX/L	-lc
shmget	shmget(S)	UX/L	-lc
show_panel	panel(S)	U/L	-lpanel, -lcurses
shutdn	shutdn(S)	UX/L	-lx
sigaction	sigaction(S)	U/C	
sigaddset	sigset(S)	U/L	-lc
sigdelset	sigset(S)	U/L	-lc
sigemptyset	sigset(S)	U/L	-lc
sigfillset	sigset(S)	U/L	-lc
sighold	sigsetv(S)	U/L	-lc
sigignore	sigsetv(S)	U/L	-lc
sigismember	sigset(S)	U/L	-lc
siglongjmp	sigsetjmp(S)	U/M	macro - setjmp.h

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
signal	signal(S)	UX/L	-lc
signgam	gamma(S)	U/V	variable - math.h
sigpause	sigsetv(S)	U/L	-lc
sigpending	sigpending(S)	U/C	
sigprocmask	sigprocmask(S)	U/C	
sigrelse	sigsetv(S)	U/L	-lc
sigsem	sigsem(S)	UX/C	-lx
sigset	sigsetv(S)	U/L	-lc
sigsetjmp	sigsetjmp(S)	U/M	macro - setjmp.h
sigsuspend	sigsuspend(S)	U/C	
sin	trig(S)	UX/L	-lm
sinh	sinh(S)	UX/L	-lm
S_ISBLK	stat(S)	U/M	macro - stat.h
S_ISCHR	stat(S)	U/M	macro - stat.h
S_ISDIR	stat(S)	U/M	macro - stat.h
S_ISFIFO	stat(S)	U/M	macro - stat.h
S_ISNAM	stat(S)	U/M	macro - stat.h
S_ISREG	stat(S)	U/M	macro - stat.h
sleep	sleep(S)	UX/L	-lc
slk_clear	curses(S)	U/L	-lcurses
slk_init	curses(S)	U/L	-lcurses
slk_label	curses(S)	U/L	-lcurses
slk_noutrefresh	curses(S)	U/L	-lcurses
slk_refresh	curses(S)	U/L	-lcurses
slk_set	curses(S)	U/L	-lcurses
slk_touch	curses(S)	U/L	-lcurses
space	plot(S)	U/L	-lplot
sprintf	printf(S)	UX/L	-lc
sputl	sputl(S)	UX/L	-lld
sqrt	exp(S)	UX/L	-lm
srand	rand(S)	UX/L	-lc
srand48	drand48(S)	UX/L	-lc
sscanf	scanf(S)	UX/L	-lc
ssignal	ssignal(S)	UX/L	-lc
standend	curses(S)	UX/L	-lcurses
standend	terminfo(S)	UX/L	-lcurses
standout	curses(S)	UX/L	-lcurses
standout	terminfo(S)	UX/L	-lcurses
start_color	curses(S)	U/L	-lcurses
starting_egid	identity(S)	U/L	-lprot
starting_euid	identity(S)	U/L	-lprot
starting_luid	identity(S)	U/L	-lprot
starting_rgid	identity(S)	U/L	-lprot
starting_ruid	identity(S)	U/L	-lprot

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
stat	stat(S)	UX/C	
statfs	statfs(S)	UX/C	
stdbl	isconv(S)	U/L	-lisam
stderr	stdio(S)	UX/V	variable - stdio.h
stdin	stdio(S)	UX/V	variable - stdio.h
stdout	stdio(S)	UX/V	variable - stdio.h
step	regexp(S)	UX/F	function - regexp.h
stfloat	isconv(S)	U/L	-lisam
stime	stime(S)	UX/C	
stint	isconv(S)	U/M	macro - isam.h
stlong	isconv(S)	U/L	-lisam
stopio	stopio(S)	U/L	-lprot
store	dbm(S)	UX/L	-ldbms
store_pw_fields	fields(S)	U/L	-lprot, -lx
store_tc_fields	fields(S)	U/L	-lprot, -lx
strcat	string(S)	UX/L	-lc
strchr	string(S)	UX/L	-lc
strcmp	string(S)	UX/L	-lc
strcoll	strcoll(S)	U/L	-lintl
strcpy	string(S)	UX/L	-lc
strcspn	string(S)	UX/L	-lc
strdup	string(S)	UX/L	-lc
strerror	strerror(S)	U/L	-lc
strftime	ctime(S)	U/L	-lintl
strftime	strftime(S)	U/L	-lintl
strlen	string(S)	UX/L	-lc
strncat	string(S)	UX/L	-lc
strncmp	string(S)	UX/L	-lc
strncoll	strcoll(S)	U/L	-lintl
strncpy	string(S)	UX/L	-lc
strxfrm	strcoll(S)	U/L	-lintl
strtoul	strtoul(S)	U/L	-lc
strpbrk	string(S)	UX/L	-lc
strchr	string(S)	UX/L	-lc
strspn	string(S)	UX/L	-lc
strstr	string(S)	UX/L	-lc
strtod	strtod(S)	UX/L	-lc
strtok	string(S)	UX/L	-lc
strtol	strtol(S)	UX/L	-lc
strxfrm	strcoll(S)	U/L	-lintl
subpad	curses(S)	U/L	-lcurses
subwin	curses(S)	UX/L	-lcurses
subwin	terminfo(S)	UX/L	-lcurses
swab	swab(S)	UX/L	-lc

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
symlink	symlink(S)	U/C	
sync	sync(S)	UX/C	
sysconf	sysconf(S)	U/C	
sys_errlist	perror(S)	UX/V	variable - string.h
sysfs	sysfs(S)	U/C	
sysi86	sysi86(S)	UX/C	
sys_nerr	perror(S)	UX/V	variable - string.h
system	system(S)	UX/L	-lc
t_accept	t_accept(S)	U/L	TLI= -lnsl, XTI= -lxti
t_alloc	t_alloc(S)	U/L	TLI= -lnsl, XTI= -lxti
t_bind	t_bind(S)	U/L	TLI= -lnsl, XTI= -lxti
t_close	t_close(S)	U/L	TLI= -lnsl, XTI= -lxti
t_connect	t_connect(S)	U/L	TLI= -lnsl, XTI= -lxti
t_errlist	t_error(S)	U/L	TLI= -lnsl, XTI= -lxti
t_errno	t_error(S)	U/L	TLI= -lnsl, XTI= -lxti
t_error	t_error(S)	U/L	TLI= -lnsl, XTI= -lxti
t_free	t_free(S)	U/L	TLI= -lnsl, XTI= -lxti
t_getinfo	t_getinfo(S)	U/L	TLI= -lnsl, XTI= -lxti
t_getstate	t_getstate(S)	U/L	TLI= -lnsl, XTI= -lxti
t_listen	t_listen(S)	U/L	TLI= -lnsl, XTI= -lxti
t_look	t_look(S)	U/L	TLI= -lnsl, XTI= -lxti
t_nerr	t_error(S)	U/L	TLI= -lnsl, XTI= -lxti
t_open	t_open(S)	U/L	TLI= -lnsl, XTI= -lxti
t_optmgmt	t_optmgmt(S)	U/L	-lnsl
t_rcv	t_rcv(S)	U/L	TLI= -lnsl, XTI= -lxti
t_rcvconnect	t_rcvconnect(S)	U/L	TLI= -lnsl, XTI= -lxti
t_rcvdis	t_rcvdis(S)	U/L	TLI= -lnsl, XTI= -lxti
t_rcvrel	t_rcvrel(S)	U/L	TLI= -lnsl, XTI= -lxti
t_rcvudata	t_rcvudata(S)	U/L	TLI= -lnsl, XTI= -lxti
t_rcvuderr	t_rcvuderr(S)	U/L	TLI= -lnsl, XTI= -lxti
t_snd	t_snd(S)	U/L	TLI= -lnsl, XTI= -lxti
t_snddis	t_snddis(S)	U/L	TLI= -lnsl, XTI= -lxti
t_sndrel	t_sndrel(S)	U/L	TLI= -lnsl, XTI= -lxti
t_sndudata	t_sndudata(S)	U/L	TLI= -lnsl, XTI= -lxti
t_sync	t_sync(S)	U/L	TLI= -lnsl, XTI= -lxti
t_unbind	t_unbind(S)	U/L	TLI= -lnsl, XTI= -lxti
tai_end	tai(S)	U/L	-lmndf
tai_get	tai(S)	U/L	-lmndf
tai_init	tai(S)	U/L	-lmndf
tan	trig(S)	UX/L	-lm
tanh	sinh(S)	UX/L	-lm
tcdrain	tcflow(S)	U/L	-lc
tcflow	tcflow(S)	U/L	-lc
tcflush	tcflow(S)	U/L	-lc

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
tcgetattr	tcattr(S)	U/L	-lc
tcgetpgrp	tcpgrp(S)	U/L	-lc
tcsendbreak	tcflow(S)	U/L	-lc
tcsetattr	tcattr(S)	U/L	-lc
tcsetpgrp	tcpgrp(S)	U/L	-lc
tdelete	tsearch(S)	UX/L	-lc
telldir	directory(S)	UX/L	-lc
tempnam	tmpnam(S)	UX/L	-lc
tfind	tsearch(S)	UX/L	-lc
tgetent	curses(S)	U/L	-lcurses
tgetent	termcap(S)	UX/L	-lcurses
tgetent	terminfo(S)	UX/L	-lcurses
tgetflag	curses(S)	U/L	-lcurses
tgetflag	termcap(S)	UX/L	-lcurses
tgetflag	terminfo(S)	UX/L	-lcurses
tgetnum	curses(S)	U/L	-lcurses
tgetnum	termcap(S)	UX/L	-lcurses
tgetnum	terminfo(S)	UX/L	-lcurses
tgetstr	curses(S)	U/L	-lcurses
tgetstr	termcap(S)	UX/L	-lcurses
tgetstr	terminfo(S)	UX/L	-lcurses
tgoto	curses(S)	U/L	-lcurses
tgoto	termcap(S)	UX/L	-lcurses
tgoto	terminfo(S)	UX/L	-lcurses
tigetflag	curses(S)	U/L	-lcurses
tigetnum	curses(S)	U/L	-lcurses
tigetstr	curses(S)	U/L	-lcurses
time	time(S)	UX/L	-lc
times	times(S)	UX/C	
timezone	ctime(S)	UX/V	variable - time.h
tmpfile	tmpfile(S)	UX/L	-lc
tmpnam	tmpnam(S)	UX/L	-lc
toascii	toascii(S)	UX/M	macro - ctype.h
toascii	ctype(S)	UX/M	macro - ctype.h
todigit	toascii(S)	UX/M	macro - ctype.h
todigit	ctype(S)	U/M	macro - ctype.h
toint	toascii(S)	U/M	macro - ctype.h
toint	ctype(S)	U/M	macro - ctype.h
_tolower	toascii(S)	UX/L	-lc
tolower	toascii(S)	UX/L	-lc
_tolower	ctype(S)	U/L	-lc
tolower	ctype(S)	UX/L	-lc
Top	libwindows(S)	U/L	-lwindows
top_panel	panel(S)	U/L	-lpanel, -lcurses

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
top_row	menu(S)	U/L	-lmenu, -lcurses
total_auths	subsystems(S)	U/L	-lprot, -lx
touchline	curses(S)	U/L	-lcurses
touchwin	curses(S)	UX/L	-lcurses
touchwin	terminfo(S)	UX/L	-lcurses
_toupper	toascii(S)	UX/L	-lc
toupper	toascii(S)	UX/L	-lc
_toupper	ctype(S)	U/L	-lc
toupper	ctype(S)	UX/L	-lc
tparm	curses(S)	U/L	-lcurses
tparm	terminfo(S)	UX/L	-lcurses
tputs	curses(S)	U/L	-lcurses
tputs	termcap(S)	UX/L	-lcurses
tputs	terminfo(S)	UX/L	-lcurses
traceoff	curses(S)	U/L	-lcurses
traceoff	terminfo(S)	UX/L	-lcurses
traceon	curses(S)	U/L	-lcurses
traceon	terminfo(S)	UX/L	-lcurses
track	tam(S)	U/L	-ltam
tsearch	tsearch(S)	UX/L	-lc
ttyname	ttyname(S)	UX/L	-lc
ttyslot	ttyslot(S)	UX/L	-lc
twalk	tsearch(S)	UX/L	-lc
typeahead	curses(S)	U/L	-lcurses
typeahead	terminfo(S)	UX/L	-lcurses
tzname	ctime(S)	UX/V	variable - time.h
tzset	ctime(S)	UX/L	-lc
uadmin	uadmin(S)	UX/C	
ulimit	ulimit(S)	UX/C	
umask	umask(S)	UX/C	
umount	umount(S)	UX/L	-lc
uname	uname(S)	UX/L	-lc
unctrl	curses(S)	UX/L	-lcurses
unctrl	terminfo(S)	UX/L	-lcurses
undial	dial(S)	UX/L	-lc
unexecseg	execseg(S)	UX/C	-lx
UNGETC	regexp(S)	UX/M	macro - regexp.h
ungetc	ungetc(S)	UX/L	-lc
ungetch	curses(S)	U/L	-lcurses
unlink	unlink(S)	UX/C	
unpost_form	form(S)	U/L	-lform, -lcurses
unpost_menu	menu(S)	U/L	-lmenu, -lcurses
update_panels	panel(S)	U/L	-lpanel, -lcurses
ustat	ustat(S)	UX/L	-lc

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
utime	utime(S)	UX/C	
utmpname	getut(S)	UX/L	-lc
va_alist	varargs(S)	UX/V	variable - stdarg.h
va_arg	varargs(S)	UX/M	macro - stdarg.h
va_dcl	varargs(S)	UX/V	variable - stdarg.h
va_end	varargs(S)	UX/M	macro - stdarg.h
va_list	varargs(S)	UX/V	variable - stdarg.h
va_start	varargs(S)	UX/M	macro - stdarg.h
vfprintf	vprintf(S)	UX/L	-lc
vidattr	curses(S)	U/L	-lcurses
vidattr	terminfo(S)	UX/L	-lcurses
vidputs	curses(S)	U/L	-lcurses
vidputs	terminfo(S)	UX/L	-lcurses
vldldptr	allocldptr(S)	U/L	-lld
vprintf	vprintf(S)	UX/L	-lc
vsprintf	vprintf(S)	UX/L	-lc
vwprintw	curses(S)	U/L	-lcurses
vwscanw	curses(S)	U/L	-lcurses
waddch	curses(S)	UX/L	-lcurses
waddch	terminfo(S)	UX/L	-lcurses
waddstr	curses(S)	UX/L	-lcurses
waddstr	terminfo(S)	UX/L	-lcurses
wait	wait(S)	UX/C	
waitpid	wait(S)	U/L	-lc
waitsem	waitsem(S)	UX/C	-lx
wattroff	curses(S)	U/L	-lcurses
wattroff	terminfo(S)	UX/L	-lcurses
wattron	curses(S)	U/L	-lcurses
wattron	terminfo(S)	UX/L	-lcurses
wattrset	curses(S)	U/L	-lcurses
wattrset	terminfo(S)	UX/L	-lcurses
wclear	curses(S)	UX/L	-lcurses
wclear	terminfo(S)	UX/L	-lcurses
wclrtobot	curses(S)	UX/L	-lcurses
wclrtobot	terminfo(S)	UX/L	-lcurses
wclrtoeol	curses(S)	UX/L	-lcurses
wclrtoeol	terminfo(S)	UX/L	-lcurses
wcmd	tam(S)	U/M	macro - tam.h
wcreate	tam(S)	U/L	-ltam, -lcurses
wcstombs	mblen(S)	U/L	-lc
wctomb	mblen(S)	U/L	-lc
wdelch	curses(S)	UX/L	-lcurses
wdelch	terminfo(S)	UX/L	-lcurses
wdelete	tam(S)	U/L	-ltam, -lcurses

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
wdeleteln	curses(S)	UX/L	-lcurses
wdeleteln	terminfo(S)	UX/L	-lcurses
wechochar	curses(S)	U/L	-lcurses
werase	curses(S)	UX/L	-lcurses
werase	terminfo(S)	UX/L	-lcurses
wexit	tam(S)	U/M	macro - tam.h
wgetc	tam(S)	U/M	macro - tam.h
wgetch	curses(S)	UX/L	-lcurses
wgetch	terminfo(S)	UX/L	-lcurses
wgetmouse	tam(S)	U/M	macro - tam.h
wgetpos	tam(S)	U/M	macro - tam.h
wgetsel	tam(S)	U/M	macro - tam.h
wgetstat	tam(S)	U/M	macro - tam.h
wgetstr	curses(S)	UX/L	-lcurses
wgetstr	terminfo(S)	UX/L	-lcurses
wgoto	tam(S)	U/M	macro - tam.h
wicoff	tam(S)	U/M	macro - tam.h
wicon	tam(S)	U/M	macro - tam.h
widest_auth	subsystems(S)	U/L	-lprot, -lx
winch	curses(S)	UX/L	-lcurses
winch	terminfo(S)	UX/L	-lcurses
wind	tam(S)	U/L	-ltam, -lcurses
winit	tam(S)	U/M	macro - tam.h
winsch	curses(S)	UX/L	-lcurses
winsch	terminfo(S)	UX/L	-lcurses
winserltn	curses(S)	UX/L	-lcurses
winserltn	terminfo(S)	UX/L	-lcurses
wlabel	tam(S)	U/M	macro - tam.h
wmove	curses(S)	UX/L	-lcurses
wmove	terminfo(S)	UX/L	-lcurses
wndelay	tam(S)	U/M	macro - tam.h
wnl	tam(S)	U/M	macro - tam.h
wnoutrefresh	curses(S)	U/L	-lcurses
wnoutrefresh	terminfo(S)	UX/L	-lcurses
wpostwait	tam(S)	U/M	macro - tam.h
wprexec	tam(S)	U/M	macro - tam.h
wprintf	tam(S)	U/M	macro - tam.h
wprintw	curses(S)	UX/L	-lcurses
wprintw	terminfo(S)	UX/L	-lcurses
wprompt	tam(S)	U/M	macro - tam.h
wputc	tam(S)	U/M	macro - tam.h
wputs	tam(S)	U/M	macro - tam.h
wrastop	tam(S)	U/M	macro - tam.h
wreadmouse	tam(S)	U/M	macro - tam.h

(Continued on next page)

(Continued)

Routine name	Manual page	OS/Type	cc library option
wrefresh	curses(S)	UX/L	-lcurses
wrefresh	tam(S)	U/L	-ltam, -lcurses
wrefresh	terminfo(S)	UX/L	-lcurses
write	write(S)	UX/C	
write_authorizations	subsystems(S)	U/L	-lprot, -lx
wscanw	curses(S)	UX/L	-lcurses
wscanw	terminfo(S)	UX/L	-lcurses
wselect	tam(S)	U/M	macro - tam.h
wsetmouse	tam(S)	U/M	macro - tam.h
wsetscreg	curses(S)	U/L	-lcurses
wsetscreg	terminfo(S)	UX/L	-lcurses
wsetstat	tam(S)	U/M	macro - tam.h
wsilk	tam(S)	U/M	macro - tam.h
wstandend	curses(S)	UX/L	-lcurses
wstandend	terminfo(S)	UX/L	-lcurses
wstandout	curses(S)	UX/L	-lcurses
wstandout	terminfo(S)	UX/L	-lcurses
wuser	tam(S)	U/M	macro - tam.h
xlist	xlist(S)	U/M	macro - x.out.h
y0	bessel(S)	UX/L	-lm
y1	bessel(S)	UX/L	-lm
yn	bessel(S)	UX/L	-lm

a64l, l64a

convert between long integer and base-64 ASCII string

Syntax

```
cc ... -lc

long a64l (s)
char*s;

char *l64a (l)
long l;
```

Description

These functions are used to maintain numbers stored in **base-64** ASCII characters. This is a notation by which long integers can be represented by up to six characters; each character represents a "digit" in a radix-64 notation.

The characters used to represent "digits" are "." for 0, "/" for 1, "0" through "9" for 2-11, "A" through "Z" for 12-37, and "a" through "z" for 38-63.

The **a64l** function takes a pointer to a null-terminated base-64 representation and returns a corresponding *long* value. If the string pointed to by *s* contains more than six characters, **a64l** will use the first six.

The **a64l** function scans the character string from left to right, decoding each character as a 6-bit Radix 64 number.

The **l64a** function takes a *long* argument and returns a pointer to the corresponding base-64 representation. If the argument is 0, **l64a** returns a pointer to a null string.

Note

The value returned by **l64a** is a pointer into a static buffer, the contents of which are overwritten by each call.

Standards conformance

a64l and **l64a** are conformant with:
AT&T SVID Issue 2.

abort

generate an abort fault

Syntax

```
cc ... -lc

#include <stdlib.h>

void abort ( )
```

Description

The **abort** function closes all open files, streams, and directory streams. This is the same as **exit(S)**, but instead of just exiting, **abort** causes **SIGABRT** to be sent to the calling process. If **SIGABRT** is neither caught nor ignored, all **stdio(S)** streams are flushed prior to the signal being sent, and a core dump results.

See also

exit(S), **signal(S)**

Diagnostics

If **SIGABRT** is neither caught nor ignored, and the current directory is writable, a core dump is produced and the message "abort - core dumped" is written by the shell.

Standards conformance

abort is conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
ANSI X3.159-1989 Programming Language — C;
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

abs

return integer absolute value

Syntax

```
cc ... -lc

#include <stdlib.h>

int abs (i)
int i;
```

Description

The `abs` function returns the absolute value of its integer operand.

See also

`fabs(S)`, `floor(S)`

Note

In two's-complement representation, the absolute value of the negative integer with largest magnitude is undefined.

Standards conformance

`abs` is conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
ANSI X3.159-1989 Programming Language — C;
Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

acceptable_password

determine if password is cryptic

Syntax

```
cc ... -lprot

#include <sys/types.h>
#include <sys/security.h>
#include <sys/audit.h>
#include <prot.h>

int acceptable_password (word, stream)
char *word;
FILE *stream;
```

Description

This routine determines if the given password is reasonable; that is, it is very hard to deduce from a number of heuristic means. The cleartext (plaintext) password is passed in as the argument *word* and the second argument, *stream*, is a file pointer to report failure-reason messages. If this checking is to be silent, the second argument should be a NULL file pointer so that no messages will be output to the stream.

When `acceptable_password` returns a 1, the password provided meets all the tests described below. When it returns a 0, the password failed to meet at least one of the tests or an internal error occurred.

The selectivity criteria for the password include but may not be limited to the following four tests:

palindrome	This test is passed if the word is not a palindrome. (A palindrome is a word that is spelled the same backwards as forwards.) Examples of palindromes that fail on this test are: mom, dad, noon, redivider, radar. Palindromes do not make good passwords because they reduce an n character password to $n/2 + 1$ characters. A penetrator knowing that palindromes were legal could use heuristics that could deduce the password much more quickly than if they were excluded.
login name	This test is passed if the password is not a system login name or a closely related string. Unsecure systems allow passwords to be the login name itself. This is a fact known by many penetrators. All login names are excluded because a user that is the owner of several pseudo-user accounts may elect to use the login name of one account as the password for all his accounts.

group name	Similar to the login name issue, this test is passed if the password is not a group name or a closely related string.
English word	This test is passed if the <code>spell(C)</code> program determines that the supplied password is not an English word so that a penetrator cannot search the on-line dictionary to find the password. The <code>spell</code> program also has some built-in rules that go beyond the actual on-line dictionary in determining what is a proper word which this routine takes advantage of.

Return value

When a supplied password meets all of the criteria, the `acceptable_password` returns a value of 1. If the supplied password fails one of the tests or if an internal error occurs, a value of 0 is returned by the routine.

Note

This routine works correctly only if `set_auth_parameters` is called as the first item in `main()`.

Files

`/etc/passwd`
`/etc/group`
`/usr/lib/spell/*`

See also

`spell(C)`, `getpwent(S)`, `getgrent(S)`

Standards conformance

`acceptable_password` is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

access, eaccess

determine accessibility of a file

Syntax

```
cc ... -lc

#include <unistd.h>

int access (path, amode)
char *path;
int amode;

int eaccess (path, amode)
char *path;
int amode;
```

Description

The **access** function checks the file named by *path* for accessibility according to the bit pattern contained in *amode*, using the real user ID in place of the effective user ID and the real group ID in place of the effective group ID. **access** should only be used by set-UID and set-GID programs; **access** is not equivalent to **eaccess** even when the program is not being used by set-UID and set-GID.

eaccess is similar to **access** except that the effective user ID and group ID are used.

The value of *amode* is either the bitwise inclusive OR of the access permissions to be checked or the existence test as follows:

Name	Description
R_OK	test for <i>read</i> permission
W_OK	test for <i>write</i> permission
X_OK	test for <i>execute (search)</i> permission
F_OK	test for existence of file

These values are defined in the *unistd.h* header file.

If any access permissions are to be checked, each is checked individually.

Access to the file is denied if one or more of the following is true:

- [EACCES] Search permission is denied on a component of the path prefix, or permission bits of the file mode do not permit the requested access.
- [EINTR] A signal was caught during the access system call.
- [EINVAL] The value of *amode* is invalid.
- [EMULTIHOP] Components of *path* require hopping to multiple remote machines.
- [ENAMETOOLONG] The length of the *path* argument exceeds PATH_MAX or a pathname component is longer than NAME_MAX while _POSIX_NO_TRUNC is in effect.
- [ENOENT] The *path* argument points to the name of a file that does not exist or the *path* points to an empty string.
- [ENOLINK] *path* points to a remote machine and the link to that machine is no longer active.
- [ENOTDIR] A component of the path prefix is not a directory.
- [EROFS] Write access is requested for a file on a read-only file system.
- [ETXTBSY] Write access is requested for a pure procedure (shared text) file that is being executed.

The owner of a file has permission checked with respect to the “owner” read, write, and execute mode bits. Members of the file’s group other than the owner have permissions checked with respect to the “group” mode bits, and all others have permissions checked with respect to the “other” mode bits.

See also

chmod(S), stat(S)

Diagnostics

If the requested access is permitted, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

Standards conformance

access is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

eaccess is not part of any currently supported standard; it is an extension of AT&T System V provided by The Santa Cruz Operation.

acct

enable or disable process accounting

Syntax

```
cc ... -lc

int acct (path)
char *path;
```

Description

acct is used to enable or disable the system process accounting routine. If the routine is enabled, an accounting record is written on an accounting file for each process that terminates. Termination can be caused by one of two things: an **exit** call or a signal. (See **exit(S)** and **signal(S)**.) The effective user ID of the calling process must be superuser to use this call.

path points to a pathname naming the accounting file. The accounting file format is given in **acct(FP)**.

The accounting routine is enabled if *path* is non-zero and no errors occur during the system call. It is disabled if *path* is zero and no errors occur during the system call.

acct fails if one or more of the following is true:

[EACCES]	The file named by <i>path</i> is not an ordinary file.
[EBUSY]	An attempt is being made to enable accounting when it is already enabled.
[EFAULT]	<i>path</i> points to an illegal address.
[ENOENT]	One or more components of the accounting file path name do not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The effective user of the calling process is not super-user.
[EROFS]	The named file resides on a read-only file system.

See also

acct(FP), **exit(S)**, **signal(S)**

Diagnostics

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

Standards conformance

acct is conformant with:

AT&T SVID Issue 2.

alarm

set a process alarm clock

Syntax

```
cc ... -lc

unsigned int alarm (seconds)
unsigned int seconds;
```

Description

The **alarm** function causes the system to send the calling process a SIGALRM signal after the number of real time *seconds* has elapsed. If *seconds* is zero (0), a pending alarm request, if any, is cancelled. Processor scheduling delays may prevent the process from handling the signal as soon as it is generated.

Refer to **signal(S)** for more information on SIGALRM.

alarm requests are not stacked; successive calls reset the alarm clock of the calling process.

The **fork(S)** system call sets the alarm clock of a child process to 0. A process created by the **exec(S)** family of calls inherits the time left on the old process's alarm clock.

See also

exec(S), **fork(S)**, **getitimer(S)**, **signal(S)**, **sigaction(S)**

Diagnostics

The **alarm** function is always successful and no return value is reserved to indicate an error. This function returns the amount of time, in seconds, remaining before the system is scheduled to generate the SIGALRM signal from the previous call to **alarm**, or zero if there was no previous alarm request.

Standards conformance

alarm is conformant with:

X/Open Portability Guide, Issue 3, 1989;
 IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
 and NIST FIPS 151-1.

assert

verify program assertion

Syntax

```
#include <assert.h>

void assert (expression)
int expression;
```

Description

The **assert** macro is inserted into a program to cause it to fail if an event occurs. This macro is used to debug a program. **assert** tests *expression* and if the expression is false (compares equal to zero), information is written to **stderr** about the call that failed and **abort(S)** is called to stop execution of the program.

The information written to **stderr** is in the form:

Assertion failed: expression, file xyz, line nnn

Where, *xyz* is the name of the source file and *nnn* the source line number of the **assert** statement. The name of the source file is taken from the **__FILE__** preprocessing macro, and the source file line number is taken from the **__LINE__** preprocessing macro.

Assertions can be stopped from being compiled into the program by:

1. Compiling with the **-DNDEBUG** preprocessor option See **cpp(CP)**.
2. Inserting the **#define NDEBUG** preprocessor control statement before the **#include <assert.h>** statement

Diagnostics

No return values or errors are defined.

See also

abort(S), **cpp(CP)**

Notes

If the macro **_KR** is defined, then **assert** is implemented as a macro. This macro places the *expression* into a string using quotation marks thus if the *expression* contains a string literal, the statement will not compile.

Standards conformance

assert is conformant with:

X/Open Portability Guide, Issue 3, 1989; and ANSI X3.159-1989 Programming Language — C.

(S)

atexit

register function to be called at termination

Syntax

```
cc ... -lc

#include <stdlib.h>

int atexit(func)
void (*func)(void);
```

Description

The `atexit` function is passed the address of a function (*func*) to be called when the program terminates normally. Successive calls to `atexit` create a register of functions that are executed “last in, first out.” No more than 32 functions can be registered with `atexit`. The functions passed to `atexit` cannot take parameters.

Return value

`atexit` returns **zero** on success or **-1** on failure.

See also

`abort(S)`, `exit(S)`

Example

```
#include <stdlib.h>
#include <stdio.h>

main( )
{
    int fn1( ), fn2( ), fn3( ), fn4( );
    atexit(fn1);
    atexit(fn2);
    atexit(fn3);
    atexit(fn4);
    printf("This is executed first.\n");
}

int fn1( )
{
    printf("next.\n");
}

int fn2( )
{
    printf("executed ");
}

int fn3( )
{
    printf("is ");
}

int fn4( )
{
    printf("This ");
}
```

Output:

```
This is executed first.
This is executed next.
```

This program pushes four functions onto the stack of functions to be executed when `atexit` is called. When the program exits, these programs are executed on a last-in, first-out basis.

Standards conformance

`atexit` is conformant with:

ANSI X3.159-1989 Programming Language — C.

atof, atoi, atol

converts ASCII to numbers

Syntax

```
cc ... -lc

#include <stdlib.h>

double atof (nptr)
char *nptr;

int atoi (nptr)
char *nptr;

long atol (nptr)
char *nptr;
```

Description

These functions convert a string pointed to by *nptr* to double, integer, and long integer numbers respectively. The first unrecognized character ends the string.

The **atof** function converts a string to a double. The string pointed to by *nptr* is converted up to the first character deemed to be inconsistent with the format of a floating-point number. Leading white-space characters are ignored. This call is analogous to:

```
strtod(str, (char **) NULL)
```

atof recognizes a string of the form:

```
[ +|- ] digits[. digits ][ e|E [ +|- ] digits ]
```

where the digits are contiguous decimal digits. Any number of tabs and spaces may precede the string. The "+" and "-" signs are optional. Either e or E may be used to mark the beginning of the exponent. If the **atof** function fails due to overflow or underflow, **errno** is set to **ERANGE**. The maximum size of an exponent is set by **DMAXEXP**, which is defined in the */usr/include/values.h* header file.

The **atoi** function converts the string pointed to by *nptr* to an integer. Processing stops when up to the first character inconsistent with the format of a decimal integer is encountered. Leading white-space characters is ignored. This call is analogous to the following except that an integer is returned.

```
strtol(str, (char **) NULL, 10)
```

atoi and **atol** recognize strings of the form:

[+|-] digits

where the digits are contiguous decimal digits. Any number of tabs and spaces may precede the string. The "+" and "-" signs are optional. If the **atoi** routine fails due to overflow or underflow, **errno** is set to **ERANGE**.

The **atol** function converts the string pointed to by *nptr* up to the first character inconsistent with the format of a decimal integer, ignoring leading white-space characters, to a long. This call is analogous to:

strtol(str, (char **) NULL, 10)

If the **atoi** function fails due to overflow or underflow, **errno** is set to **ERANGE**.

Note

If a value cannot be represented, the behavior is undefined.

See also

scanf(S)

Standards conformance

atof, **atoi** and **atol** are conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);

and ANSI X3.159-1989 Programming Language — C.

audit_close, audit_open, audit_read

open and access audit session data records

Syntax

```
cc ... -lprot

int audit_close( )

int audit_open (session)
int session;

struct audit_header *audit_read( )
```

Description

The **audit_open** routine provides an interface for opening an audit data session which has been previously collected on the system. The routine requires one argument, *session*, which indicates the audit session number. This may be acquired from a session list by using the **auditsh(ADM)** utility.

Once a session is open, **audit_read** sequentially retrieves audit records from the audit session data files. Each call returns a pointer to the next audit record header which identifies the record size, the record type, the event type, and other audit related information. The actual record formats are defined in **audit(HW)**. The routine returns a NULL pointer if an error or end-of-file occurs.

Before another session may be accessed, use **audit_close** to terminate processing for the current session. Another session may then be opened.

Return value

Upon successful completion, **audit_open** and **audit_close** return a value of "0". Otherwise, they return a value of "-1" with **errno** set to indicate the appropriate error. **audit_read** returns a pointer to the next audit record on success. **audit_read** returns a NULL if an error occurs or EOF is encountered.

See also

auditsh(ADM), audit(HW), authaudit(S)

Standards conformance

audit_close, audit_open and **audit_read** are not part of any currently supported standard; they are extensions of AT&T System V provided by the Santa Cruz Operation.

authaudit

produce audit records due to authentication events

Syntax

cc ... -lprot

```
#include <sys/types.h>
#include <sys/security.h>
#include <sys/audit.h>
#include <prot.h>

int audit_adjust_mask (pr)
struct pr_passwd *pr;

void audit_auth_entry (desired_entry, type, problem)
char *desired_entry;
int type;
char *problem;

void audit_lax_file (path, problem)
char *path;
char *problem;

int audit_lock (name, code, trys)
char *name;
int code;
int trys;

int audit_login (pr, pwd, terminal_name, code)
struct pr_passwd *pr;
struct passwd *pwd;
char *terminal_name;
int code;

void audit_no_resource (resource, object, problem)
char *resource;
int object;
char *problem;

int audit_passwd (name, code)
char *name;
int code;

void audit_security_failure (object, expected, curr, action, result)
int object;
long expected;
long curr;
char *action;
char *result;
```

(S)

```

void audit_subsystem (subsystem_type, action, result)
int subsystem_type;
char *action;
char *result;

```

Description

These routines provide standard interfaces to the secure audit facility from routines and programs that manipulate the Authentication database. Depending on the circumstances (UIDs, privileges), they either write directly to the audit special device */dev/auditw*, or they pass the information onto the *dlvr_audit(ADM)* utility.

audit_adjust_mask

If the user has a special mask, present it here to the audit subsystem.

audit_auth_entry

produces an audit record noting that the name *desired_entry* has a *problem* in the *type* database, which is a component database (one of: */etc/passwd*, */etc/group*, Protected Password database, Terminal Control database, File control database, Command Control database, System Default database, Subsystem database) of the Authentication database. This is the means to report a database inconsistency in an entry of the appropriate database.

audit_lax_file

produces an audit record about the file *path* and the exact problem that makes the file differ from the File Control database entry describing it. This is the means to report a breakdown of a sanity check on the proper setup of system files.

audit_lock Audit both successful and unsuccessful database locking operations. This audit record is always written directly to the audit device because the programs that must use locking (for writing) are run as the superuser, who can access the device.

audit_login

Audit both successful and unsuccessful logins. This audit record is always written directly to the audit device because login is run as the superuser, who can access the device.

audit_no_resource

prints an audit record that says the *resource* could not be obtained of system type *object*, as defined in *<sys/audit.h>*. The *problem* that results is also recorded. Typically, this is used to denote that a vital resource like memory could not be allocated and a security operation had to be aborted.

audit_passwd

Audit both successful and unsuccessful password changes. This audit record is always written directly to the audit device because login is run as the superuser, who can access the device.

audit_security_failure

records a problem involving a system *object*, as defined in the *sys/audit.h* header file. Even though kernel auditing may have recorded the same security problem as one or a series of failed system calls, this routine produces an audit record specifically noting the high-level security problem in terms of trusted entity failure. If appropriate, the *expected* and current *curr* values are recorded to further help in diagnosing the problem. The high level *action* attempted and the *level* of the failure are required. This is the means to report a high-level security problem that prevents or impedes the correct operation of a trusted process or subsystem. If the trusted process detects and corrects security problems, the invocation of this routine is the detection component of that mechanism.

audit_subsystem

records an audit record for high-level security events specific to a subsystem as defined by *subsystem_type* defined in *<sys/audit.h>*. The high-level *action* and either positive or negative *result* is recorded. This is the means to report a problem or significant event in a specific subsystem.

Notes

These routines work correctly only if *set_auth_parameters(S)* (described on the *identity(S)* manual page) is called as the first item in the *main ()* routine.

Files

/dev/auditw

See also

audit(HW), *identity(S)*

Standards conformance

authaudit is an extension of AT&T System V provided by the Santa Cruz Operation.

authcap

get information from the authentication database

Syntax

```
cc ... -lprot -lx

#include <sys/types.h>
#include <sys/security.h>
#include <sys/audit.h>
#include <prot.h>

int agetcommand (filepos, fd)
long *filepos;
int fd;

int agetdefault (filepos, fd)
long *filepos;
int fd;

int agetfile (filepos, fd)
long *filepos;
int fd;

int agetflag (id)
char *id;

int agettty (filepos, fd)
long *filepos;
int fd;

char *agetstr (id, area)
char *id;
char **area;

int agetuser (user)
char *user;

void asetdefaults (new_defaults)
int new_defaults;
```

Description

agetcommand - find next command entry in authcap file
agetdefault - find next default entry in authcap file
agetfile - find next file entry in authcap file
agetflag - returns id flag
agettty - find next tty entry in authcap file
agetuser - find entry with user name
agetstr - returns string associated with id
asetdefaults - change authcap database used to get system defaults

These routines manage ASCII database files in hierarchies as described in **authcap(F)**. All program references to the database should use these routines. Also, these routines are the ones that understand the **authcap** format and are guaranteed to provide the same interface, even if the database format or name designations change.

agetuser finds the entry associated with the user name argument. **agetcommand** finds the next command entry in the file described by *fd* starting at the file position *filepos* within the file. *filepos* should point to the beginning of a valid entry or at the end of the file. **agetfile** finds the next file entry in the file in the same way as **agetcommand**. Similarly, **agettty** finds the next tty entry in the file in the same way as **agetcommand**. Tty names are the components without the */dev* part. Example tty names in the database are *console*, and *tty3*. **agetdefault** finds the next default entry in the file in the same way as **agetcommand**.

Each of the above routines returns a status indicator. A return value of "1" means the entry was found. A return value of 0 means the entry was not found in the file. The calls with the *filepos* argument may update the position referenced by *filepos*.

Once one of the above routines obtains an entry, the next three routines obtain capabilities from that most recently chosen entry. **agetstr** returns the string associated with the id argument. The place the string goes is referenced by the area argument. Not the area argument is a *pointer* to an allocated string, not merely a string. The 0 pointer ((char *) 0) is returned when the capability cannot be found. A null string is returned as an empty string (**area == '\0'). **agetflag** returns the flag associated with the id argument. If the flag is set, "1" is returned. If the flag is not set (the "@" attribute appears with the id in the file), "0" is returned. If the capability flag cannot be found at all (different from not set), "-1" is returned.

asetdefaults changes the database used to find system defaults. It can be set to one of the following values for all system references to the database until it is reset again by **asetdefaults**. Initially it is **NORMAL**, where system values are found in the site-selectable part of the database. The value **WEAK** uses an

authcap(S)

insecure version, while the value **STRONG** uses a secure version of the system-wide values. When none of these values is used, the value **STRONG** is used.

Notes

agetuser, **agetcommand**, **agetfile**, **agettty** and **agetdefault** make use of static areas for an entry. Successive calls to any of these routines overwrites data that is used by later calls to **agetstr** and **agetflag**.

See also

authcap(F), **fields(S)**, **getprfient(S)**, **getprpwent(S)**, **getprtcent(S)**

Standards conformance

authcap is an extension of AT&T System V provided by the Santa Cruz Operation.

bessel: j0, j1, jn, y0, y1, yn

bessel functions

Syntax

```
cc ... -lm

#include <math.h>

double j0 (x)
double x;

double j1 (x)
double x;

double jn (n, x)
int n;
double x;

double y0 (x)
double x;

double y1 (x)
double x;

double yn (n, x)
int n;
double x;
```

Description

j0 and **j1** return Bessel functions of x of the first kind of orders 0 and 1 respectively. **jn** returns the Bessel function of x of the first kind of order n .

y0 and **y1** return Bessel functions of x of the second kind of orders 0 and 1 respectively. **yn** returns the Bessel function of x of the second kind of order n . The value of x must be positive.

Diagnostics

Non-positive arguments cause **y0**, **y1**, and **yn** to return the value **-HUGEi_VAL** and to set **errno** to **EDOM**. In addition, a message indicating **DOMAIN** error is printed on the standard error output.

Arguments too large in magnitude cause **j0**, **j1**, **y0**, and **y1** to return zero and to set **errno** to **ERANGE**. In addition, a message indicating **TLOSS** error is printed on the standard error output.

These error-handling procedures may be changed with the function **matherr(S)**.

bessel(S)

See also

matherr(S)

Standards conformance

j0, j1, jn, y0, y1 and yn are conformant with:

AT&T SVID Issue 2;

and X/Open Portability Guide, Issue 3, 1989.

brk, sbrk

change data segment space allocation

Syntax

```
cc ... -lc

int brk (endds)
char *endds;

char *sbrk (incr)
int incr;
```

Description

The **brk** and **sbrk** system calls are used to change dynamically the amount of space allocated for the calling process's data segment (see **exec(S)**). The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases. Newly allocated space is set to zero. If, however, the same memory space is reallocated to the same process, its contents are undefined.

The **brk** system call sets the break value to *endds* and changes the allocated space accordingly.

The **sbrk** system call adds *incr* bytes to the break value and changes the allocated space accordingly. *incr* can be negative, in which case the amount of allocated space is decreased.

The **brk** and **sbrk** system calls fail without making any change in the allocated space if one or more of the following is true:

- [EAGAIN] Total amount of system memory available for a read during physical IO is temporarily insufficient (see **shmop(S)**). This may occur even though the space requested was less than the system-imposed maximum process size.
- [ENOMEM] Use of **brk** or **sbrk** would result in more space being allocated than is allowed by the system-imposed maximum process size. To be able to access more memory, use **configure(ADM)** to change the **MAXUMEM** system variable.

Return value

Upon successful completion **brk** returns a value of 0, and **sbrk** returns the old break value. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

See also

end(S), exec(S), shmop(S)

Standards conformance

brk and **sbrk** conform to:

AT&T SVID Issue 2 and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2), but **brk** has been withdrawn from X/Open Portability Guide, Issue 3, 1989.

brkctl

allocates data in a 286 far segment

Syntax

```
cc ... -lbrkctl

#include <sys/brk.h>

char far *brkctl (command, increment, ptr)
int command;
long increment;
char far *ptr;
```

Description

The **brkctl** system call allocates and deallocates memory in additional data segments in small and middle model 286 programs. For the C compiler to make use of the return values in small and middle model programs, **brkctl** must be declared to return a far pointer. To enable the “far” keyword for small model C programs, the **-Me** option to the compiler must be used. Middle model C programs require the **-Mme** option.

command is **BR_ARGSEG**, **BR_FREESEG**, **BR_NEWSEG**, or **BR_IMPSEG**.

increment is a signed long increment. If positive, it must be less than 64K; if negative, its absolute value must be less than the sum of the total memory in all far segments plus the amount allocated in the near segment after process creation.

ptr is used only when **command** is **BR_ARGSEG**.

If **increment** is positive, **brkctl** returns a far pointer to the base of at least **increment** number of bytes of memory (see box on next page).

If the **command** is **BR_IMPSEG**, and a negative **increment** causes one or more segments to be freed, the ‘segment in question’ (see the **Return Values** section) is the last remaining segment that was not freed. **BR_IMPSEG** implies the use of the last data segment. Unless the process is small or middle model and currently has only one data segment, a positive **increment** that would overflow the last data segment causes a new segment to be allocated.

If the **command** is **BR_ARGSEG**, the **increment** may not be more negative than the size of the segment. The third argument (**ptr**), is assumed to be a far pointer in all models; the offset portion is never used.

If the **command** is **BR_NEWSEG**, the **increment** may not be negative at all. Any memory allocated is guaranteed to be at the base of a new segment.

BR_FREESEG frees memory allocated by a previous call to **BR_NEWSEG**. A use of the command is:

```
if ((fp=(long)brkctl(BR_FREESEG, (long)sizeof(int)*921,fp0)) == -1)
{
    /* error */
}
```

Return value

brkctl almost always returns a far pointer to the base of the affected region, (char far *)-1 on error.

When the *increment* is greater than 0, the return value is a pointer to the base of the newly allocated memory.

When the *increment* is less than or equal to 0, the return value is a pointer to the first illegal byte in the segment in question (usually the base of the deallocated memory). If that segment is full (exactly 64K bytes), the return value is a pointer to the base of the next segment (which may or may not exist).

Command	Increment	Ptr	Action
BR_ARGSEG	0	<valid far ptr>	report on segment
BR_ARGSEG	other	<valid far ptr>	increment specified segment
BR_NEWSEG	0	-	allocate new segment, size = 0
BR_NEWSEG	other	-	allocate new segment, size = increment
BR_IMPSEG	0	-	report on last segment; may free up empty segment(s)
BR_IMPSEG	other	-	increment last segment; on large model (or small and middle model with multiple data segments) may allocate new segment

Example

The example of `brkctl` below uses the `BR_NEWSEG` parameter to allocate space for 20,000 integers in a far data segment (on a 286 machine) and fills this memory with the integers from 1 to 20,000. Remember to compile this program with the `cc -Me` option.

```
#include <sys/brk.h>

#define FNULL (int far*)0

#ifdef M_I386
#define FAILURE (int *)-1
#else
#define FAILURE (int far*)-1
#endif

main()
{
    int i,j;

#ifdef M_I386
    int *fp, *brkctl();
#else
    int far *fp, far *brkctl(); /* both fars are necessary */
#endif

    fp=brkctl(BR_NEWSEG, (long)sizeof(int)*20000, FNULL);
    if (fp==FAILURE){
        perror("brkctl failed");
        exit(1);
    }
    for (i=0; i<20000; ++i)
        fp[i]=i+1;
    for (i=0; i<20000; ++i)
        printf("%d\n", fp[i]);
}
```

Notes

The `brkctl` system call should be used only for dynamically allocating additional segments in 80286 small and middle model programs. All other uses should be avoided in favor of `sbrk(S)`, `malloc(S)`, and other standard system services. The functionality of `brkctl` may change in future releases.

`brkctl` is currently available only in protected mode.

In all models, the "near" data segment must be the first data segment.

`brkctl` calls with `BR_IMPSEG` and a negative *increment* that would affect a shared data segment are refused.

brkctl(S)

See also

cc(CP), **ld**(CP), **machine**(M), **malloc**(S), **sbrk**(S)

Standards conformance

brkctl is an extension of AT&T System V provided by the Santa Cruz Operation.

bsearch

binary search a sorted table

Syntax

```
cc ... -lc

#include <stdlib.h>

void *bsearch (key, base, nel, width, compar)
void *key, *base;
size_t nel, width;
int (*compar)();
```

SVID Syntax

```
#include <search.h>

char *bsearch (key, base, nel, width, compar)
void *key, *base;
size_t nel, width;
int (*compar)();
```

Description

The **bsearch** function is a binary search routine. It returns a pointer into a table indicating where a datum may be found. The table must be previously sorted in increasing order according to a provided comparison function. *key* points to a datum instance to be sought in the table. *base* points to the element at the base of the table. *nel* is the number of elements in the table. *compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero if the first argument is to be considered less than, equal to, or greater than the second.

Note

For compatibility with the System V Interface Definition (SVID), include the *search.h* header file instead of *stdlib.h*.

Return value

The **bsearch** function returns a pointer to a matching member of the array, or a null pointer if no match is found. If two or more members compare equal, which member returned is unspecified.

Example

The example below searches a table containing pointers to nodes consisting of a string and its length. The table is ordered alphabetically on the string in the node pointed to by each entry.

This code fragment reads in strings and either finds the corresponding node and prints out the string and its length, or prints an error message.

```
#include <stdio.h>
#include <stdlib.h>

#define TABSIZE 1000

struct node {
    char *string;
    int length;
};

struct node table[TABSIZE]; /* table to be searched */
.
.
.
{
    struct node *node_ptr, node;
    int node_compare( ); /* routine to compare 2 nodes */
    char str_space[20]; /* space to read string into */
    .
    .
    .
    node.string = str_space;
    while (scanf("%s", node.string) != EOF) {
        node_ptr = (struct node *)bsearch((char *)(&node),
            (char *)table, TABSIZE,
            sizeof(struct node), node_compare);
        if (node_ptr != NULL) {
            (void)printf("string = %20s, length = %d\n",
                node_ptr->string, node_ptr->length);
        } else {
            (void)printf("not found: %s\n", node.string);
        }
    }
}

/*
   This routine compares two nodes based on an
   alphabetical ordering of the string field.
*/
int
node_compare(node1, node2)
char *node1, *node2;
{
    return (strcmp(
        ((struct node *)node1)->string,
        ((struct node *)node2)->string));
}
```

Diagnostics

A NULL pointer is returned if the key cannot be found in the table.

Notes

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Although `bsearch` is declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

See also

`hsearch(S)`, `lsearch(S)`, `qsort(S)`, `tsearch(S)`

Standards conformance

`bsearch` is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

ANSI X3.159-1989 Programming Language — C;

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

catgets

read a program message

Syntax

```
cc ... -lintl

#include <nl_types.h>

char *catgets (catd, set_id, msg_id, defmsg)
nl_catd catd;
int set_id, msg_id;
char *defmsg;
```

Description

catgets attempts to read message *msg_id*, in set *set_id*, from the message catalog identified by *catd*. *catd* is a catalog descriptor returned from an earlier call to **catopen(S)**. The *defmsg* argument points to a default message string which is returned by **catgets** if it cannot retrieve the identified message.

catgets fails if any of the following is true:

I_EBADC	The <i>set_id</i> and <i>msg_id</i> values are out of range.
I_EBADF	Invalid message catalog descriptor.
I_ENMEM	Could not allocate memory allocation while executing.
I_ENMSG	No message in the file for this combination of <i>set_id</i> and <i>msg_id</i> values; however, the values are not out of range.
I_EREAD	The message could not be read.

Return value

If the identified message is retrieved successfully, **catgets** returns a pointer to an internal buffer area containing the null terminated message string. If the call is unsuccessful because the message catalog identified by *catd* is not currently available, or if the message catalog is available, but the identified message is not contained therein, "s" (a pointer to the default message) is returned.

Application usage

The message-text is contained in an internal buffer area and should be copied by the application if it is to be saved or reused after further calls to **catgets**.

See also

catopen(S)

Standards conformance

catgets is conformant with:

X/Open Portability Guide, Issue 3, 1989.

(S)

catopen, catclose

open/close a message catalog

Syntax

```
cc ... -lintl

#include <nl_types.h>

nl_catd catopen (name, oflag)
char *name;
int oflag;

int catclose (catd)
nl_catd catd;
```

Description

catopen opens a message catalog and returns a catalog descriptor for use by **catgets(S)**. The *name* argument is the name of the message catalog to be opened. If *name* contains a "/" then *name* specifies a pathname for the message catalog. Otherwise, the environment variable NLSPATH is used with *name* substituted for %N (see **environ(M)**). If NLSPATH does not exist in the environment, or if a message catalog cannot be opened in any of the paths specified by NLSPATH, then */usr/lib/nls/msg/%l/%N.cat* is used. "%l" is replaced by the language component defined for \$LANG.

oflag is reserved for future use and should be set to 0 (zero). The results of setting this field to any other value are undefined.

Archives:

If the special syntax for name:

<programname>@<archivename>

is used, then only archives are opened by **catopen**. The environment variable NLSPATH is used with the *archivename* part of name substituted for %A (see **environ(M)**). Only parts of the path containing a %A are recognized. When the archive is opened successfully, the entry named *programname* is used as message catalog.

catclose closes the message catalog identified by the *catd* catalog descriptor.

The **catopen** routine fails if one of the following is true:

- [ENOMEM]** Insufficient storage space is available.
- I_EIBNV** The **LANG** variable is not defined or contains incorrect data.
- I_ENOOP** No message file was found according to **NLSPATH**.
- I_ENMEM** Out of memory.

The **catclose** routine fails if the following is true:

- I_EBADC** The message file could not be found or the message catalog file descriptor has already been closed.

Return value

If successful, **catopen** returns a message catalog descriptor for use on subsequent calls to **catgets** and **catclose**. If unsuccessful, **catopen** returns **(nl_catd) -1**.

catclose returns 0 if successful; otherwise it returns -1.

Notes

Using **catopen** may cause another file descriptor to be allocated by the calling process, in addition to *stdin*, *stdout* and *stderr*. Applications should take care not to close this file descriptor by mistake.

See also

catgets(S), **environ(M)**

Standards conformance

catopen and **catclose** are conformant with:
X/Open Portability Guide, Issue 3, 1989.

cfspeed: cfgetispeed, cfgetospeed, cfsetispeed, cfsetospeed

baud rate functions

Syntax

```
cc ... -lc

#include <termios.h>

speed_t cfgetospeed (termios_p)
struct termios *termios_p;

int cfsetospeed (termios_p, speed)
struct termios *termios_p;
speed_t speed;

speed_t cfgetispeed (termios_p)
struct termios *termios_p;

int cfsetispeed (termios_p, speed)
struct termios *termios_p;
speed_t speed;
```

Description

The following interfaces are provided for getting and setting the values of the input and output baud rates in the **termios** structure. The effects on the terminal device described below do not become effective until the **tcsetattr** function is called successfully.

The input and output baud rates are stored in the `termios` structure. The values shown in the following table are supported. The name symbols in this table are defined in `<termios.h>`.

Name	Description	Name	Description
BO	Hang up	B600	600 baud
B50	50 baud	B1200	1200 baud
B75	75 baud	B1800	1800 baud
B110	110 baud	B2400	2400 baud
B134	134.5 baud	B4800	4800 baud
B150	150 baud	B9600	9600 baud
B200	200 baud	B19200	19200 baud
B300	300 baud	B38400	38400 baud

The type `speed_t` is defined in `<termios.h>` and as an unsigned integral type.

The `termios_p` argument is a pointer to a `termios` structure. `cfgetospeed` returns the output baud rate stored in the `termios` structure pointed to by `termios_p`.

`cfsetospeed` sets the output baud rate stored in the `termios` structure pointed to by `termios_p` to `speed`. The zero baud rate, BO, is used to terminate the connection. If BO is specified, the modem control lines are no longer asserted. Normally, this disconnects the line.

`cfgetispeed` returns the input baud rate stored in the `termios` structure.

`cfsetispeed` sets the input baud rate stored in the `termios` structure to `speed`. If the input baud rate is set to zero, the input baud rate is specified by the value of the output baud rate. Both `cfsetispeed` and `cfsetospeed` return a value of zero if successful and -1 to indicate an error. Attempts to set unsupported baud rates are ignored, and it is implementation-defined whether an error is returned by any or all of `cfsetispeed`, `cfsetospeed`, or `tcsetattr`. This refers both to changes to baud rates not supported by the hardware, and to changes setting the input and output baud rates to different values if the hardware does not support this.

Standards conformance

`cfgetispeed`, `cfgetospeed`, `cfsetispeed` and `cfsetospeed` are conformant with: IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1); and NIST FIPS 151-1.

chdir

change working directory

Syntax

```
cc ... -lc

int chdir (path)
char *path;
```

Description

path points to the pathname of a directory. **chdir** causes the named directory to become the current working directory, the starting point for path searches for pathnames not beginning with *.*.

chdir fails and the current working directory is unchanged if one or more of the following is true:

[EACCES]	Search permission is denied for any component of the path-name.
[EFAULT]	<i>path</i> points outside the allocated address space of the process.
[EINTR]	A signal was caught during the chdir system call.
[EMULTIHOP]	Components of <i>path</i> require hopping to multiple remote machines.
[ENOENT]	The named directory does not exist.
[ENOLINK]	<i>path</i> points to a remote machine and the link to that machine is no longer active.
[ENOTDIR]	A component of the pathname is not a directory.

See also

chroot(S)

Diagnostics

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

Standards conformance

chdir is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

chmod

change mode of file

Syntax

```
cc ... -lc

int chmod (path, mode)
char *path;
int mode;
```

Description

The *path* argument points to a path name naming a file. The `chmod` system call sets the access permission portion of the named file's mode according to the bit pattern contained in *mode*.

Access permission bits are interpreted as follows:

04000	Set user ID on execution.
020#0	Set group ID on execution if # is "7", "5", "3", or "1" Enable mandatory file/record locking if # is "6", "4", "2", or "0"
01000	Save text image after execution.
00400	Read by owner.
00200	Write by owner.
00100	Execute (search if a directory) by owner.
00070	Read, write, execute (search) by group.
00007	Read, write, execute (search) by others.

The effective user ID of the process must match the owner of the file or be super-user to change the mode of a file.

If the effective user ID of the process is not super-user and the file is not a directory, mode bit 01000 (save text image on execution) is cleared.

If the effective user ID of the process is not super-user and the effective group ID of the process does not match the group ID of the file, mode bit 02000 (set group ID on execution) is cleared.

If a 410 executable file has the sticky bit (mode bit 01000) set, the operating system will not delete the program text from the swap area when the last user process terminates. If a 413 executable file has the sticky bit set, the operating system will not delete the program text from memory when the last user process terminates. In either case, if the sticky bit is set, the text will already be available (either in a swap area or in memory) when the next user of the file executes it, thus making execution faster.

Overall, if a directory is writable and has the sticky bit set, files within that directory can only be removed if one or more of the following is true (see `unlink(S)`):

- the user owns the file
- the user owns the directory
- the file is writable to the user
- the user is the super-user

If the mode bit 02000 (set group ID on execution) is set and the mode bit 00010 (execute or search by group) is not set, mandatory file/record locking will exist on a regular file. This may effect future calls to `open(S)`, `creat(S)`, `read(S)`, and `write(S)` on this file.

`chmod` will fail and the file mode will be unchanged if one or more of the following is true:

- [ENOTDIR] A component of the path prefix is not a directory.
- [ENOENT] The named file does not exist.
- [EACCES] Search permission is denied on a component of the path prefix.
- [EPERM] The effective user ID does not match the owner of the file and the effective user ID is not super-user.
- [EROFS] The named file resides on a read-only file system.
- [EFAULT] *path* points outside the allocated address space of the process.
- [EINTR] A signal was caught during the `chmod` system call.
- [ENOLINK] *path* points to a remote machine and the link to that machine is no longer active.
- [EMULTIHOP] Components of *path* require hopping to multiple remote machines.

Diagnostics

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

See also

`chown(S)`, `chmod(C)`, `creat(S)`, `fcntl(S)`, `mknod(S)`, `open(S)`, `read(S)`, `write(S)`

Standards conformance

chmod conforms with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

chown

change owner and group of a file

Syntax

```
cc ... -lc

#include <sys/types.h>

int chown (path, owner, group)
char *path;
uid_t owner;
gid_t group;
```

Description

path points to a pathname naming a file. The owner ID and group ID of the named file are set to the numeric values contained in *owner* and *group* respectively.

If the process has **chown** kernel authorization, only processes with effective user ID equal to the file owner or super user may change the ownership of a file. Otherwise, only the super user may change the ownership of a file. **chown** kernel authorization is set by a system administrator using the **sysadmsh(ADM)** utility.

If **chown** is invoked by other than the super user, the set-user-ID and set-group-ID bits of the file mode, 04000 and 02000 respectively, will be cleared.

chown fails and the owner and group of the named file remains unchanged if one or more of the following is true:

[EACCES]	Search permission is denied on a component of the path prefix.
[EFAULT]	<i>path</i> points outside the allocated address space of the process.
[EINTR]	A signal was caught during the chown system call.
[EMULTIHOP]	Components of <i>path</i> require hopping to multiple remote machines.
[ENOENT]	The named file does not exist.
[ENOLINK]	<i>path</i> points to a remote machine and the link to that machine is no longer active.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not super user.

[EROFS] The named file resides on a read-only file system.

Diagnostics

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

See also

chmod(S), chown(C)

Standards conformance

chown is conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

chroot

change root directory

Syntax

```
cc ... -lc

int chroot (path)
char *path;
```

Description

The *path* argument points to a pathname naming a directory. The **chroot** system call causes the named directory to become the root directory, the starting point for path searches for pathnames beginning with */*. The user's working directory is unaffected by the **chroot** system call.

The effective user ID of the process must be super user to change the root directory.

The *..* entry in the root directory is interpreted to mean the root directory itself. Thus, *..* cannot be used to access files outside the subtree rooted at the root directory.

The **chroot** system call fails and the root directory remains unchanged if one or more of the following is true:

[EFAULT]	The <i>path</i> argument points outside the allocated address space of the process.
[EINTR]	A signal was caught during the chroot system call.
[EMULTIHOP]	Components of <i>path</i> require hopping to multiple remote machines.
[ENOENT]	The named directory does not exist.
[ENOLINK]	The <i>path</i> argument points to a remote machine and the link to that machine is no longer active.
[ENOTDIR]	Any component of the pathname is not a directory.
[EPERM]	The effective user ID is not super user.

Diagnostics

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned, and *errno* is set to indicate the error.

See also

chdir(S)

Standards conformance

chroot is conformant with:

AT&T SVID Issue 2;
and X/Open Portability Guide, Issue 3, 1989.

chsize

changes the size of a file

Syntax

```
cc ... -lx

int chsize (fildes, size)
int fildes;
long size;
```

Description

fildes is a file descriptor obtained from a **creat**, **open**, **dup**, **fcntl**, or **pipe** system call. **chsize** changes the size of the file associated with the file descriptor *fildes* to be exactly *size* bytes in length. The routine either truncates the file, or pads it with an appropriate number of bytes. If *size* is less than the initial size of the file, then all allocated disk blocks between *size* and the initial file size are freed.

The maximum file size as set by **ulimit(S)** is enforced when **chsize** is called, rather than on subsequent writes. Thus **chsize** fails, and the file size remains unchanged if the new changed file size would exceed the **ulimit**.

Return value

Upon successful completion, a value of 0 is returned. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

Notes

In general if **chsize** is used to expand the size of a file, when data is written to the end of the file, intervening blocks are filled with zeros. In a few rare cases, reducing the file size may not remove the data beyond the new end-of-file. This routine must be linked with the linker option **-lx**.

See also

creat(S), **dup(S)**, **lseek(S)**, **open(S)**, **pipe(S)**, **ulimit(S)**

Standards conformance

chsize is an extension of AT&T System V provided by the Santa Cruz Operation.

clock

report CPU time used

Syntax

```
cc ... -lc  
long clock ()
```

Description

The **clock** function returns the amount of CPU time (in microseconds) used since the first call to **clock**. The time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed **wait(S)**, **pclose(S)**, or **system(S)**.

The resolution of the clock is 10 milliseconds.

Notes

The value returned by **clock** is defined in microseconds for compatibility with systems that have CPU clocks with much higher resolution. Because of this, the value returned wraps around after accumulating only 2147 seconds of CPU time (about 36 minutes).

The frequency of the system clock in ticks per second can be set via the environment variable **HZ**. **clock** uses **gethz(S)**, which in turn uses **HZ**, in calculating the return value. If this fails, the default value, set in **<sys/param.h>**, is used instead and a warning message is printed to **stderr**.

See also

popen(S), **system(S)**, **times(S)**, **wait(S)**

Standards conformance

clock is conformant with:

AT&T SVID Issue 2;
and X/Open Portability Guide, Issue 3, 1989.

close

close a file descriptor

Syntax

```
cc ... -lc

int close (fildes)
int fildes;
```

Description

The `close` function breaks the connection between a file descriptor and an open file, and frees the file descriptor for use with some other file.

The *fildes* argument is a file descriptor obtained from a `creat`, `open`, `dup`, `fcntl`, or `pipe` system call. The `close` system call closes the file descriptor indicated by *fildes*. All outstanding record locks owned by the process (on the file indicated by *fildes*) are removed.

If a STREAMS file is closed, and the calling process had previously registered to receive a SIGPOLL signal (see `signal(S)` and `sigset(S)`) for events associated with that file, the calling process is unregistered for events associated with the file. The last `close` for a stream causes the stream associated with *fildes* to be dismantled. If `O_NONBLOCK` is not set and there have been no signals posted for the stream, `close` waits up to 15 seconds, for each module and driver, for any output to drain before dismantling the stream. If the `O_NONBLOCK` flag is set or if there are any pending signals, `close` does not wait for output to drain and dismantles the stream immediately.

The named file is closed unless one or more of the following is true:

- [EBADF] The *fildes* argument is not a valid open file descriptor.
- [EINTR] A signal was caught during the `close` system call.
- [ENOLINK] *fildes* is on a remote machine and the link to that machine is no longer active.

Diagnostics

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned, and `errno` is set to indicate the error.

Example

The function is typically used to close files that are no longer needed. For example, the following program fragment closes the standard input if the argument count is greater than 1.

```
int fd;

if (argc >1)
    close(0);
```

Note that all open files in a program are closed when a program terminates normally or when the `exit` function is called, so no explicit call to `close` is required.

See also

`creat(S)`, `dup(S)`, `exec(S)`, `fcntl(S)`, `Intro(S)`, `open(S)`, `pipe(S)`, `signal(S)`, `sigset(S)`

Standards conformance

`close` is conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

creat

create a new file or rewrite an existing one

Syntax

```
cc ... -lc

#include <sys/types.h>
#include <sys/stat.h>

int creat (path, mode)
char *path;
int mode;
```

Description

The **creat** system call creates a new ordinary file or prepares to rewrite an existing file named by the path name pointed to by *path*.

If the file exists, the length is truncated to 0 and the mode and owner are unchanged. Otherwise, the file's owner ID is set to the effective user ID of the process; the group ID of the process is set to the effective group ID of the process; and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows:

- All bits set in the process's file mode creation mask are cleared (see **umask(S)**).
- The "save text image after execution bit" of the mode is cleared (see **chmod(S)**).

Upon successful completion, a write-only file descriptor is returned and the file is open for writing, even if the mode does not permit writing. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across **exec** system calls (see **fcntl(S)**). No process may have more than **FOPEN_MAX** files open simultaneously. A new file may be created with a mode that forbids writing.

Symbolic constants defining the access permission bits are specified in the *<sys/stat.h>* header file and should be used to construct *mode* (see **chmod(S)**).

The call **creat(path, mode)** is equivalent to the following (see **open(S)**):

```
open(path, O_WRONLY | O_CREAT | O_TRUNC, mode)
```

The **creat** system call fails if one or more of the following is true:

[EACCESS]	1. Search permission is denied on a component of the path prefix. 2. The file does not exist and the directory in which the file is to be created does not permit writing. 3. The file exists and write permission is denied.
[EAGAIN]	The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file (see chmod(S)).
[EFAULT]	The <i>path</i> argument points outside the allocated address space of the process.
[EINTR]	A signal was caught during the creat system call.
[EISDIR]	The named file is an existing directory.
[EMFILE]	NOFiles file descriptors are currently open.
[EMULTIHOP]	Components of <i>path</i> require hopping to multiple remote machines.
[ENFILE]	The system file table is full.
[ENOENT]	1. A component of the path prefix does not exist. 2. The pathname is null.
[ENOLINK]	<i>path</i> points to a remote machine and the link to that machine is no longer active.
[ENOSPC]	The filesystem is out of inodes.
[ENOTDIR]	A component of the path prefix is not a directory.
[EROFS]	The named file resides or would reside on a read-only file system.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed.

Diagnostics

Upon successful completion, a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

See also

chmod(S), **close(S)**, **dup(S)**, **fcntl(S)**, **lseek(S)**, **open(S)**, **read(S)**, **umask(S)**, **write(S)**

Standards conformance

creat is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

(S)

create_file_securely

create a file using an attribute specification from the file control database

Syntax

```
cc ... -lprot -lx

#include <sys/types.h>
#include <prot.h>

int create_file_securely (path, mode, purpose)
char *path;
int mode;
char *purpose;
```

Description

create_file_securely(S) is used to create a file and to assign to that file attributes specified for it in the file control database, */etc/auth/system/files*. Attributes that may be specified for a file control database entry that are used by this routine include:

f_owner Owner of the file
f_group Group of the file
f_mode File permission bits

Other attributes supported by the file control database have no effect during the creation of a file but are intended for use by programs such as **integrity(ADM)**.

The *path* argument is the name of a file to be created using attributes from the file control database. The file name should match exactly a file name entry in the file control database.

mode is either **AUTH_SILENT** or **AUTH_VERBOSE**. **AUTH_VERBOSE** mode results in signals being blocked until the routine is finished and also will result in the generation of audit records for failure conditions. Additionally, since the routine is used to create temporary files on behalf of callers, a temporary file may already exist at the time the routine is invoked. The use of **AUTH_VERBOSE** mode allows the routine to wait for any existing temporary file to be removed, using an adaptive time-out/retry mechanism. This allows synchronization of simultaneous database updates. If the mode is specified as **AUTH_SILENT**, signals are not blocked by the routine nor are audit records generated when an operation fails. Additionally, if the entry refers to a temporary file and that file already exists, no attempt is made to wait until it is removed.

The *purpose* argument is a pointer to a string identifying the reason for using the routine to create the file. This string is used as part of audit record when the mode is AUTH_VERBOSE.

The routine sets the attributes of the file according to a specific sequence designed to minimize the need for the calling process to have certain effective privileges. The file is first created using the O_EXCL option and the open(S) system call. This will fail if the file already exists. Once the file has been created, it is closed since certain attributes can only be set on the file if the file is not in use. If the file control database entry does not specify an owner, the owner of the file will be set to the real user id of the process. Likewise, if the entry does not specify the file group, it is set to the real group id of the process. If a value for *mode* is not specified for the file, then *mode* will be set to its default value, 0.

The file permissions are set according to the file control database entry, if a value is present, or to a mode of 0. This is done using the chmod(S) system call while the file is still owned by the calling user.

Lastly, the owner and group of the file are set using the chown(S) system call.

Diagnostics

create_file_securely(S) returns a status code that provides detailed information on the result of the create operation. The return codes are:

CFS_GOOD_RETURN

The file was created successfully and all of the attributes specified in the file control database entry were successfully set on the new file.

CFS_CAN_NOT_OPEN_FILE

The routine was unable to create the file identified by *path*.

CFS_NO_FILE_CONTROL_ENTRY

There was not a file control database entry for *path*.

CFS_CAN_NOT_CHG_MODE

The permissions specified by the file control database could not be set on the created file.

CFS_CAN_NOT_CHG_OWNER_GROUP

The owner/group specified in the file control database entry could not be set for the created file.

If the file can not be created or an attempt to set any of the attributes specified by the file control database entry fails, the routine will return one of the above failure codes. If the failure resulted from the inability to set an attribute on the created file, the file is also removed before returning.

Note

Programs using this routine must be compiled with **-lprot** or **-lprot_s**, together with **-lx**.

See also

getprfient(S), **files(F)**.

Standards conformance

create_file_securely(S) is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

creatsem

creates an instance of a binary semaphore

Syntax

```
cc ... -lx

int creatsem (sem_name, mode)
char *sem_name;
int mode;
```

Description

creatsem defines a binary semaphore named by *sem_name* to be used by **waitsem(S)** and **sigsem(S)** to manage mutually exclusive access to a resource, shared variable, or critical section of a program. **creatsem** returns a unique semaphore number, *sem_num*, which may then be used as the parameter in **waitsem** and **sigsem** calls. Semaphores are special files of 0 length. The filename space is used to provide unique identifiers for semaphores. *mode* sets the accessibility of the semaphore using the same format as file access bits. Access to a semaphore is granted only on the basis of the read access bit; the write and execute bits are ignored.

A semaphore can be operated on only by a synchronizing primitive, such as **waitsem** or **sigsem**, by **creatsem** which initializes it to some value, or by **opensem** which opens the semaphore for use by a process. Synchronizing primitives are guaranteed to be executed without interruption once started. These primitives are used by associating a semaphore with each resource (including critical code sections) to be protected.

The process controlling the semaphore should issue:

```
sem_num = creatsem(semaphore, mode);
```

to create, initialize, and open the semaphore for that process. All other processes using the semaphore should issue:

```
sem_num = opensem(semaphore);
```

to access the semaphore's identification value. Note that a process cannot open and use a semaphore that has not been initialized by a call to **creatsem**, nor should a process open a semaphore more than once in one period of execution. Both the creating and opening processes use **waitsem** and **sigsem** to use the semaphore *sem_num*.

Diagnostics

creatsem returns the value -1 if an error occurs. **errno** is set under these conditions:

creatsem(S)

- [EEXIST] The semaphore named by **sem_name** is already open for use by other processes.
- [ENAVAIL] The semaphore has not been initialized by a call to **creatsem**.
- [ENOTNAM] The file specified exists but is not a semaphore type.

Notes

After a calling **creatsem**, use **waitsem** to gain control of a given resource.

See also

opensem(S), **sigsem(S)**, **waitsem(S)**

Compatibility

creatsem can only be used to define UNIX version 3.0 type semaphores, not UNIX System V type semaphores.

This feature is a XENIX specific enhancement and may not be present in all UNIX implementations.

Standards conformance

creatsem is an extension of AT&T System V provided by the Santa Cruz Operation.

crypt

password and file encryption functions

Syntax

```
cc -x2.3 ... -lcrypt      (OMF binaries)
cc .../usr/lib/libcrypt_d.a (domestic, COFF binaries)
cc .../usr/lib/libcrypt_i.a (international, COFF binaries)

void setkey (key)
char *key;

char *crypt (key, salt)
char *key, *salt;

void encrypt (block, flag)
char *block;
int flag;

char *des_crypt (key, salt)
char *key, *salt;

void des_encrypt (block, flag)
char *block;
int flag;

void des_setkey (key)
char *key;

int cryptopen (p)
int p[2];

int run_setkey (p, key)
int p[2];
char *key;

int run_crypt (offset, buffer, count, p)
long offset;
char *buffer;
unsigned int count;
int p[2];

int crypt_close(p)
int p[2];
```

(S)

Description

crypt - encrypts a password

encrypt - encrypts a password

setkey - creates encryption key

des_crypt - encrypts password from DES primitive

des_encrypt - encrypts or decrypts data byte with DES primitive

des_setkey - creates encryption key with DES primitive

cryptopen - open */bin/crypt* connection

run_setkey - creates encryption key for */bin/crypt* connection

run_crypt - encrypts data using */bin/crypt* connection

crypt_close - terminates */bin/crypt* connection

des_crypt is the password encryption function. It is based on a one-way hashing encryption algorithm with variations intended (among other things) to frustrate use of hardware implementations of a key search.

key is a user's typed password. *salt* is a two-character string chosen from the set [*a-z, A-Z, 0-9*]. This string is used to perturb the hashing algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password. The first two characters are the salt itself.

The **des_setkey** and **des_encrypt** entries provide (rather primitive) access to the actual hashing algorithm. The argument of **des_setkey** is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored; this gives a 56-bit key which is set into the machine. This is the key that is used with the hashing algorithm to encrypt the string *block* with the function **des_encrypt**.

The argument to the **des_encrypt** entry is a character array of length 64 containing only the characters with numerical value 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the hashing algorithm using the key set by **des_setkey**. If *edflag* is zero, the argument is encrypted; if non-zero, it is decrypted.

Note that decryption is not provided in the international version of **crypt(S)**. If decryption is attempted with the international version of **des_encrypt**, an error message is printed.

crypt, **setkey**, and **encrypt** are front-end routines that invoke **des_crypt**, **des_setkey**, and **des_encrypt** respectively.

The routines **run_setkey** and **run_crypt** are designed for use by applications that need cryptographic capabilities (such as **ed(C)** and **vi(C)**) that must be compatible with the **crypt(C)** user-level utility. **run_setkey** establishes a two-way pipe connection with **crypt(C)**, using **key** as the password argument.

cryptopen returns a pair of file descriptors for input and output streams connections to the **/bin/crypt** encryption utility. **run_crypt** takes a block of characters and transforms the cleartext or ciphertext using **crypt(C)**. **offset** is the relative byte position from the beginning of the file that the block of text provided in **buffer** is coming from. **count** is the number of characters in **buffer**, and **p** is an array containing file descriptor indices to a table of input and output file streams. When encryption is finished, **crypt_close** is used to terminate the connection with **crypt(C)**.

run_setkey returns -1 if a connection with **crypt(C)** cannot be established. This occurs on international versions of the Operating System where **crypt(C)** is not available. If a null key is passed to **run_setkey**, 0 is returned. Otherwise, 1 is returned. **cryptopen** returns 1 if successful and -1 on an error. **run_crypt** returns -1 if it cannot write output or read input from the pipe attached to **crypt**. Otherwise it returns 0.

Diagnostics

In the international version of **crypt(S)**, a flag argument of 1 to **des_encrypt** is not accepted, and an error message is printed.

Note

The return value in **crypt** points to static data that is overwritten by each call.

See also

crypt(C), **getpass(S)**, **login(M)**, **passwd(C)**, **passwd(FP)**

Standards conformance

crypt is conformant with:

AT&T SVID Issue 2;
and X/Open Portability Guide, Issue 3, 1989.

ctermid

generate terminal filename

Syntax

```
cc ... -lc

#include <stdio.h>

char *ctermid (s)
char *s;
```

Description

The **ctermid** function generates the path name of the controlling terminal for the current process and stores it in a string.

If *s* is a NULL pointer, the string is stored in an internal static area, the contents of which are overwritten at the next call to **ctermid**, and the address of which is returned. Otherwise, *s* is assumed to point to a character array of at least **L_ctermid** elements; the path name is placed in this array, and the value of *s* is returned. The constant **L_ctermid** is defined in the *<stdio.h>* header file.

Notes

The difference between **ctermid** and **ttyname(S)** is that **ttyname** must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while **ctermid** returns a string (*/dev/tty*) that refers to the terminal if used as a file name. Thus **ttyname** is useful only if the process already has at least one file open to a terminal.

See also

ttyname(S)

Standards conformance

ctermid is conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

ctime, localtime, gmtime, asctime, strftime, tzset

convert date and time to string

Syntax

```
cc ... -lc

#include <sys/types.h>
#include <time.h>

extern long altzone;

char *asctime (tm)
struct tm *tm;

char *ctime (clock)
time_t *clock;

extern int daylight;

struct tm *gmtime (timer)
time_t *timer;

struct tm *localtime (clock)
time_t *clock;

extern long timezone;

extern char *tzname[2];

void tzset ()

cc ... -lintl

size_t strftime (s, maxsize, format, timeptr)
char *s;
size_t maxsize;
char *format;
struct tm *timeptr;
```

(S)

Description

In many of the functions described on this manual page, “epoch” is used to describe the moment at which time in the UNIX operating system started. The exact starting point is 00:00:00 Greenwich Mean Time (GMT) on January 1, 1970. This point is arbitrary and does not have any special significance.

ctime converts the time pointed to by *clock* representing seconds since the epoch to local time in the form of a string. **ctime** is equivalent to **asctime(localtime(clock))**. The **time_t** type is defined in the *sys/types.h* header file.

localtime converts the time in seconds pointed to by *clock* into a **tm** structure. The time in *clock* is the number of seconds since the epoch. **localtime** corrects for the timezone and seasonal time adjustments. The output is the same as that generated by **tzset**. The **time_t** type is defined in the *sys/types.h* header file.

The return value is a pointer to a 26-character string in the following form. All the fields have constant width.

```
Fri Sep 13 00:00:00 1986
```

localtime and **gmtime** return pointers to **tm** structures, described below. **localtime** corrects for the main time zone and possible alternate (Daylight Savings) time zone; **gmtime** converts directly to Coordinated Universal Time (UTC), which is the same as Greenwich Mean Time (GMT).

asctime converts a **tm** structure to a 26-character string, and returns a pointer to the string.

The **asctime** routine uses the equivalent of the following algorithm to create the date string:

```
char *asctime(timeptr)
struct tm *timeptr;
{
    static char wday_name[7][3] = {
        "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
    };
    static char mon_name[12][3] = {
        "Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
    };
    static char result[26];

    sprintf(result, "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",
        wday_name[timeptr->tm_wday],
        mon_name[timeptr->tm_mon],
        timeptr->tm_min, timeptr->tm_sec,
        1900 + timeptr->tm_year);
    return result;
}
```

Declarations of all the functions and externals, and the `tm` structure, are in the `<time.h>` header file. The structure declaration is:

```
struct tm {
    int    tm_sec;    /* seconds after the minute - [0, 59] */
    int    tm_min;    /* minutes after the hour - [0, 59] */
    int    tm_hour;   /* hour since midnight - [0, 23] */
    int    tm_mday;   /* day of the month - [1, 31] */
    int    tm_mon;    /* months since January - [0, 11] */
    int    tm_year;   /* years since 1900 */
    int    tm_wday;   /* days since Sunday - [0, 6] */
    int    tm_yday;   /* days since January 1 - [0, 365] */
    int    tm_isdst;  /* flag for daylight savings time */
    long   tm_tzadj;  /* seconds from GMT (east < 0) */
    char   tm_name[LTZNMAX]; /* name of timezone */
};
```

`tm_isdst` is non-zero if the alternate time zone is in effect.

Set the `LANG` environment variable to specify the language that `strftime` uses for output. If `LANG` is empty, unset, or set to an unsupported language, the last language requested is used (the default is the *usa-english* strings).

The `strftime` function places its output (according to the format string *format* and the time values contained in the structure pointed to by *timeptr*), followed by the null character (`\0`) in consecutive bytes starting at *s*. The maximum length of the output string is specified by *maxsize*, the maximum number of characters including the terminating null character.

The `strftime` function converts and formats the time values contained in the structure pointed to by *timeptr* under control of the format string *format*. The string *format* is a character string that can contain two types of objects. These are plain characters, which are simply copied to the output string, and directives.

Each directive is introduced by the percent character (%). The `%d`, `%H`, `%I`, `%j`, `%m`, `%M`, `%S`, `%U`, `%w`, `%W`, `%y`, `%Z`, and `%%` directives are independent of the program's current locale (see `locale(M)`). The strings used to replace the `%a`, `%A`, `%b`, `%B`, `%c`, `%p`, `%x`, and `%X` directives are dependent on the program's current locale, and are defined using the utility `timtbl(M)`.

The directives are:

<code>%a</code>	is replaced by the locale's abbreviated weekday name
<code>%A</code>	is replaced by the locale's full weekday name
<code>%b</code>	is replaced by the locale's abbreviated month name
<code>%B</code>	is replaced by the locale's full month name
<code>%c</code>	is replaced by the locale's appropriate date and time representation
<code>%d</code>	is replaced by the day of the month as a decimal number (01-31)

%D	is replaced by the date (%m/%d/%y).
%h	is replaced by the locale's abbreviated month name
%H	is replaced by the hour (24-hour clock) as a decimal number (00-23)
%I	is replaced by the hour (12-hour clock) as a decimal number (01-12)
%j	is replaced by the day of the year as a decimal number (001-366)
%m	is replaced by the month as a decimal number (01-12)
%M	is replaced by the minute as a decimal number (00-59)
%n	is replaced by a newline character.
%p	is replaced by the locale's equivalent of AM or PM.
%r	is replaced by the time in AM or PM notation according to the British or US conventions (%I:%M:%S\ [AM PM]).
%S	is replaced by the second as a decimal number (00-61)
%t	is replaced by a tab character.
%T	is replaced by the time (%H:%M:%S).
%U	is replaced by the week number of the year (Sunday as the first day of the week) as a decimal number (00-53).
%w	is replaced by the weekday as a decimal number [0(Sunday)-6].
%W	is replaced by the week number of the year (Monday as the first day of the week) as a decimal number (00-53).
%x	is replaced by the locale's appropriate date representation.
%X	is replaced by the locale's appropriate time representation.
%y	is replaced by the year without a century as a decimal number (00-99).
%Y	is replaced by the year with a century as a decimal number.
%Z	is replaced by the timezone name, or by no characters if no timezone exists.
%%	is replaced by %

The external long variable **timezone** contains the difference, in seconds, between GMT and the main time zone; the external long variable **altzone** contains the difference, in seconds, between GMT and the alternate time zone; both, **timezone** and **altzone** default to 0 (GMT). The external variable **daylight** is non-zero if an alternate time zone exists. The time zone names are contained in the external variable **tzname**, which by default is set to

```
char *tzname[2] = { "GMT", " " };
```

The functions handle the peculiarities of this conversion for various time periods for the USA (specifically, the years 1974, 1975, 1976, and 1987). The functions handle the new daylight savings time starting with the first Sunday in April, 1987, 1989.

tzset uses the contents of the environment variable **TZ** to override the value of the different external variables. See *timezone(F)* or *environ(M)* for the syntax of **TZ**.

LTZNMAX specifies the maximum number bytes in the timezone name.

tzset uses the value of the environment variable **TZ** to set time conversion information used by *localtime()*, *ctime()*, *strftime()* and *mktime()*. If **TZ** is absent from the environment, implementation-defined default time zone information is used.

For example, the setting for New Jersey in 1986 could be

```
"EST5EDT4,117/2:00:00,299/2:00:00" .
```

or simply

```
EST5EDT
```

A southern hemisphere setting could be

```
"KDT9:30KST10:00,303/20:00,64/5:00"
```

When the longer format is used, the variable must be surrounded by double quotes as shown. For more details, see *timezone(F)* and *environ(M)*. In the longer version of the New Jersey example of **TZ**, **tzname[0]** is **EST**, **timezone** is set to 5*60*60, **tzname[1]** is **EDT**, **altzone** is set to 4*60*60, the starting date of the alternate time zone is the 117th day at 2 AM, the ending date of the alternate time zone is the 299th day at 2 AM, and **daylight** is set to non-zero. Starting and ending times are relative to the current time. If the alternate time zone start and end dates and the time are not provided, the days for the United States that year are used and the time is 2 AM. The effects of **tzset** are thus to change the values of the external variables **timezone**, **altzone**, **daylight**, and **tzname**. **tzset** is called by *localtime* and may also be called explicitly by the user.

Note that in most installations, **TZ** is set to the correct value by default when the user logs on, via the local */etc/profile* file (see *profile(M)*).

Return value

The **strftime** function returns the number of characters placed in the string *s* (not including the terminating null character), or zero if the length of the output string would exceed *maxsize*. In the latter case, the content of the output string *s* is undefined.

Zero is also returned by **strftime** if the function detects a situation which would lead to infinite recursion; for example, if a format string refers to itself.

Note

The return values for **ctime**, **localtime** and **gmtime** point to static data whose content is overwritten by each call.

Setting the time during the interval of change from **timezone** to **altzone** or vice versa can produce unpredictable results.

The system administrator must change the Julian start and end days annually if the full form of the TZ variable is specified.

See also

difftime(S), **environ(M)**, **getenv(S)**, **mktime(S)**, **nl_cxtime(S)**, **printf(S)**, **profile(M)**, **putenv(S)**, **stime(S)**, **strftime(S)**, **time(S)**, **timezone(F)**, **timtbl(M)**

Standards conformance

asctime, **ctime**, **gmtime** and **localtime** are conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
ANSI X3.159-1989 Programming Language — C;
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

strftime is conformant with:

ANSI X3.159-1989 Programming Language — C;
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
X/Open Portability Guide, Issue 3, 1989;
and NIST FIPS 151-1.

tzset is conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

ctype

character handling routines

Syntax

```
#include <ctype.h>
```

```
int isdigit(c);  
int c;
```

```
...
```

```
tolower(c)  
int c;
```

Description

The character classification macros listed below are defined by the `<ctype.h>` header file. They each return nonzero for true, zero for false.

These functions classify character-coded integer values according to the rules of the coded character set defined by character type information in the program's locale (category `LC_TYPE`).

In the C locale, or in a locale where character type information is not defined, characters are classified according to the rules of the US-ASCII 7-bit coded character set, returning zero for values above octal 0177.

In all cases the argument is an integer, the value of which must be representable as an **unsigned char** or must equal the value of the macro `EOF`. If the argument has any other value, the behaviour is undefined. `isascii` is defined on all integer values.

isdigit	tests for the digits 0 through 9.
isxdigit	tests for any character for which isdigit is true or for the letters <i>a</i> through <i>f</i> or <i>A</i> through <i>F</i> .
islower	tests for any lowercase letter as defined by the character set.
isupper	tests for any uppercase letter as defined by the character set.
isalpha	tests for any character for which islower or isupper is true and possibly any others as defined by the character set.
isalnum	tests for any character for which isalpha or isdigit is true.
isspace	tests for a space, horizontal-tab, carriage return, newline, vertical-tab, or form-feed.

iscntrl	tests for “control characters” as defined by the character set.
ispunct	tests for any character other than the ones for which isalnum , iscntrl , or isspace is true.
isprint	tests for a space or any character for which isalnum or ispunct is true or other “printing character” as defined by the character set.
isgraph	tests for any character for which isprint is true, except for space.
isascii	tests for an ASCII character (a non-negative number less than 0200).

The conversion functions and macros translate a character from lowercase (uppercase) to uppercase (lowercase).

tolower	if the character is one for which isupper is true and there is a corresponding lowercase character, tolower returns the corresponding lowercase character. Otherwise, the character is returned unchanged.
toupper	if the character is one for which islower is true and there is a corresponding uppercase character, otherwise, the character is returned unchanged.
toascii	turns off the bits that are not part of the ASCII character set.
_tolower	returns the lowercase representation of a character for which isupper is true, otherwise undefined.
_toupper	returns the uppercase representation of a character for which islower is true, otherwise undefined.

The conversion macros have the same functionality of the functions on valid input, but the macros are faster because they do not do range checking.

All the character classification macros and the conversion functions and macros do a table lookup.

See also

ascii(M), **chrtbl(M)**, **environ(M)**, **stdio(S)**

Diagnostics

If the argument to any of the character handling macros is not in the domain of the function, the result is undefined.

Warning

If a character variable or constant is passed to these functions or macros, undefined results may occur on machines which sign-extend characters by default.

Standards conformance

isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, tolower and toupper are conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

ANSI X3.159-1989 Programming Language — C;

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

_tolower and _toupper are conformant with:

AT&T SVID Issue 2;

and X/Open Portability Guide, Issue 3, 1989.

(S)

curses

terminal screen handling and optimization package

Synopsis

The **curses** manual page is organized as follows:

In Syntax

- compiling information
- summary of parameters used by **curses** routines
- alphabetical list of **curses** routines and their parameters

In Description:

- An overview of how **curses** routines should be used

In Routines, each **curses** routine is described under the appropriate heading:

- Overall Screen Manipulation
- Window and Pad Manipulation
- Output
- Input
- Output Options Setting
- Input Options Setting
- Environment Queries
- Color Manipulation
- Soft Labels
- Low-level Curses Access
- Terminfo-Level Manipulations
- Termcap Emulation
- Miscellaneous
- Use of *curscr*
- Obsolete calls

Then come sections on:

- ATTRIBUTES
- COLORS
- FUNCTION KEYS
- LINE GRAPHICS

Syntax

cc [flag ...] file ... -lcurses [library ...]

```
#include <curses.h>      /* automatically includes <stdio.h>, */  
                        /* <termio.h>, and <unctrl.h> */
```

There are two different **curses** libraries for creating both COFF and XENIX binaries: "termcap" **curses** and "terminfo" **curses**. Each version of **curses** uses a different terminal capabilities database, has different functionality, and has a corresponding header file.

For the most part, **termcap curses** is a subset of **terminfo curses**. The following documentation serves for both libraries. Routines followed by the comment "/* terminfo only */" are not available services under **termcap curses**. Likewise those followed by the comment "/* termcap only */" are not available in **terminfo curses**. Routines followed by the comment "/* macro */" are defined macros which have no return value.

When compiling UNIX/386 binaries (COFF), the **-lcurses** library is linked to the **terminfo** library */usr/lib/libcurses.a*, which, in turn, is a link to the **terminfo curses** library */usr/lib/libtinfo.a*. If **-lcurses** is used in conjunction with the **-xenix**, **-x2.3**, or **-xout** options to **cc**(CP), to create XENIX/386 binaries (**x.out**), **-lcurses** is linked to the library */lib/386/Slibcurses.a*, which, in turn, is linked to the **terminfo curses** library */lib/386/Slibtinfo.a*. If **-lcurses** is used in conjunction with the **-M2** option to **cc**(CP), to create XENIX/286 binaries (**x.out**), **-lcurses** is linked to the */lib/[SLMC]libcurses.a* libraries.

The header file */usr/include/curses.h* defines **M_TERMINFO** if neither **M_TERMINFO** nor **M_TERMCAP** has yet been defined. (These variables can be set at compile time.) Depending on which variable is defined, the header file automatically includes either */usr/include/tinfo.h* or */usr/include/tcap.h*.

Specifying terminfo curses

While you will most commonly use whichever **curses** is established as the default at your site, **terminfo curses** can be specifically selected. Include the following line in your source code:

```
#include <curses.h>
```

and using the following flag on the **cc**(CP) command line:

```
-DM_TERMINFO file -ltinfo [libraries...]
```

Specifying termcap curses

While you will most commonly use whichever **curses** is established as the default at your site, **termcap curses** can be specifically selected. **Termcap curses** requires an extra library and may be used as follows. Include the following line in your source code:

```
#include <curses.h>
```

and compile with the following options:

```
-DM_TERMCAP file -ltcap -lterm [libraries...]
```

Changing the default

This default may be changed to use termcap curses. To change the default, modify `/usr/include/curses.h` to define `M_TERMCAP` instead of `M_TERMINFO`. If you make this change, the compiler includes `/usr/include/tcap.h` and links the default curses library to the termcap curses library.

For COFF libraries, link the file `/usr/lib/libcurses.a` to `/usr/lib/libtcap.a`. For x.out libraries, link the file `/lib/386/libcurses.a` to `/lib/386/libtcap.a`. Note that for non-386 x.out binaries, a number of links must be changed, one for each possible memory model. Link the files `/lib/[SMLC]libcurses.a` to `/lib/[SMLC]libtcap.a`.

Summary of parameters

The parameters in the following list are not global variables; this is a summary of the parameters used by the curses 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 defined in `<curses.h>`.)

bool `bf`

```
char **area, *boolnames[], *boolcodes[], *boolfnames[]
char *bp, *cap, *capname, *codename
char *filename, *fmt, *label, *name
char *numnames[], *numcodes[], *numfnames[]
char *str, *strnames[], *strcodes[], *strfnames[]
char *term, *type
```

ctype `c`, `ch`, `horch`, `vertch`

FILE `*infd`, `*outfd`

```
int attr, begin_x, begin_y, begline, bot, col, count
int dmaxcol, dmaxrow, dmincol, dminrow, *errret, fides
int (*init()), labfmt, labnum, line
int ms, ncols, newcol, newrow, nlines, numlines
int oldcol, oldrow, overlay
int pmincol, pminrow, (*putc()), row
int smaxcol, smaxrow, smincol, sminrow, start
int tenths, top, visibility, x, y
short pair, f, b, color, r, g, b
long p1, p2, p9
```

SCREEN `*new`

TERMINAL `*cur_term`, `*nterm`, `*oterm`

va_list `varglist`

```
WINDOW *curscr, *dstwin, *orig
WINDOW *pad, *srcwin, *stdscr, *win
```

List of routines

```

int addch(ch)
int addkey(str,bf) /* termcap only */
int addstr(str)
int attroff(attrs)
int attron(attrs)
int attrset(attrs)
int baudrate() /* terminfo only */
int beep()
int box(win, vertch, horch)
bool can_change_color() /* terminfo only */
int cbreak()
int clear()
int clearok(win, bf)
int clrtoebot()
int clrtoeol()
int color_content(color, &r, &g, &b) /* terminfo only */
int copywin(srcwin, dstwin, sminrow, smincol, dminrow, dmincol,
            dmaxrow, dmaxcol, overlay) /* terminfo only */
int crmode()
int curs_set(visibility) /* terminfo only */
int curon() /* termcap only */
int curoff() /* termcap only */
int def_prog_mode() /* terminfo only */
int def_shell_mode() /* terminfo only */
int del_curterm(oterm) /* terminfo only */
int delay_output(ms) /* terminfo only */
int delch()
int deleteln()
int delwin(win)
int doupdate() /* terminfo only */
int draino(ms) /* terminfo only */
WINDOW *dupwin(win)
int echo()
int echochar(ch)
int endwin()
int erase()
char erasechar()
int filter() /* terminfo only */
int fixterm() /* terminfo only */
int flash() /* terminfo only */
int flushing() /* terminfo only */
int garbagedlines(win, begline, numlines) /* terminfo only */
getbegyx(win, y, x) /* terminfo only */ /* macro */
getorg(win, y, x) /* termcap only */ /* macro */
int getch()
getmaxyx(win, y, x) /* terminfo only */ /* macro */
getdim(win, y, x) /* termcap only */ /* macro */
int getstr(str)
int getsyx(y, x) /* terminfo only */

```

```

int gettmode
getyx(win, y, x) /* macro */
int halfdelay(tenths) /* terminfo only */
bool has_colors() /* terminfo only */
int has_ic() /* terminfo only */
int has_il() /* terminfo only */
int idlok(win, bf) /* terminfo only */
cctype inch()
int init_color(color, r, g, b) /* terminfo only */
int init_pair(pair, f, b) /* terminfo only */
WINDOW *initscr()
int insch(ch)
int insertln()
int intrflush(win, bf) /* terminfo only */
int isendwin() /* terminfo only */
char *keyname(c) /* terminfo only */
int keypad(win, bf)
char killchar() /* terminfo only */
int leaveok(win, bf)
char *longname()
int meta(win, bf) /* terminfo only */
int move(y, x)
int mvaddch(y, x, ch)
int mvaddstr(y, x, str)
int mvcur(oldrow, oldcol, newrow, newcol)
int mvdelch(y, x)
int mvgetch(y, x)
int mvgetstr(y, x, str)
cctype mvinch(y, x)
int mvinsch(y, x, ch)
int mvprintw(y, x, fmt [, arg...])
int mvscanw(y, x, fmt [, arg...])
int mvwaddch(win, y, x, ch)
int mvwaddstr(win, y, x, str)
int mvwdelch(win, y, x)
int mvwgetch(win, y, x)
int mvwgetstr(win, y, x, str)
int mvwin(win, y, x)
cctype mvwinch(win, y, x)
int mvwinsch(win, y, x, ch)
int mvwprintw(win, y, x, fmt [, arg...])
int mvwscanw(win, y, x, fmt [, arg...])
int napms(ms) /* terminfo only */
WINDOW *newpad(nlines, ncols) /* terminfo only */
SCREEN *newterm(type, outfd, infd) /* terminfo only */
WINDOW *newwin(nlines, ncols, begin_y, begin_x)
int nl()
int nocbreak()
int nocrmode()
int nodelay(win, bf)
int noecho()

```

```

int nonl()
int noraw()
int notimeout(win, bf) /* terminfo only */
int overlay(srcwin, dstwin)
int overwrite(srcwin, dstwin)
int pair_content(pair, &f, &b) /* terminfo only */
int pechochar(pad, ch) /* terminfo only */
int pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol,
    smaxrow, smaxcol) /* terminfo only */
int prefresh(pad, pminrow, pmincol, sminrow, smincol,
    smaxrow, smaxcol) /* terminfo only */
int printw(fmt [, arg ...])
int putp(str) /* terminfo only */
int raw()
int refresh()
int reset_prog_mode() /* terminfo only */
int reset_tty() /* termcap only */
int reset_shell_mode() /* terminfo only */
int resetterm() /* terminfo only */
int set_tty() /* termcap only */
int resetty()
int restartterm(term, fildes, errret) /* terminfo only */
int ripoffline(line, init) /* terminfo only */
int saveterm() /* terminfo only */
int savetty()
int scanw(fmt [, arg ...])
int scr_dump(filename) /* terminfo only */
int scr_init(filename) /* terminfo only */
int scr_restore(filename) /* terminfo only */
int scroll(win)
int scrollok(win, bf)
TERMINAL *set_curterm(nterm) /* terminfo only */
int setterm(name) /* terminfo only */
SCREEN *set_term(new) /* terminfo only */
int setscreg(top, bot) /* terminfo only */
int setsyx(y, x) /* terminfo only */
int setupterm(term, fildes, errret) /* terminfo only */
int slk_clear() /* terminfo only */
int slk_init(labfmt) /* terminfo only */
char *slk_label(labnum) /* terminfo only */
int slk_noutrefresh() /* terminfo only */
int slk_refresh() /* terminfo only */
int slk_restore() /* terminfo only */
int slk_set(labnum, label, labfmt) /* terminfo only */
int slk_touch() /* terminfo only */
int standend()
int standout()
int start_color() /* terminfo only */
WINDOW *subpad(orig, nlines, ncols, begin_y, begin_x) /* terminfo only */
WINDOW *subwin(orig, nlines, ncols, begin_y, begin_x)
int tgetent(bp, name)

```

(S)


```

int tgetflag(codename)
int tgetnum(codename)
char *tgetstr(codename, area)
char *tgoto(cap, col, row)
int tigetflag(capname) /* terminfo only */
int tigetnum(capname) /* terminfo only */
char *tigetstr(capname) /* terminfo only */
int touchline(win, start, count) /* terminfo only */
int touchwin(win)
char *tparm(str, p1, p2, ..., p9)
int tputs(str, count, putc)
int traceoff() /* terminfo only */
int traceon() /* terminfo only */
int typeahead(fildes) /* terminfo only */
char *unctrl(c)
int ungetch(c)
int vidattr(attrs) /* terminfo only */
int vidputs(attrs, putc) /* terminfo only */
int vwprintw(win, fmt, varglist) /* terminfo only */
int vwscanw(win, fmt, varglist) /* terminfo only */
int waddch(win, ch)
int waddstr(win, str)
int wattroff(win, attrs)
int wattron(win, attrs)
int wattrset(win, attrs)
int wclear(win)
int wclrto bot(win)
int wclrtoeol(win)
int wdelch(win)
int wdeleteln(win)
int wechochar(win, ch)
int werase(win)
int wgetch(win)
int wgetstr(win, str)
chtype winch(win)
int winsch(win, ch)
int winsertln(win)
int wmove(win, y, x)
int wnoutrefresh(win) /* terminfo only */
int wprintw(win, fmt [, arg...])
int wrefresh(win)
int wscanw(win, fmt [, arg...])
int wsetscrreg(win, top, bot) /* terminfo only */
int wstandend(win)
int wstandout(win)

```

Description

The **curses** routines give the user a terminal-independent method of updating screens with reasonable optimization.

The file `<curses.h>` must be included at the beginning of programs that use any **curses** routines. In addition, 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. To do this, execute `tput init` command after the shell environment variable **TERM** has been exported. For further details, see `profile(M)`, `tput(C)`, and the "Tabs and Initialization" subsection of `terminfo(M)`.

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 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 be underlined or shown 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`.

(S)

Routines that manipulate color on color alphanumeric terminals are new in this release of **curses**. To use these routines **start_color()** must be called, usually right after **initscr()**. Colors are always used in pairs (referred to as color-pairs). A color-pair consists of a foreground color (for characters) and a background color (for the field the characters are displayed on). A programmer initializes a color-pair with the routine **init_pair()**. After it has been initialized, **COLOR_PAIR(n)**, a macro defined in **<curses.h>**, can be used in the same ways other video attributes can be used. If a terminal is capable of redefining colors the programmer can use the routine **init_color()** to change the definition of a color. The routines **has_color()** and **can_change_color()** return **TRUE** or **FALSE**, depending on whether the terminal has color capabilities and whether the user can change the colors. The routine **color_content()** allows a user to identify the amounts of red, green, and blue components in an initialized color. The routine **pair_content()** allows a user to find out how a given color-pair is currently defined.

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 causes 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" function. More information on using **curscr** is provided where its use is appropriate.

If the environment variable **TERMINFO** is defined, any program using **curses** checks for a local terminal definition before checking in the standard place. For example, if the environment variable **TERM** is set to **wy60**, then the compiled terminal definition is found in **/usr/lib/terminfo/w/wy60**. (The **w** is copied from the first letter of **wy60** to avoid creation of huge directories.) However, if **TERMINFO** is set to **\$HOME/myterms**, **curses** first checks **\$HOME/myterms/w/wy60**, and, if that fails, then checks **/usr/lib/terminfo/w/wy60**. 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 are filled in by **initscr()** with the size of the screen. (For more information, see the subsection "Terminfo-Level Manipulations.") The integer variables **COLORS** and **COLOR_PAIRS** are also defined in **<curses.h>** and contain, respectively, the maximum number of colors and color-pairs the terminal can support. They are initialized by **start_color()**. 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 Syntax 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.

Sometimes the description of a routine refers to a second routine. If the routine referred to is prefixed with a **w**, then you should assume that other versions of the second routine behave similarly. For example, the description of **initscr()** refers to **wrefresh()**. This implies that the same result occurs if **refresh()** is called.

See the list above to determine which routines are **termcap** curses and which are **terminfo** curses only.

Overall screen manipulation

WINDOW *initscr() The first routine called should almost always be **initscr()**. (The exceptions are **slk_init()**, **filter()**, and **ripoffline()**.) This determines the terminal type and initializes all curses data structures. **initscr()** also arranges that the first call to **wrefresh()** clears the screen. If errors occur, **initscr()** writes 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(S)** call, for example. This routine restores **tty(M)** modes, moves the cursor to the lower left corner of the screen, and resets 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(S)** 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 copies 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()` 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 are 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 is at position `(y, x)`. If the move would cause any portion of the window to be moved 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 affect the character image of both windows. When changing the image of a subwindow, it is necessary to call `touchwin()` or `touchline()` on `orig` before calling `wrefresh()` on `orig`.

delwin(win) Delete the named window, freeing up all memory associated with it. If you try to delete a main window before all of its subwindows have been deleted, ERR is returned.

dupwin(win) Duplicate the named 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 is on the screen at one time. Automatic refreshes of pads (for example, 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 affect the character image of both windows. When changing the image of a subwindow, it is necessary to call **touchwin()** or **touchline()** on **orig** before calling **prefresh()** on **orig**.

prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, maxcol)

pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, maxcol)

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 manipulate text in 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(S)**). At the right margin, an automatic newline is performed. At the bottom of the scrolling region, if **scrollok()** is enabled, the scrolling region is scrolled up one line.

If **ch** is a tab, newline, or backspace, the cursor is moved appropriately within the window. A newline also does a **wclrtoeol()** before moving. Tabs are considered to be at every eighth column. If **ch** is another control character, it is drawn in the ^X notation. (Calling **winch()** on a position in the window containing a control character does not return the control character, but instead returns one character of the representation of the control character.)

Video attributes can be combined with a character by or-ing them into the parameter. This results in these attributes also being set. (The intent here is that text, including attributes, can be copied from one place to another using **winch()** and **waddch()**.) See **wstandout()**, below.

Note that **ch** is actually of type **chtype**, not a character. Also 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. Also 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 the constants **A_STANDOUT**, **A_REVERSE**, **A_BOLD**, **A_DIM**, **A_BLINK**, **A_UNDERLINE**, and **A_ALTCHARSET**, as well as the macro **COLOR_PAIR(n)**. These attributes 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 are displayed as the graphic rendition of the characters put on the screen.

wattrset(win, attrs) sets the current attributes of the given window to **attrs**. **wattroff(win, attrs)** turns off the named attributes without turning on or off any other attributes. **wattron(win, attrs)** turns on the named attributes without affecting any others. **wstandout(win, attrs)** is the same as **wattron(win, A_STANDOUT)**. **wstandend(win, attrs)** is the same as **wattrset(win, 0)**, that is, it turns off all attributes.

Note that **wattroff()**, **wattron()**, **wattrset()**, **wstandend()**, and **wstandout()** return 1 at all times.

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()** sounds the audible alarm on the terminal, if possible, and if not, flashes the screen (visible bell), if that is possible. **flash()** flashes the screen, and if that is not possible, sounds the audible signal. If neither signal is possible, nothing happens. 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**, are 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 is cleared completely on the next call to **wrefresh()** for that window, and repainted from scratch. Note that **clear()** is a macro.

clrtoobot()
wclrtoobot(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 **clrtoobot()** 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. Note that getyx() is a macro, so no “&” is necessary before the variables y and x.
getbegyx(win, y, x) getmaxyx(win, y, x)	The current beginning coordinates (getbegyx()) or size (getmaxyx()) of the specified window are placed in the two integer variables y and x. Note that getbegyx() and getmaxyx() are macros, so no “&” is necessary before the variables y and x.
getorg(win, y, x) getdim(win, y, x)	These are the termcap functions that get the beginning coordinates and dimensions of the specified window.
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, 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. Also note that insch() , mvinsch() , and mvwinsch() are macros.

insertln() wininsertln(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) <i>y</i> , column <i>x</i> . This does not move the physical cursor of the terminal until wrefresh() 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 text from <i>srcwin</i> on top of text from <i>dstwin</i> wherever the two windows overlap. 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 finer control over the overlay() and overwrite() routines. As in the prefresh() routine, a rectangle is specified in the destination window, (<i>dminrow</i> , <i>dmincol</i>) and (<i>dmaxrow</i> , <i>dmaxcol</i>), and the upper-left-corner coordinates of the source window, (<i>sminrow</i> , <i>smincol</i>). 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(S) . The string which would be output by printf(S) is instead output using waddstr() on the given window.
vwprintw(win, fmt, varlist)	This routine corresponds to vfprintf(S) . It performs a wprintw() using a variable argument list. The third argument is a <i>va_list</i> , a pointer to a list of arguments, as defined in <i><varargs.h></i> . See the vprintf(S) and varargs(S) 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.

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 affects the other window, but the records of which lines have been changed in the other window does 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 hangs until the system passes text through to the program. Depending on the setting of **cbreak()**, this is after one character (CBREAK mode), or after the first newline (NOCBREAK mode). In HALF-DELAY mode, the program hangs until a character is typed or the specified timeout has been reached. Unless **noecho()** has been set, the character is also echoed into the designated window.

When **wgetch()** is called, before getting a character, it calls **wrefresh()** if anything in the window has changed (for example, the cursor has moved or text changed).

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(M)** driver when each character is typed, the program may produce undesirable results.

If **wgetch()** encounters a **^D**, it is returned (unlike **stdio** routines, which would return a null string and have a return code of -1).

If **keypad(win, TRUE)** has been called, and a function key is pressed, the token for that function key is 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** sets a timer. If the remainder of the sequence is not received within the designated time, the character is passed through, otherwise the function key value is returned. For this reason, on many terminals, there is 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 `wgetch()` is made, until a newline, carriage return, or enter key is received. The resulting value (except for this terminating character) is placed in the area pointed at by the character pointer `str`. The user's erase and kill characters are interpreted. See `wgetch()` for how it handles characters differently from `stdio` routines (especially `^D`).

Note that `getstr()`, `mvgetstr()`, and `mvwgetstr()` are macros.

`ungetch(c)` Place `c` onto the input queue, to be returned by the next call to `wgetch()`.

`flushinp()` Throws away any typeahead that has been typed by the user and has not yet been read by the program. Note that `flushinp()` does not throw away any characters supplied by `ungetch()`.

`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 are 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(S)`, as do their arguments and return values. `wgetstr()` is called on the window, and the resulting line is used as input for the scan. The return value for these routines is the number

of **arg** values that are converted by **fmt**. **arg** values that are not converted are lost. See **wgetstr()** for how it handles strings differently from the **stdio** routines (especially **^D**).

vwscanw(win, fmt, ap)

This routine is similar to **vwprintw()** in that it 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(S)** and **varargs(S)** 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 clears the screen completely and redraws 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** considers using the hardware "insert/delete-line" feature of terminals so equipped. If disabled (**bf** is **FALSE**), **curses** very seldom uses 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** redraws the changed portions of all lines. Not calling **idlok()** saves approximately 5000 bytes of memory.

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 1 causes 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 are probably be used by the output routines.)

Note that `setscrreg()` is a macro.

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()`.)

Note that `scrollok()` always returns `OK`.

Input options setting

These routines set options within `curses` that deal with input. The options involve using `ioctl(S)` and therefore interact with `curses` routines. It is not necessary to turn these options off before calling `endwin()`.

cbreak()
crmode()
nocbreak()
nocrmode()

These 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 buffers characters typed until a newline or carriage return is typed. Interrupt and flow-control characters are unaffected by this mode (see `termio(M)`). Initially the terminal may or may not be in `CBREAK` mode, as it is inherited, therefore, a program should explicitly call `cbreak()` or `crmode()` to be placed in `CBREAK` mode and `nocbreak()` or `nocrmode()` to be taken out of `CBREAK` mode. Most interactive programs using `curses` set `CBREAK` mode.

Note that **cbreak()** performs a subset of the functionality of **raw()**. See **wgetch()** 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 **wgetch()** as they are typed. Echoing by the tty driver is always disabled, but initially **wgetch()** 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 **wgetch()** under “Input” for a discussion of how these routines interact with **cbreak()** and **nocbreak()**.

nl()
nonl()

These routines control whether carriage return is translated into newline on input by **wgetch()**. Initially, this translation is done; **nonl()** turns the translation off. Note that translation by the **tty(M)** driver is disabled in CBREAK mode.

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 is 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 is 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 **curses** to obtain information from the keypad of the user's terminal. If enabled, the user can press a function key (such as an arrow key) and **wgetch()** returns a single value representing the function key, as in **KEY_LEFT**. If disabled, **curses** does 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), calling **keypad (win, TRUE)** turns it on.

addkey(str, val)	This routine is available in termcap curses only. To add additional sequences to the keymode tree. str is the character sequence sent by the key. val is the integer to be returned when the key sequence is recognized on input.
meta(win, bf)	Initially, whether the terminal returns 7 or 8 significant bits on input depends on the control mode of the tty driver (see termio(M)). To force 8 bits to be returned, invoke meta (win, TRUE) . To force 7 bits to be returned, invoke meta (win, FALSE) . The window argument, win , is always ignored. If the terminfo(F) capabilities smm (meta_on) and rm (meta_off) are defined for the terminal, smm is sent to the terminal when meta (win, TRUE) is called and rm is sent when meta (win, FALSE) is called.
nodelay(win, bf)	This option causes wgetch() to be a non-blocking call. If no input is ready, wgetch() returns ERR . If disabled, wgetch() hangs until a key is pressed.
notimeout(win, bf)	While interpreting an input escape sequence, wgetch() sets a timer while waiting for the next character. If notimeout(win, TRUE) is called, then wgetch() does 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; however, in RAW mode, the interrupt, quit, suspend, and flow control characters are passed through uninterpreted, instead of generating a signal as they do in CBREAK mode. The behavior of the BREAK key depends on other bits in the tty(M) 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 is postponed until **wrefresh()** 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, is 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 is 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()** or **idlok()**.

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.

Color manipulation

This section describes the color manipulation routines introduced in this release of **curses**.

can_change_color() This routine requires no arguments. It returns **TRUE** if the terminal supports colors and can change their definitions, **FALSE** otherwise. This routine facilitates writing terminal-independent programs.

color_content(color, &r, &g, &b)
This routine gives users a way to find the intensity of the red, green, and blue (RGB) components in a color. It requires four arguments: the color number, and three addresses of **shorts** for storing the information about the amounts of red, green, and blue components in the given color. The value of the first argument must be between 0 and **COLORS-1**. The values that are stored at the addresses pointed to by the last three arguments are between 0 (no component) and 1000 (maximum amount of component). This routine returns **ERR** if the color does not exist (the first argument is outside the valid range), or if the terminal cannot change color definitions, **OK** otherwise.

has_colors() This routine requires no arguments. It returns **TRUE** if the terminal can manipulate colors, **FALSE** otherwise. This routine facilitates writing terminal-independent programs. For example, a programmer can use it to decide whether to use color or some other video attribute.

init_color(color, r, g, b)
This routine changes the definition of a color. It takes four arguments: the number of the color to be changed followed by three RGB values (for the amounts of red, green, and blue components). (See the section **COLOR** for the default color index.) The value of the first argument must be between 0 and **COLORS-1**. The last three arguments must each be a value between 0 and 1000. When **init_color()** is used, all occurrences of that color on the screen immediately change to the new definition. It returns **OK** if it was able to change the definition of the color, **ERR** otherwise.

init_pair(pair, f, b) This routine changes the definition of a color-pair. It takes three arguments: the number of the color-pair to be changed, the foreground color number, and the background color number. The value of the first argument must be between 1 and **COLOR_PAIRS-1**. The value of the second and third arguments must be

between 0 and **COLORS-1**. If the color-pair was previously initialized, the screen is refreshed and all occurrences of that color-pair are changed to the new definition. The routine returns **OK** if it was able to change the definition of the color-pair, **ERR** otherwise.

pair_content(pair, &f, &b)

This routine allows users to find out what colors a given color-pair consists of. It requires three arguments: the color-pair number, and two addresses of **shorts** for storing the foreground and the background color numbers. The value of the first argument must be between 1 and **COLOR_PAIRS-1**. The values that are stored at the addresses pointed to by the second and third arguments are between 0 and **COLORS-1**. The routine returns **ERR** if the color_pair has not been initialized, **OK** otherwise.

start_color()

This routine requires no arguments. It must be called if the user wants to use colors, and before any other color manipulation routine is called. It is good practice to call this routine right after **initscr()**. **start_color()** initializes eight basic colors (black, blue, green, cyan, red, magenta, yellow, and white), and two global variables, **COLORS** and **COLOR_PAIRS** (respectively defining the maximum number of colors and color-pairs the terminal can support). It also restores the terminal's colors to the values they had when the terminal was just turned on. It returns **ERR** if the terminal does not support colors, **OK** otherwise.

Soft labels

If desired, **curses** manipulates 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** takes over the bottom line of **stdscr**, reducing the size of **stdscr** and the variable **LINES**. **curses** standardizes on 8 labels of 8 characters each. If a **curses** program changes the values of the soft labels, it can restore them only to the default settings for that terminal. Therefore, if before calling a **curses** program a user changes the values of the soft labels, those values cannot be reset when the **curses** program terminates.

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 puts 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()** most likely soon follows.

char *slk_label(labnum)

The current label for label number **labnum** is returned, in the same format as it was in when it was passed to **slk_set()**; that is, how it looked prior to being justified according to the **labfmt** argument of **slk_set()**.

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.

set_tty()

reset_tty()

These routines are available only in **termcap** **curses** and perform the same functions as **reset_shell_mode()** and **reset_prog_mode()**.

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**. If **leaveok()** is currently **TRUE**, then -1,-1 is returned. If lines have been removed from the top of the screen using **ripline()**, **y** and **x** include these lines; therefore, **y** and **x** should be used only as arguments for **setsyx()**.

Note that **getsyx()** is a macro, so no "&" is necessary before the variables **y** and **x**.

setsyx(y, x)

The virtual screen cursor is set to **y, x**. If **y** and **x** are both -1, then **leaveok()** is 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 change 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()**.

ripline(line, init)

This routine provides access to the same facility that **slk_init()** uses to reduce the size of the screen. **ripline()** must be called before **initscr()** or **newterm()** is called. If **line** is positive, a line is removed from the top of **stdscr**; if negative, a line is 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.

ripline() 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**.

dmp_win(win,fp,margin)

This routine is available only through **termcap** **curses** and performs the same function as **scr_dump()**.

- scr_restore(filename)** The virtual screen is set to the contents of *filename*, which must have been written using **scr_dump()**. **ERR** is returned if the contents of *filename* are not compatible with the current release of **curses** software. The next call to **doupdate()** restores 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** bases 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(S)** call to share the screen with another process which has done a **scr_dump()** after its **endwin()** call. The data is declared invalid if the **-terminfo(F)** capability **nrrmc** is true or the time-stamp of the tty is old. Note that **keypad()**, **meta()**, **slk_clear()**, **curs_set()**, **flash()**, and **beep()** do not affect the contents of the screen, but makes the tty's time-stamp old.
- curs_set(visibility)** The cursor state is set to invisible, normal, or very visible for **visibility** equal to 0, 1 or 2. If the terminal supports the **visibility** requested, the previous cursor state is returned; otherwise, **ERR** is returned.
- curoff()**
curoff()
is_curoff() These routines are available only through **termcap** **curses** and turn the cursor display on and off, respectively.
- draino(ms)** Wait until the output has drained enough that it only takes **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.

mvcur(oldrow, oldcol, newrow, newcol)
Low-level cursor motion.

Terminfo-Level manipulations

These low-level routines must be called by programs that need to deal directly with the **terminfo(F)** 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 defines the set of terminal-dependent variables defined in the **terminfo(F)** database. The **terminfo(F)** variables **lines** and **columns** (see **terminfo(F)**) 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(C)**), the size of the current layer is used. Otherwise, the values for **lines** and **columns** specified in the **terminfo(F)** 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(F)** 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(F)**). (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(F)** database, initializing the **terminfo(F)** 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** is used. All output is to the file descriptor **fildes**. If **errret** is not **NULL**, then **setupterm()** returns **OK** or **ERR** and stores 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(F)** database could not be found. If **errret** is **NULL**, **setupterm()** prints an error message upon finding an error and exits. Thus, the simplest call is **setupterm ((char *)0, 1, (int *)0)**, which uses all the defaults.

The `terminfo(F)` 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(F)` boolean, numeric and string variables refer to. The pointer may be saved before calling `setupterm()` again. Further calls to `setupterm()` allocates new space rather than reuses 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(F)` 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(F)` boolean, numeric and string variables thereafter may refer to invalid memory locations until another `setupterm()` has been called.

restartterm(term, fildes, errret)

Similar to `setupterm()`, except that it is called after restoring memory to a previous state; for example, after a call to `scr_restore()`. It assumes that the windows and the input and output options are the same as when memory was saved, but the terminal type and baud rate may be different.

char *tparm(str, p₁, p₂, ..., p₉)

Instantiate the string `str` with parms `pi`. 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(F)` 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(S)`-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(S)`-like routine `putc()`.

vidattr(attrs) Similar to `vidputs()`, except that it outputs through `putchar(S)`.

The following routines return the value of the capability corresponding to the character string containing the terminfo(F) capname passed to them. For example, `rc = tigetstr("acsc")` causes the value of `acsc` to be returned in `rc`.

tigetflag(capname) The value -1 is returned if `capname` is not a boolean capability. The value 0 is returned if `capname` is not defined for this terminal.

tigetnum(capname) The value -2 is returned if `capname` is not a numeric capability. The value -1 is returned if `capname` is not defined for this terminal.

tigetstr(capname) The value (char *) -1 is returned if `capname` is not a string capability. A null value is returned if `capname` is not defined for this terminal.

`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(F) 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(F) 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`.

tigetnum(codename) 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, <code>/usr/lib/libdcurses.a</code> . 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 <code>c</code>. Control characters are displayed in the <code>^X</code> notation. Printing characters are displayed as is. <code>unctrl()</code> is a macro, defined in <code><unctrl.h></code>, which is automatically included by <code><curses.h></code>.
char *keyname(c)	A character string corresponding to the key <code>c</code> is returned.
filter()	This routine is one of the few that is to be called before <code>initscr()</code> or <code>newterm()</code> is called. It arranges things so that <code>curses</code> thinks that there is a 1-line screen. <code>curses</code> does 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 causes 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 is 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 <code>cbreak()</code> .
fixterm()	Replaced by <code>reset_prog_mode()</code> .
gettmode()	A no-op.
nocrmode()	Replaced by <code>nocbreak()</code> .

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 **wattron()**, **wattroff()**, and **wattrset()**, or OR'ed with the characters passed to **waddch()**.

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
COLOR_PAIR(n)	Color_pair defined in n (Note that this is a macro.)
A_CHARTTEXT	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: wattrset (win, A_NORMAL))
A_COLOR	Bit-mask to extract color_pair field information
PAIR_NUMBER(attrs)	Returns the pair number associated with the COLOR_PAIR(n) attribute. (Note that this is a macro.)

Colors

In `<curses.h>` the following macros are defined to have the numeric value shown. These are the default colors. `curses` also assumes that color 0 (zero) is the default background color for all terminals.

COLOR_BLACK	0
COLOR_RED	1
COLOR_GREEN	2
COLOR_YELLOW	3
COLOR_BLUE	4
COLOR_MAGENTA	5
COLOR_CYAN	6
COLOR_WHITE	7

Note that the ANSI console driver uses the ISO color mapping scheme, as described in `<curses.h>`.

Function keys

The following function keys, defined in `<curses.h>`, might be returned by `wgetch()` 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(F)` 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

(Continued on next page)

(Continued)

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

(Continued on next page)

(Continued)

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 has the `A_ALTCHARSET` bit turned on. Otherwise, the default character listed below is 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

See also

cc(CP), ld(CP), ioctl(S), plot(S), putc(S), scanf(S), stdio(S), system(S),
vprintf(S), profile(M), terminfo(F), varargs(S).

termio(M), tty(M).

Diagnostics

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 getsyx(), getyx(), getbegyx(), getmaxyx(). For these macros, no useful value is returned.

Routines that return pointers always return (type *) NULL on error.

Warnings

The plotting library `plot(S)` and the curses library `curses(S)` both use the names `erase()` and `move()`. The curses versions are macros. If you need both libraries, put the `plot(S)` code in a different source file from the `curses(S)` code, and/or `#undef move()` and `erase()` in the `plot(S)` 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(S)`) for output during that time can cause unpredictable results.

If a pointer passed to a routine as a window argument is null or out of range, the results are undefined (core may be dumped).

Notes

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 may change 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.

Standards conformance

curses is conformant with:

AT&T SVID Issue 2.

cuserid

get character login name of the user

Syntax

```
cc ... -lc

#include <stdio.h>

char *cuserid (s)
char *s;
```

Description

The **cuserid** routine creates a character-string of the login name that the owner of the current process is logged in under. If *s* is a NULL pointer, this string is generated in an internal static area, the address of which is returned by **cuserid**. Otherwise, the string is left in the array, assumed to contain at least **L_cuserid** characters, to which *s* points. The constant **L_cuserid** is defined in the *<stdio.h>* header file.

Return value

If the argument *s* is not NULL and the routine is successful, **cuserid** returns *s*. If *s* is NULL, the routine returns the address where the string is internally stored. If the login name is not found and *s* is NULL, **cuserid** returns a NULL pointer. If the login name is not found and *s* is not NULL, a null character('\0') will be placed in *s[0]* and *s* is returned by **cuserid**.

Note

When *s* is NULL, as mentioned above, the string is stored in an internal static area whose contents are overwritten by each call to **cuserid**.

The **cuserid** routine uses the **getpwuid** routine, and therefore the results of a **getpwuid** or a **getpwnam** routine may be destroyed by a subsequent call to **cuserid**.

See also

getlogin(S), **getpwent(S)**

Standards conformance

cuserid conforms with:

X/Open Portability Guide, Issue 3, 1989;

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

(S)

dblock: enter_quiet_zone, exit_quiet_zone, make_transition_files, replace_file

authentication database locking routines

Syntax

```
cc ... -lprot -lx

#include <sys/types.h>
#include <prot.h>

void enter_quiet_zone ()

void exit_quiet_zone ()

int make_transition_files (pathname, ptemp, pold)
register char *pathname;
register char **ptemp;
register char **pold;

int replace_file (temp, pathname, old)
register char *temp;
register char *pathname;
register char *old;
```

Description

enter_quiet_zone - block all keyboard signals

exit_quiet_zone - unblock keyboard signals

make_transition_files - create transition file names

replace_file - perform a database update using transition file names

The **enter_quiet_zone(S)** and **exit_quiet_zone(S)** routines provide a way to enter and exit regions of code where the process is undisturbed by all external signals. This is helpful on sensitive regions of code where the security state is being altered and to disrupt the entire operation would leave the system in an insecure state or a state from which it is hard to recover.

enter_quiet_zone() blocks off all keyboard signals, including **SIGHUP**, **SIGINT**, and **SIGQUIT**. The result of this blocking is that delivery of any signals will be delayed until they are unblocked in the call to **exit_quiet_zone()**.

Pairs of calls to **enter_quiet_zone()** and **exit_quiet_zone()** may be nested, but it is important that the number of **exit_quiet_zone()** calls match the number of **enter_quiet_zone()** calls. Signals are blocked on the first call to **enter_quiet_zone()**, and only unblocked on the last call to **exit_quiet_zone()**.

The routines `make_transition_files(S)` and `replace_file(S)` are used together to update one of the authentication database files.

`make_transition_files()` takes the input argument *pathname* and generates two new file names. Pointers to the new file names are stored in the locations referenced by the pointers *poldp* and *ptempp*. The pointer referenced by *poldp* points to the name used to rename the existing file referenced by *pathname* so that it may be recovered if the update fails. The pointer referenced by *ptempp* points to the name to be used for the new file intended to replace the existing file referenced by *pathname*. Note that `malloc(S)` is used to create space for *poldp* and *ptempp*, and that files are not actually opened or otherwise referenced by `make_transition_files()`.

`replace_file()` uses the same arguments (those given to and modified by a previous call to `make_transition_files()`) to manipulate the actual files. It expects that *pathname* references the existing file, *tempp* references the newly created file, and *oldp* references the place that the existing file is to be moved to should an error occur in placing the new file in *pathname*. `replace_file()` frees the space allocated for the names in `make_transition_files()`.

Notes

Any program calling these routines should call `set_auth_parameters(S)` as the first action in `main()`.

Return values

`make_transition_files()` returns a value of 1 if the names were generated, and 0 if they were not.

`replace_file()` returns a value of 1 if the file operations were successful and 0 if they were not.

See also

`identity(S)`, `sigprocmask(S)`, `malloc(S)`

Standards conformance

The `dblock` routines are extensions of AT&T System V provided by the Santa Cruz Operation.

dbm: dbminit, delete, fetch, firstkey, nextkey, store

performs database functions

Syntax

```
cc ... -ldbm

#include <dbm.h>

typedef struct ( char *dptr; int dsize; ) datum;

int dbminit(file)
char *file;

int delete(key)
datum key;

datum fetch(key)
datum key;

datum firstkey();

datum nextkey(key);
datum key;

int store(key, content)
datum key, content;
```

Description

These functions maintain key/content pairs in a database. The functions handle very large (a billion blocks) databases and access a keyed item in one or two file system accesses. The functions are obtained with the loader option **-ldbm**.

keys and *contents* are described by the **datum** typedef. A **datum** specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The database is stored in two files. One file is a directory containing a bit map and has *.dir* as its suffix. The second file contains all data and has *.pag* as its suffix.

Before a database can be accessed, it must be opened by **dbminit**. At the time of this call, the files *file.dir* and *file.pag* must exist. (An empty database is created by creating zero-length *.dir* and *.pag* files.)

Once open, the data stored under a key is accessed by **fetch** and data is placed under a key by **store**. A key (and its associated contents) is deleted by **delete**. A linear pass through all keys in a database may be made, in an (apparently)

random order, by use of **firstkey** and **nextkey**. **firstkey** returns the first key in the database. With any key **nextkey** returns the next key in the database. This code traverses the database:

```
for(key=firstkey(); key.dptr!=NULL; key=nextkey(key))
```

Example

The example program below uses **dbm**, **dbminit**, **store**, **fetch** and **delete** functions. It reads in keys and data from a datafile (shown below) and stores them in a database called *testfile*. Newlines ("**\n**") are used for delimiters. It then reads the keys from "datafile" and uses them to fetch the data out, print the data, and delete the record.

To run this program the files *testfile.dir* and *testfile.pag* must exist and be empty (0 bytes). The *datafile* and test program are as follows:

```
101
UNIX Programming
105
System Administration
234
Intro to UNIX
```

```
101
105
234
```

```

0 #include <stdio.h>
1 #include <dbm.h>
2 #define KEYSIZE      5
3 #define DATASIZE     25
4 datum key, data, testdata;
5 FILE *fp, *fopen();
6 char    keybuf[KEYSIZE];
7 char    keybuf2[KEYSIZE];
8 char    databuf1[DATASIZE];
9 main()
10 {
11 char c;
12 dbminit("testfile");
13 fp = fopen("datafile", "r");
14 while ((c = getc(fp)) != '\n') {
15     key.dptr = keybuf;
16     *key.dptr++ = c;
17     key.dsize = 1;
18     while((c = getc(fp)) != '\n') {
19         *key.dptr++ = c;
20         key.dsize++;
21     }
22     data.dsize = 0;
23     data.dptr = databuf1;
24     while((c = getc(fp)) != '\n') {
25         *data.dptr++ = c;
26         data.dsize++;
27     }
28     *data.dptr = '\0';
29     data.dsize++;
30     data.dptr = databuf1;
31     key.dptr = keybuf;
32     printf("datadsize %d keysize %d\n", data.dsize, key.dsize);
33     printf("datadptr %d keydptr %d\n", data.dptr, key.dptr);
34     store(key,data);
35 }
36 printf("\nData base loaded and now going for the read\n\n");
37 key.dptr = keybuf2;
38 while ((*key.dptr++ = getc(fp)) != EOF){
39     key.dsize = 1;
40     while ((c=getc(fp)) != '\n') {
41         *key.dptr++ = c;
42     }
43     key.dptr = keybuf2;
44     testdata = fetch(key);
45     printf("Key: %s Data: %s\n", key.dptr, testdata.dptr);
46     delete(key);
47     testdata = fetch(key);
48     printf("Deleted Key : %s Data : %s\n", key.dptr, testdata.dptr);
49 }
50 }

```

Lines 12-13	Initialize the database
Lines 14-15	Read in keys and data until a new line
Lines 16-20	Read in a key
Lines 22-26	Read in a data field
Lines 28-34	Store a record in a testfile database
Lines 38-41	Read in keys from the datafile and use
Lines 43-44	Fetch record specified by key
Line 46	Delete the record
Lines 47-49	Printf to show data is now null

Diagnostics

All functions that return an *int* indicate errors with negative values. A zero return indicates ok. Routines that return a *datum* indicate errors with a null (0) *dptr*.

Notes

The *.pag* file contains holes so that its apparent size is several times its actual content. Older version of some systems may create real file blocks for these holes when touched. These files cannot be copied by normal means (*cp*, *cat*, *tp*, *tar*, *ar*) without filling in the holes.

dptr pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 1024 bytes). Moreover all key/content pairs that hash together must fit on a single block. *store* returns an error in the event that a disk block fills with inseparable data.

delete does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by *firstkey* and *nextkey* depends on a hashing function.

These routines are not reentrant, so they should not be used on more than one database at a time.

Standards conformance

dbm is not part of any currently supported standard; it was developed at the University of California at Berkeley and is used by permission.

defopen, defread

reads default entries

Syntax

```
cc ... -lc

int defopen(filename)
char *filename;

char *defread(pattern)
char *pattern;
```

Description

defopen and **defread** are a pair of routines designed to allow easy access to default definition files. The UNIX operating system is normally distributed in binary form; the use of default files allows OEMs or site administrators to customize utility defaults without having the source code.

defopen opens the default file named by the pathname in *filename*. **defopen** returns null if it is successful in opening the file, or the **fopen** failure code (**errno**) if the open fails.

defread reads the previously opened file from the beginning until it encounters a line beginning with *pattern*. **defread** then returns a pointer to the first character in the line after the initial *pattern*. If a trailing newline character is read it is replaced by a null byte.

When all items of interest have been extracted from the opened file the program may call **defopen** with the name of another file to be searched, or it may call **defopen** with NULL, which closes the default file without opening another.

Files

The UNIX style convention is for a system program *xyz* to store its defaults (if any) in the file */etc/default/xyz*.

Diagnostics

defopen returns zero on success and nonzero if the open fails. The return value is the **errno** value set by **fopen(S)**.

defread returns NULL if a default file is not open, if the indicated pattern could not be found, or if it encounters any line in the file greater than the maximum length of 128 characters.

Notes

The return value points to static data, whose contents are overwritten by each call.

See also

fopen(S)

Standards conformance

defopen and **defread** are extensions of AT&T System V provided by the Santa Cruz Operation.

(S)

dial

establish an outgoing terminal line connection

Syntax

```
cc ... -lc

#include <dial.h>

int dial (call)
CALL call;

void undial (fd)
int fd;
```

Description

dial returns a file-descriptor for a terminal line open for read/write. The argument to **dial** is a **CALL** structure (defined in the *<dial.h>* header file).

When finished with the terminal line, the calling program must invoke **undial** to release the semaphore that has been set during the allocation of the terminal device.

The definition of **CALL** in the *<dial.h>* header file is:

```
typedef struct (
    struct termio *attr;    /* pointer to termio attribute struct */
    int    baud;           /* transmission data rate */
    int    speed;          /* 212A modem: low=300, high=1200 */
    char   *line;          /* device name for outgoing line */
    char   *telno;         /* pointer to tel-no digits string */
    int    modem;          /* specify modem control
                           for direct lines */
    char   *device;        /* unused */
    int    dev_len;        /* unused */
) CALL;
```

The **CALL** element **speed** is intended only for use with an outgoing dialed call, in which case its value should be either 300 or 1200 to identify the 113A modem, or the high- or low-speed setting on the 212A modem. Note that the 113A modem or the low-speed setting of the 212A modem transmits at any rate between 0 and 300 bits per second. However, the high-speed setting of the 212A modem transmits and receives at 1200 bits per second only. The **CALL** element **baud** is for the desired transmission baud rate. For example, one might set **baud** to 110 and **speed** to 300 (or 1200). However, if **speed** is set to 1200, **baud** must be set to high (1200).

If the desired terminal line is a direct line, a string pointer to its device-name should be placed in the line element in the CALL structure. Legal values for such terminal device names are kept in the *Devices* file. In this case, the value of the baud element should be set to -1. This causes dial to determine the correct value from the *Devices* file.

The telno element is for a pointer to a character string representing the telephone number to be dialed. Such numbers may consist only of these characters:

- 0-9 dial 0-9
- * dial *
- # dial #
- = wait for secondary dial tone
- delay for approximately 4 seconds

The CALL element modem is used to specify modem control for direct lines. This element should be non-zero if modem control is required. The CALL element attr is a pointer to a termio structure, as defined in the *<termio.h>* header file. A NULL value for this pointer element may be passed to the dial function, but if such a structure is included, the elements specified in it is set for the outgoing terminal line before the connection is established. This is often important for certain attributes such as parity and baud-rate.

The CALL elements device and dev_len are no longer used. They are retained in the CALL structure for compatibility reasons.

Diagnostics

On failure, a negative value indicating the reason for the failure is returned. Mnemonics for the negative indices as listed here are defined in the *<dial.h>* header file.

INTRPT	-1	/* interrupt occurred */
D_HUNG	-2	/* dialer hung (no return from write) */
NO_ANS	-3	/* no answer within 10 seconds */
ILL_BD	-4	/* illegal baud-rate */
A_PROB	-5	/* acu problem (open() failure) */
L_PROB	-6	/* line problem (open() failure) */
NO_Ldv	-7	/* can't open <i>Devices</i> file */
DV_NT_A	-8	/* requested device not available */
DV_NT_K	-9	/* requested device not known */
NO_BD_A	-10	/* no device available at requested baud */
NO_BD_K	-11	/* no device known at requested baud */
DV_NT_E	-12	/* requested speed does not match */
BAD_SYS	-13	/* system not in <i>Systems</i> file */

Warnings

Including the `<dial.h>` header file automatically includes the `<termio.h>` header file.

The above routine uses `<stdio.h>`. This causes the routine to increase the size of programs that are not using standard I/O more than might be expected.

Notes

An `alarm(S)` system call for 3600 seconds is made (and caught) within the `dial` module for the purpose of "touching" the `LCK...` file and constitutes the device allocation semaphore for the terminal device. The `LCK...` file prevents `uucp(C)` from trying to transmit more than one file at a time using the particular device. If the `LCK...` file is not "touched" then `uucp(C)` may simply delete the file on its 90-minute clean-up rounds. The alarm may go off while the user program is in a `read(S)` or `write(S)` system call, causing an apparent error return. If the user program expects to be around for an hour or more, error returns from `reads` should be checked for (`errno==EINTR`), and the `read` possibly re-issued.

Files

`/usr/lib/uucp/Devices`
`/usr/lib/uucp/Systems`
`/usr/spool/locks/LCK..tty-device`

See also

`alarm(S)`, `read(S)`, `termio(M)`, `uucp(C)`, `write(S)`

Standards conformance

`dial` and `undial` are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

difftime

computes the difference between time values

Syntax

```
cc ... -lc

#include <time.h>

double difftime(time2, time1)
time_t time2;
time_t time1;
```

Description

The **difftime** function computes the difference of *time2* – *time1*.

See the example which follows.

Return value

The **difftime** function returns the elapsed time in seconds from *time1* to *time2* as a double-precision number.

See also

ctime(S), **mktime(S)**, **nl_cxtime(S)**, **stime(S)**, **time(S)**

Standards conformance

difftime is conformant with:

ANSI X3.159-1989 Programming Language — C.

Example

```

#include <time.h>

int mark[10000];

main()
{
    time_t start, finish;
    register int i, loop, n, num, step;
    printf("This program takes about 3 minutes ");
    printf("on an AT and 8 on a PC.\n Working...\n");
    time(&start);

    for (loop = 0; loop < 1000; ++loop)
        for (num = 0, n = 3; n < 10000; n += 2)
            if (!mark[n])
            {
                /* printf("%d\n",n); */
                step = 2*n;
                for (i = 3*n; i < 10000; i += step)
                    mark[i] = -1;
                ++num;
            }
    time(&finish);

    /* Prints average of 1000 loops through "sieve": */
    printf("\nProgram takes %f seconds to find %d primes.\n",
        difftime(finish, start), num);
}

```

Output:

Program takes 0.482000 seconds to find 1228 primes.

This program calculates the amount of time needed to find the prime numbers between 3 and 10,000. To display the prime numbers, delete the outermost loop and the comment delimiters around the expression `printf("%d",n);`.

directory: closedir, opendir, readdir, rewinddir, seekdir, telldir

directory operations

Syntax

```
cc ... -lc

#include <sys/types.h>
#include <dirent.h>

void closedir(dirp)
DIR *dirp;

DIR *opendir (filename)
char *filename;

struct dirent *readdir (dirp)
DIR *dirp;

void rewinddir (dirp)
DIR *dirp;

void seekdir (dirp, loc)
DIR *dirp;
long loc;

long telldir (dirp)
DIR *dirp;
```

Description

opendir opens the directory named by *filename* and associates a **directory** stream with it. **opendir** returns a pointer to be used to identify the **directory** stream in subsequent operations. The pointer NULL is returned if *filename* cannot be accessed or is not a directory, or if it cannot **malloc** enough memory to hold a DIR structure or a buffer for the directory entries.

readdir returns a pointer to the next active directory entry. No inactive entries are returned. It returns NULL upon reaching the end of the directory or upon detecting an invalid location in the directory.

telldir returns the current location associated with the named **directory** stream.

seekdir sets the position of the next **readdir** operation on the **directory** stream. The new position, *loc*, is the position associated with the **directory** stream when the **telldir** operation was performed. Values returned by **telldir** are good only if the directory has not changed due to compaction or

expansion. This is not a problem with System V, but it may be with some file system types.

rewinddir resets the position of the named **directory** stream to the beginning of the directory.

closedir closes the named **directory** stream and frees the **DIR** structure.

The following errors can occur as a result of these operations.

opendir:

- [EACCES]** A component of *filename* denies search permission.
- [EFAULT]** *filename* points outside the allocated address space.
- [EMFILE]** The maximum number of file descriptors are currently open.
- [ENOTDIR]** A component of *filename* is not a directory.

readdir:

- [EBADF]** The file descriptor determined by the **DIR** stream is no longer valid. This results if the **DIR** stream has been closed.
- [ENOENT]** The current file pointer for the directory is not located at a valid entry.

closedir:

- [EBADF]** The **dirp** argument does not refer to an open directory stream.
- [EINTR]** The **closedir** function was interrupted by a signal.

Example

Sample code which searches a directory for entry *name*:

```
dirp = opendir( "." );
while ( (dp = readdir( dirp )) != NULL )
    if ( strcmp( dp->d_name, name ) == 0 )
    {
        closedir( dirp );
        return FOUND;
    }
closedir( dirp );
return NOT_FOUND;
```

Warnings

rewinddir is implemented as a macro, so its function address cannot be taken.

See also

dirent(FP), getdents(S)

Standards conformance

closedir, opendir, readdir and **rewinddir** are conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

seekdir and **telldir** are conformant with:

AT&T SVID Issue 2;

and X/Open Portability Guide, Issue 3, 1989.

(S)

div

divides integers

Syntax

```
cc ... -lc

#include <stdlib.h>

struct div_t {
    int quot;      /* Quotient */
    int rem;       /* Remainder */
}

div_t div(numerator, denominator)
int numerator;
int denominator;
```

Description

The **div** function divides *numerator* by *denominator*, computing the quotient and the remainder. The sign of the quotient is the same as that of the mathematical quotient. Its absolute value is the largest integer that is less than the absolute value of the mathematical quotient. If the denominator is zero, the program terminates with an error message.

See the example which follows.

Return value

The **div** function returns a structure of type **div_t**, comprising both the quotient and the remainder. The structure is defined in *stdlib.h*.

See also

ldiv(S)

Standards conformance

div is conformant with:
ANSI X3.159-1989 Programming Language — C.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

main(argc, argv)
int argc;
char **argv;
{
    int x,y;
    div_t div_result;
    x = atoi(argv[1]);
    y = atoi(argv[2]);
    printf("x is %d, y is %d\n", x,y);
    div_result = div(x,y);
    printf("The quotient is %d, and the remainder is %d\n",
        div_result.quot, div_result.rem);
}
```

The example above takes two integers as command line arguments and displays the results of the integer division. This program accepts two arguments on the command line following the program name, then calls `div` to divide the first argument by the second. Finally, it prints the structure members `quot` and `rem`.

Assuming the executable file is named *tdiv*, it might be executed as:

```
tdiv 5 2
```

The output would read:

```
x is 5, y is 2
The quotient is 2, and the remainder is 1
```

drand48, erand48, jrand48, lcong48, lrand48, nrand48, mrand48, srand48, seed48

generate uniformly distributed pseudo-random numbers

Syntax

```
cc ... -lc

double drand48 ( )

double erand48 (xsubi)
unsigned short xsubi[3];

long jrand48 (xsubi)
unsigned short xsubi[3];

void lcong48 (param)
unsigned short param[7];

long lrand48 ( )

long nrand48 (xsubi)
unsigned short xsubi[3];

long mrand48 ( )

void srand48 (seedval)
long seedval;

unsigned short *seed48 (seed16v)
unsigned short seed16v[3];
```

Description

This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

Functions **drand48** and **erand48** return non-negative double-precision floating-point values uniformly distributed over the interval [0.0, 1.0).

Functions **lrnd48** and **nrand48** return non-negative long integers uniformly distributed over the interval [0, 2^{31}).

Functions **mrnd48** and **jrnd48** return signed long integers uniformly distributed over the interval [-2^{31} , 2^{31}).

Functions **srand48**, **seed48**, and **lcong48** are initialization entry points, one of which should be invoked before either **drand48**, **lrnd48**, or **mrnd48** is called. (Although it is not recommended practice, constant default initializer values

are supplied automatically if **drand48**, **lrand48**, or **mrand48** is called without a prior call to an initialization entry point.) Functions **erand48**, **nrand48**, and **jrand48** do not require an initialization entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values, X_i , according to the linear congruential formula

$$X_{n+1} = (aX_n + c)_{\text{mod } m} \quad n \geq 0.$$

The parameter $m = 2^{48}$; hence 48-bit integer arithmetic is performed. Unless **lcong48** has been invoked, the multiplier value a and the addend value c are given by

$$\begin{aligned} a &= 5\text{DEECE66D}_{16} = 273673163155_8 \\ c &= \text{B}_{16} = 13_8. \end{aligned}$$

The value returned by any of the functions **drand48**, **erand48**, **lrand48**, **nrand48**, **mrand48**, or **jrand48** is computed by first generating the next 48-bit X_i in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (leftmost) bits of X_i and transformed into the returned value.

The functions **drand48**, **lrand48**, and **mrand48** store the last 48-bit X_i generated in an internal buffer, and must be initialized prior to being invoked. The functions **erand48**, **nrand48**, and **jrand48** require the calling program to provide storage for the successive X_i values in the array specified as an argument when the functions are invoked. These routines do not have to be initialized; the calling program must place the desired initial value of X_i into the array and pass it as an argument. By using different arguments, functions **erand48**, **nrand48**, and **jrand48** allow separate modules of a large program to generate several independent streams of pseudo-random numbers; that is, the sequence of numbers in each stream does *not* depend upon how many times the routines have been called to generate numbers for the other streams.

The initializer function **srand48** sets the high-order 32 bits of X_i to the 32 bits contained in its argument. The low-order 16 bits of X_i are set to the arbitrary value 330E_{16} .

The initializer function **seed48** sets the value of X_i to the 48-bit value specified in the argument array. In addition, the previous value of X_i is copied into a 48-bit internal buffer used only by **seed48**, and a pointer to this buffer is the value returned by **seed48**. This returned pointer, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time — use the pointer to get at and store the last X_i value, and then use this value to reinitialize via **seed48** when the program is restarted.

The initialization function **lcong48** allows the user to specify the initial X_i , the multiplier value a , and the addend value c . Argument array elements **param[0-2]** specify X_i , **param[3-5]** specify the multiplier a , and **param[6]** specifies the 16-bit addend c . After **lcong48** has been called, a subsequent call

to either **srand48** or **seed48** restores the “standard” multiplier and addend values, *a* and *c*, specified on the previous page.

Notes

The source code for the portable version can be used on computers which do not have floating-point arithmetic. In such a situation, functions **drand48** and **erand48** are replaced by the two new functions below.

```
long irand48 (m)
unsigned short m;
```

```
long krand48 (xsubi, m)
unsigned short xsubi[3], m;
```

Functions **irand48** and **krand48** return non-negative long integers uniformly distributed over the interval $[0, m-1]$.

See also

rand(S)

Standards conformance

drand48, **erand48**, **jrand48**, **lrand48**, **mrand48**, **nrand48**, **seed48** and **srand48** are conformant with:

AT&T SVID Issue 2;
and X/Open Portability Guide, Issue 3, 1989.

lcong48 is conformant with:

AT&T SVID Issue 2.

dup

duplicate an open file descriptor

Syntax

```
cc ... -lc

int dup (fildes)
int fildes;
```

Description

The *fildes* argument is a file descriptor obtained from a **creat**, **open**, **dup**, **fcntl**, or **pipe** system call. The **dup** system call returns a new file descriptor having the following in common with the original:

- Same open file (or pipe)
- Same file pointer (that is, both file descriptors share one file pointer)
- Same access mode (read, write, or read/write)

The new file descriptor is set to remain open across **exec** system calls. See **fcntl(S)**.

The file descriptor returned is the lowest one available.

The **dup** system call fails if one or more of the following is true:

[EBADF]	The <i>fildes</i> argument is not a valid open file descriptor.
[EINTR]	A signal was caught during the dup system call.
[EMFILE]	NFILE file descriptors are currently open.
[ENOLINK]	<i>fildes</i> is on a remote machine and the link to that machine is no longer active.

Diagnostics

Upon successful completion a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

See also

close(S), **creat(S)**, **exec(S)**, **fcntl(S)**, **open(S)**, **pipe(S)**, **lockf(S)**

Standards conformance

dup is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

dup2

duplicate an open file descriptor

Syntax

```
cc ... -lc

int dup2 (fildes, fildes2)
int fildes, fildes2;
```

Description

The *fildes* argument is a file descriptor referring to an open file, and *fildes2* is a non-negative integer less than `_NFILE`. `dup2` causes *fildes2* to refer to the same file as *fildes*. If *fildes2* already referred to an open file, it is closed first.

The `dup2` function fails if one or more of the following is true:

- [EBADF] *fildes* is not a valid open file descriptor.
- [EMFILE] `_NFILE` file descriptors are currently open.

Diagnostics

Upon successful completion a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of -1 is returned, and `errno` is set to indicate the error.

See also

`creat(S)`, `close(S)`, `exec(S)`, `fcntl(S)`, `lockf(S)`, `open(S)`, `pipe(S)`

Standards conformance

`dup2` is conformant with:

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

ecvt, fcvt, gcvt

convert floating-point number to string

Syntax

cc ...-lc

```
char *ecvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;
```

```
char *fcvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;
```

```
char *gcvt (value, ndigit, buf)
double value;
int ndigit;
char *buf;
```

Description

ecvt - converts *value* to string of *ndigit*

fcvt - converts *value* to string of *ndigit* rounded for FORTRAN F-format

gcvt - converts *value* to a null-terminated string in an array

The **ecvt** function converts *value* to a null-terminated string of *ndigit* digits and returns a pointer thereto. The high-order digit is non-zero, unless the value is zero. The low-order digit is rounded up if its value is 5 or greater and rounded down if it is less than 5. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). The decimal point is not included in the returned string. If the sign of the result is negative, the word pointed to by *sign* is non-zero, otherwise it is zero.

fcvt is identical to **ecvt**, except that the correct digit has been rounded for printf "%f" (FORTRAN F-format) output of the number of digits specified by *ndigit*.

gcvt converts the *value* to a null-terminated string in the array pointed to by *buf* and returns *buf*. It attempts to produce *ndigit* significant digits in FORTRAN F-format if possible, otherwise E-format, ready for printing. A minus sign, if there is one, or a decimal point is included as part of the returned string. Trailing zeros are suppressed.

Notes

The values returned by `ecvt` and `fcvt` point to a single static data array whose content is overwritten by each call.

See also

`printf(S)`

Standards conformance

`ecvt`, `fcvt` and `gcvt` are conformant with:
Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

(S)

end, etext, edata

last locations in program

Syntax

```
extern end;  
extern etext;  
extern edata;
```

Description

end - first address above uninitialized data region

etext - first address above program text

edata - first address above initialized data region

These names refer neither to routines nor to locations with interesting contents. The address of **etext** is the first address above the program text, **edata** above the initialized data region, and **end** above the uninitialized data region.

When execution begins, the program break (the first location beyond the data) coincides with **end**, but the program break may be reset by the routines of **brk(S)**, **malloc(S)**, standard input/output **stdio(S)**, the profile (-p) option of **cc(CP)**, and so on. Thus, the current value of the program break should be determined by **sbrk ((char *)0)**. See **brk(S)**.

See also

brk(S), **cc(CP)**, **malloc(S)**, **stdio(S)**

Standards conformance

end, **etext** and **edata** are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

erf, erfc

error function and complementary error function

Syntax

```
cc ... -lm

#include <math.h>

double erf (x)
double x;

double erfc (x)
double x;
```

Description

The **erf** function returns the error function of x . This is defined as:

$$\pm \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

erfc, which returns 1.0 minus **erf(x)**, is provided because of the extreme loss of relative accuracy if **erf(x)** is called for a large x and the result subtracted from 1.0 (for example, for $x = 5$, 12 places are lost).

See also

exp(S)

Standards conformance

erf and **erfc** are conformant with:

AT&T SVID Issue 2;
and X/Open Portability Guide, Issue 3, 1989.

ev_block

wait until the queue contains an event

Syntax

```
cc ... -levent

#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/page.h>
#include <sys/event.h>
#include <mouse.h>

int ev_block()
```

Description

After a process has opened an event queue with **ev_init** and **ev_open**, **ev_block** causes the process to sleep until there is an event in the event queue.

Diagnostics

A call to **ev_block** returns -1 if the process does not have an open event queue, or if it is interrupted. It returns zero if it succeeds.

See also

ev_close(S), **ev_count(S)**, **ev_flush(S)**, **ev_getdev(S)**, **ev_getemask(S)**, **ev_gindev(S)**, **ev_init(S)**, **ev_open(S)**, **ev_pop(S)**, **ev_read(S)**, **ev_resume(S)**, **ev_setemask(S)**, **ev_suspend(S)**

Standards conformance

ev_block is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

ev_close

close the event queue and all associated devices

Syntax

```
cc ... -levent

#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/page.h>
#include <sys/event.h>
#include <mouse.h>

int ev_close()
```

Description

ev_close closes the event queue and any event devices currently open. This call takes no arguments.

An event queue must have been opened previously with **ev_init** and **ev_open**.

Diagnostics

This routine returns a negative number to indicate an error. Making this call before obtaining an open event queue is an example of such an error.

See also

ev_block(S), **ev_count(S)**, **ev_flush(S)**, **ev_getdev(S)**, **ev_getemask(S)**, **ev_gindev(S)**, **ev_init(S)**, **ev_open(S)**, **ev_pop(S)**, **ev_read(S)**, **ev_resume(S)**, **ev_setemask(S)**, **ev_suspend(S)**

Standards conformance

ev_close is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

ev_count

returns the number of events currently in the queue

Syntax

```
cc ... -levent

#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/page.h>
#include <sys/event.h>
#include <mouse.h>

int ev_count( )
```

Description

ev_count Returns the number of events currently in the queue. An event queue must have been opened with **ev_init** and **ev_open**.

Diagnostics

ev_count returns -1 if there is not an open event queue.

See also

ev_block(S), **ev_close(S)**, **ev_flush(S)**, **ev_getdev(S)**, **ev_getemask(S)**,
ev_gindev(S), **ev_init(S)**, **ev_open(S)**, **ev_pop(S)**, **ev_read(S)**, **ev_resume(S)**,
ev_setemask(S), **ev_suspend(S)**

Standards conformance

ev_count is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

ev_flush

discard all events currently in the queue

Syntax

```
cc ... -levent

#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/page.h>
#include <sys/event.h>
#include <mouse.h>

int ev_flush()
```

Description

ev_flush discards all events currently in the queue. Events in the queue when **ev_flush** is invoked are not further available to the program.

Diagnostics

ev_flush returns -1 if there is no open event queue. Normally it returns zero.

See also

ev_block(S), **ev_close(S)**, **ev_count(S)**, **ev_getdev(S)**, **ev_getemask(S)**, **ev_gindev(S)**, **ev_init(S)**, **ev_open(S)**, **ev_pop(S)**, **ev_read(S)**, **ev_resume(S)**, **ev_setemask(S)**, **ev_suspend(S)**

Standards conformance

ev_flush is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

ev_getdev

gets a list of devices feeding an event queue

Syntax

cc ... -levent

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/page.h>
#include <sys/event.h>
#include <mouse.h>

struct devinfo * ev_getdev (dev_mask, devinfo)
dmask_t dev_mask;
struct devinfo *devinfo;
```

Description

ev_getdev allows a program to examine the devices that are attached to its event queue. An open event queue must have been previously obtained with **ev_init** and **ev_open**. This routine takes two arguments, a bitmask of device classes and a pointer to a **device_info** structure. The device mask indicates the classes of devices in which the program is interested. The device pointer is used to cycle through the devices attached to the queue.

The device mask is made by OR'ing together a subset of **D_REL**, **D_ABS**, **D_STRING1** and **D_OTHER**. These values represent classes of graphics input devices. **D_REL** refers to relative locator devices like mice. **D_ABS** refers to absolute locator devices like bitpads and lightpens. **D_STRING** refers to character stream devices like the keyboard.

The device pointer parameter is **NULL** for the first call. Each call returns a pointer which should be passed in in subsequent calls. When the routine has iterated through all the devices attached to the queue, it returns **NULL**.

The device information pointer points to a structure which looks like this:

```
struct devinfo {
    short  handle;          /* not used by application */
    short  class;           /* REL, ABS, STRING or OTHER */
    short  type;            /* The type of hardware */
    char   *name;           /* Device name, from data files */
    char   *key;            /* Device key, from data files */
    short  buttons;
};
```

An application can examine this information and decide whether or not to use the device. The `ev_gindev` routine allows a program to exclude or later re-include a device. The pointer returned by `ev_getdev` is passed in to `ev_gindev`.

When a queue is opened, a bitmask specifying what kinds of devices to attach is supplied. All devices of a class which is masked in are attached to the queue. This routine is used to examine those devices.

Diagnostics

This routine returns -1 to a program which does not have an open event queue. It returns -2 if no devices of any class which is masked in are found. Normally it returns zero.

See also

`ev_block(S)`, `ev_close(S)`, `ev_count(S)`, `ev_flush(S)`, `ev_getemask(S)`, `ev_gindev(S)`, `ev_init(S)`, `ev_open(S)`, `ev_pop(S)`, `ev_read(S)`, `ev_resume(S)`, `ev_setemask(S)`, `ev_suspend(S)`

Notes

The keyboard is attached to an event queue whenever devices of class `D_STRING` are requested. If the keyboard is attached to an event queue, then the keyboard does not generate normal *stdin* input until the event queue is closed.

Standards conformance

`ev_getdev` is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

ev_getemask

return the current event mask

Syntax

```
cc ... -levent

#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/page.h>
#include <sys/event.h>
#include <mouse.h>

int ev_getemask(emaskp)
emask_t *emaskp;
```

Description

ev_getemask returns the current event mask. This call takes a pointer to an event mask which is filled in. The program must have already opened an event queue.

This call complements **ev_setemask**. The manual page for **ev_setemask** describes an event mask in detail.

Diagnostics

ev_getemask returns -1 if there is no open event queue. Otherwise it returns 0.

See also

ev_block(S), **ev_close(S)**, **ev_count(S)**, **ev_flush(S)**, **ev_getdev(S)**,
ev_gindev(S), **ev_init(S)**, **ev_open(S)**, **ev_pop(S)**, **ev_read(S)**, **ev_resume(S)**,
ev_setemask(S), **ev_suspend(S)**

Standards conformance

ev_getemask is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

ev_gindev

include/exclude devices for event input

Syntax

```
cc ... -levent

#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/page.h>
#include <sys/event.h>
#include <mouse.h>

int ev_gindev(devinfo, action)
struct devinfo *devinfo;
char action;
```

Description

ev_gindev is used in concert with **ev_getdev** to exclude or later re-include devices from feeding the event queue. The argument *devinfo* contains the pointer to the device to be included or excluded. The arguments are a pointer to the device and **EXCLUDE**. The pointer is obtained through the **getdev** function.

EXCLUDE is defined in *<mouse.h>*.

Diagnostics

This routine returns 0 if it succeeds. It returns -1 if there is no active event queue. It returns -2 if the *devinfo* argument does not point to a valid device. It returns -3 on attempts to exclude an excluded device, or attempts to reinclude an included device. It returns -4 if the action argument is invalid.

See also

ev_block(S), **ev_close(S)**, **ev_count(S)**, **ev_flush(S)**, **ev_getdev(S)**, **ev_getemask(S)**, **ev_init(S)**, **ev_open(S)**, **ev_pop(S)**, **ev_read(S)**, **ev_resume(S)**, **ev_setemask(S)**, **ev_suspend(S)**

Standards conformance

ev_gindev is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

ev_init

invokes the event manager

Syntax

```
cc ... -levent

#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/page.h>
#include <sys/event.h>
#include <mouse.h>

int ev_init()

int ev_initf (fd)
int fd;
```

Description

ev_init reads the system event-configuration files and initializes the event manager.

ev_initf reads the system event-configuration files and initializes the event manager, using a file descriptor (*fd*) as argument to specify where the event queue should be attached. The file descriptor (*fd*) for **ev_init** is always 0.

This is the first of two steps that a program follows to obtain an event queue. Devices such as mice or the keyboard may be read through an event queue. When **ev_init** or **ev_initf** is called, the configuration files are read and checked for syntax. If there is an error or inconsistency, **ev_init** and **ev_initf** return an error. After the event manager is initialized, **ev_open(S)** should be called to obtain an event queue.

Diagnostics

ev_init and **ev_initf** return 0 if they succeed in reading the data files and initializing an event queue. Otherwise they return -1.

See also

ev_block(S), ev_close(S), ev_count(S), ev_flush(S), ev_getdev(S), ev_getemask(S), ev_gindev(S), ev_open(S), ev_pop(S), ev_read(S), ev_resume(S), ev_setemask(S), ev_suspend(S)

Files

/usr/lib/event/devices
/usr/lib/event/ttys

Standards conformance

ev_init and *ev_initf* are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

(S)

ev_open

opens an event queue for input

Syntax

```
cc ... -levent

#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/page.h>
#include <sys/event.h>
#include <mouse.h>

int ev_open (dmask)
dmask_t *dmaskp;
```

Description

ev_open opens an event queue for input. The argument points to a bitmask of device types. **ev_open** attaches available devices whose class is masked in. **ev_open** fills in the mask to indicate what kinds of devices it finds.

The bitmask is made of one or more of four classes of devices. The four classes are **D_STRING**, **D_REL**, **D_ABS**, or **D_OTHER**. **D_STRING** refers to character stream devices like the keyboard. **D_REL** refers to relative locator devices like mice. **D_ABS** refers to absolute locator devices like bitpads. These values are defined in *<mouse.h>*.

ev_open attempts to open devices of the types indicated in the argument and sets the mask to indicate the devices successfully opened. If no devices can be successfully opened, **ev_open** returns -1 as an error condition.

If **ev_open** succeeds in opening an event queue and devices, it returns a file descriptor for the event queue. The file descriptor is for use with the select system call and should not be used for reading or writing.

This is a program fragment that opens an event queue with a mouse and the keyboard attached:

```
main()
{
    dmask_t dmask;           /* device mask */
    int qfd;                 /* event queue file descriptor */

    ev_init();               /* initialize event manager */

    dmask = D_REL|D_STRING;   /* device mask for mouse & kbd */
    qfd = ev_open(&dmask);    /* try to open event queue */
    if ( qfd < 0 )
        exit(1);             /* error on open */

    if (dmask != (D_REL | D_STRING))
        exit(2);             /* could not attach both devices */
                               /* event queue is open */
}
```

Diagnostics

The routine returns a negative number if it fails.

It returns -1 if there was a configuration error in the configuration files (see `ev_init(S)`). `ev_open` returns -2 if it does not find any devices to attach. It returns -3 if it is unable to open devices it finds. It returns -4 if it is unable to open an event queue.

See also

`ev_block(S)`, `ev_close(S)`, `ev_count(S)`, `ev_flush(S)`, `ev_getdev(S)`,
`ev_getemask(S)`, `ev_gindev(S)`, `ev_init(S)`, `ev_pop(S)`, `ev_read(S)`,
`ev_resume(S)`, `ev_setemask(S)`, `ev_suspend(S)`

Standards conformance

`ev_open` is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

ev_pop

pop the next event off the queue

Syntax

```
cc ... -levent

#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/page.h>
#include <sys/event.h>
#include <mouse.h>

int ev_pop()
```

Description

ev_pop clears the next event off the queue and returns the number of events that were lost due to queue overrun since the last **ev_pop** call. An event queue must have been opened with **ev_init** and **ev_open**.

After an application is done with an event, the event is popped off the queue. The queue is of fixed size, so if events are not popped fast enough some might be lost due to overrun. A counter maintains the number of lost events. When **ev_pop** is called, it clears the top event off of the queue, clears the counter and returns the number of lost events. This should always be zero, unless a program stops reading its event queue. If the queue is empty, **ev_pop** returns -1.

When **ev_pop** is called, the most recent pointer returned by **ev_read** must be considered invalid, since that storage may be overwritten by the event driver.

Diagnostics

ev_pop returns -1 if there is not an open event queue. It returns -2 if there is nothing to pop because the queue is empty.

See also

ev_block(S), **ev_close(S)**, **ev_count(S)**, **ev_flush(S)**, **ev_getdev(S)**,
ev_getemask(S), **ev_gindev(S)**, **ev_init(S)**, **ev_open(S)**, **ev_read(S)**,
ev_resume(S), **ev_setemask(S)**, **ev_suspend(S)**

Standards conformance

ev_pop is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

ev_read

read the next event in the queue

Syntax

```
cc ... -levent

#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/page.h>
#include <sys/event.h>
#include <mouse.h>

EVENT* ev_read ( )
```

Description

ev_read returns a pointer to the next event in the queue or **NULL** if the queue is empty. Multiple calls to this routine return the same pointer until **ev_pop** is called.

An error occurs if this routine is called from a program without an open event queue.

Diagnostics

This routine returns **NULL** if there is no event to read or if there is not an open event queue.

See also

ev_block(S), **ev_close(S)**, **ev_count(S)**, **ev_flush(S)**, **ev_getdev(S)**, **ev_getemask(S)**, **ev_gindev(S)**, **ev_init(S)**, **ev_open(S)**, **ev_pop(S)**, **ev_resume(S)**, **ev_setemask(S)**, **ev_suspend(S)**

Standards conformance

ev_read is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

ev_resume

restart a suspended queue

Syntax

```
cc ... -levent

#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/page.h>
#include <sys/event.h>
#include <mouse.h>

int ev_resume ( )
```

Description

ev_resume restarts an event queue suspended by an **ev_suspend** call.

Diagnostics

This routine returns -1 to a program which does not have an open event queue. It returns -2 if the queue is not suspended. Normally it returns 0.

See also

**ev_block(S), ev_close(S), ev_count(S), ev_flush(S), ev_getdev(S),
ev_getemask(S), ev_gindev(S), ev_init(S), ev_open(S), ev_pop(S), ev_read(S),
ev_setemask(S), ev_suspend(S)**

Standards conformance

ev_resume is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

ev_setemask

sets event mask

Syntax

```
cc ... -levent

#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/page.h>
#include <sys/event.h>
#include <mouse.h>

int ev_setemask (emask)
emask_t emask;
```

Description

ev_setemask sets the event mask on an event queue. Events whose tag is not masked-in are prevented from entering an event queue. Event masks are always initialized to allow all events.

The different types of events are:

- R Reserved
- a "Other" Device events
- b Button events
- c String events
- d Relative Locator movement events
- e Absolute Locator movement events

The bits that make up the mask number have the following definitions:

R	R	R	R	R	R	R	R	R	R	R	e	d	c	b	a

15

0

Diagnostics

If there is no open event queue, -1 is returned. If the new event mask would cause no events to enter the queue, -2 is returned and the event mask is not changed. For example, on a queue with only a mouse, any event mask which did not include `D_REL` would not allow any events to be entered into the queue.

See also

`ev_block(S)`, `ev_close(S)`, `ev_count(S)`, `ev_flush(S)`, `ev_getdev(S)`,
`ev_getemask(S)`, `ev_gindev(S)`, `ev_init(S)`, `ev_open(S)`, `ev_pop(S)`, `ev_read(S)`,
`ev_resume(S)`, `ev_suspend(S)`

Standards conformance

`ev_stemsk` is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

ev_suspend

suspends an event queue.

Syntax

```
cc ... -levent

#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/page.h>
#include <sys/event.h>
#include <mouse.h>

int ev_suspend ( )
```

Description

ev_suspend suspends an event queue from receiving input. For example, if an application wants to fork a subshell, a call to **ev_suspend** suspends events until the subshell returns and the queue is resumed with an **ev_resume** call. That way a process in the subshell can also have an event queue. This is required because the event manager only allows one active event queue per terminal or multiscreen.

Diagnostics

This function returns -1 if no event queue is opened. It returns -2 if the queue is already suspended. Normally it returns zero.

See also

ev_block(S), **ev_close(S)**, **ev_count(S)**, **ev_flush(S)**, **ev_getdev(S)**, **ev_getemask(S)**, **ev_gindex(S)**, **ev_init(S)**, **ev_open(S)**, **ev_pop(S)**, **ev_read(S)**, **ev_resume(S)**, **ev_setemask(S)**

Standards conformance

ev_suspend is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

exec: execl, execv, execl, execve, execlp, execvp

execute a file

Syntax

```
cc ... -lc

int execl (path, arg0, arg1, ..., argn, (char *)0)
char *path, *arg0, *arg1, ..., *argn;

int execl (path, arg0, arg1, ..., argn, (char *)0, envp)
char *path, *arg0, *arg1, ..., *argn, *envp[ ];

int execlp (file, arg0, arg1, ..., argn, (char *)0)
char *file, *arg0, *arg1, ..., *argn;

int execv (path, argv)
char *path, *argv[];

int execve (path, argv, envp)
char *path, *argv[], *envp[];

int execvp (file, argv)
char *file, *argv[];
```

Description

The `exec` system call in all its forms transforms the calling process into a new process. The new process is constructed from an ordinary, executable *file*. Executable files consist of a header (see *a.out(FP)*), a text segment, and a data segment. The data segment contains an initialized portion and an uninitialized portion (bss). There can be no return from a successful `exec` because the calling process is overlaid by the new process.

Executables created by C programs always use the function `main` as their entry point. The prototype for `main` is (arguments are optional):

```
main(argc, argv, envp)
int argc;
char **argv, **envp;
```

where *argc* is the argument count, *argv* is an array of character pointers to the arguments themselves, and *envp* is an array of character pointers to the environment strings. As indicated, *argc* is conventionally at least one, and the first member of the array points to a string containing the name of the file.

The *path* argument points to a path name that identifies the new process file.

The *file* argument points to the new process file. The path prefix for this file is obtained by a search of the directories passed as the *environment* line "PATH="; see *environ*(M). The environment is supplied by the shell; see *sh*(C).

arg0, *arg1*, ..., *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, at least *arg0* must be present and point to a string that is the same as *path* (or its last component).

argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process. By convention, *argv* must have at least one member, and it must point to a string that is the same as *path* (or its last component). *argv* is terminated by a null pointer.

envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process. *envp* is terminated by a null pointer. For *execl* and *execv*, the C run-time start-off routine places a pointer to the environment of the calling process in the global cell:

```
extern char **environ;
```

and it is used to pass the environment of the calling process to the new process.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set; see *fcntl*(S). For those file descriptors that remain open, the file pointer is unchanged.

Signals set to terminate the calling process are set to terminate the new process. Signals set to be ignored by the calling process are set to be ignored by the new process. Signals set to be caught by the calling process are set to terminate the new process; see *signal*(S).

For signals set by *sigset*, *exec* ensures that the new process has the same system signal action for each signal type whose action is *SIG_DFL*, *SIG_IGN*, or *SIG_HOLD* as the calling process. However, if the action is to catch the signal, then the action is reset to *SIG_DFL*, and any pending signal for this type is held.

If the set-user-ID mode bit of the new process file is set (see *chmod*(S)), *exec* sets the effective user ID of the new process to the owner ID of the new process file. Similarly, if the set-group-ID mode bit of the new process file is set, the effective group ID of the new process is set to the group ID of the new process file. The real user ID and real group ID of the new process remain the same as those of the calling process.

The shared memory segments attached to the calling process are not attached to the new process; see *shmop*(S).

Profiling is disabled for the new process; see *profil*(S).

The new process also inherits the following attributes from the calling process:

- nice value; see **nice(S)**
- process ID
- parent process ID
- process group ID
- **semadj** values; see **semop(S)**
- tty group ID; see **exit(S)** and **signal(S)**
- trace flag; see **ptrace(S)** request 0
- time left until an alarm clock signal; see **alarm(S)**
- current working directory
- root directory
- file mode creation mask; see **umask(S)**
- file size limit; see **ulimit(S)**
- **utime**, **stime**, **cutime**, and **cstime**; see **times(S)**
- file-locks; see **fcntl(S)** and **lockf(S)**

The **exec** system call fails and returns to the calling process if one or more of the following is true:

[E2BIG]	The number of bytes in the new process's argument list is greater than the system-imposed limit of 5120 bytes.
[EACCES]	• Search permission is denied for a directory listed in the new process file's path prefix. • The new process file is not an ordinary file. • The new process file mode denies execution permission.
[EAGAIN]	Not enough memory.
[EFAULT]	• Required hardware is not present. • <i>path</i>, <i>argv</i>, or <i>envp</i> point to an illegal address.
[EINTR]	A signal was caught during the exec system call.
[ELIBACC]	Required shared library does not have execute permission.
[ELIBEXEC]	Trying to exec a shared library directly.
[EMULTIHOP]	Components of <i>path</i> require hopping to multiple remote machines.
[ENOENT]	One or more components of the new process path name of the file do not exist.
[ENOEXEC]	The exec is not an execvp or execvp , and the new process file has the appropriate access permission but an invalid magic number in its header.

[ENOLINK]	<i>path</i> points to a remote machine and the link to that machine is no longer active.
[ENOMEM]	The new process requires more memory than is allowed by the system-imposed maximum MAXMEM .
[ENOTDIR]	A component of the new process path of the file prefix is not a directory.
[ETXTBSY]	The new process file is a pure procedure (shared text) file that is currently open for writing by some process.

Notes

Security services provided by the operating system provide the following:

- Executing a `setuid/setgid` file without a valid login UID fails.
- The login UID is inherited from the parent process.

See also

`a.out(FP)`, `alarm(S)`, `environ(M)`, `exit(S)`, `fcntl(S)`, `fork(S)`, `lockf(S)`, `nice(S)`, `ptrace(S)`, `semop(S)`, `sh(C)`, `signal(S)`, `sigset(S)`, `times(S)`, `ulimit(S)`, `umask(S)`

Diagnostics

If `exec` returns to the calling process, an error has occurred; the return value is -1 and `errno` is set to indicate the error.

Standards conformance

`execl`, `execle`, `execlp`, `execv`, `execve` and `execvp` are conformant with:

AT&T SVID Issue 2;
 X/Open Portability Guide, Issue 3, 1989;
 IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
 and NIST FIPS 151-1.

execseg

makes a data region executable

Syntax

```
cc ... -lx

#include <xdata.h>

excode_t execseg(oldaddr, size)
exdata_t oldaddr;
unsigned size;

int unexecseg(addr)
excode_t addr;
```

Description

execseg is passed the current data address and size of the region to be executed and it returns the starting address of a region that is at least *size* number of bytes which can safely be branched to. On the Intel 8086 and 80286 processor, an alias CS descriptor is associated with the same memory as the data segment in which the *oldaddr* region lies. This means that offsets in the executable segment to access a given byte are essentially the same as the offsets in the original data segment, except the selector is different.

Note that **excode_t** and **exdata_t** are “far” pointers on the 8086 and 80286 and segment selectors on the 386. On an architecture where pages in the same “segment” are any combination of read/write/execute, the returned address is identical to the parameter passed to **execseg**.

We recommend that programs using this function on 8086- and 80286-based processors be large model, or that programmers be very familiar with “hybrid model” as well as with the use and misuse of far data.

When an error occurs, **execseg** returns **((excode_t)-1)**, with **errno** set to **ENONEM**. Errors include an invalid data address or *size*, and an inability to allocate a new data selector.

The **unexecseg** system call disables an *addr* previously returned from **execseg** from being used as an executable region. Specifically, on the 8086 and 80286 architectures, this call frees the pointer used for the executable region. It returns 0 on success, or a -1 on error. For example, if *addr* is not an address returned by **execseg**, then a -1 is returned and it can be used as an executable region.

Example

```

excode_t funcp; char far *datap;
.
.
.
datap=brkctl(BR_NEWSEG,1000L,0L);

load_with_code(datap,1000);          /* load executable code into */
                                   /* data region datap */

funcp=execseg(datap,1000); (*funcp)();

if (unexecseg (funcp)==-1){          /*call subroutine*/
    printf("unexecseg failed\n"); exit(1); }

```

Notes

On the Intel 8086 and 80286 architectures, `execseg` expects far addresses to be passed. Only experienced programmers should use this feature.

Since the `execseg` return value and address arguments are "far" pointers on 86 and 286 machines, any program including `xdata.h` must be compiled using the `-Me` option.

The following restrictions apply to the execute data system call.

1. Even though an address and size are passed to `execseg`, the entire segment containing the requested addresses are aliased. The address and size are validated before the aliasing is allowed.
2. No part of the data segment that is aliased may be deallocated (via `sbrk(S)` or `brkctl(S)`) while it is aliased. This restriction applies to the entire segment that is aliased, even if only a small piece of the segment was aliased. After `unexecseg` the aliased segment, the data segment may be deallocated.
3. Each call to `execseg` results in a new alias segment being used, even if the data segment is already aliased.

Due to compiler confusion, you may get the message "at least one void operand" when using `execseg`. Please ignore it.

Under UNIX-386 the `oldaddr` parameter passed to `execseg` is a segment selector, not a pointer. `execseg` returns a selector value which provides a code segment alias to the original segment.

Similarly, the `addr` parameter passed to `unexecseg` is a selector value that has previously been returned by `execseg`.

Note that since `execseg` returns a segment selector on 386 machines, the return value from `execseg` is not directly useable in a C program. In an assembler program this selector value would be used as the top 16 bits of a 48 bit far pointer.

See also

`brkctl(S)`, `sbrk(S)`

Standards conformance

`execseg` and `unexecseg` are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

exit, _exit

terminate process

Syntax

```
cc ... -lc

#include <stdlib.h>

void exit (status)
int status;

void _exit (status)
int status;
```

Description

exit - terminates calling process

_exit - circumvents cleanup

The **exit** system call terminates the calling process with the following consequences:

All of the file descriptors open in the calling process are closed.

If the parent process of the calling process is executing a **wait**, it is notified of the calling process's termination and the low order eight bits (that is, bits 0377) of *status* are made available to it (see **wait(S)**).

If the parent process of the calling process is not executing a **wait**, the calling process is transformed into a *zombie process*. A zombie process is a process that only occupies a slot in the process table. It has no other space allocated either in user or kernel space. The process table slot that it occupies is partially overlaid with time accounting information (see *<sys/proc.h>*) to be used by **times**.

The parent process ID of all of the calling processes' existing child processes and zombie processes is set to 1. This means the initialization process (see **Intro(S)**) inherits each of these processes.

Each attached shared memory segment is detached and the value of **shm_nattach** in the data structure associated with its shared memory identifier is decremented by 1.

For each semaphore for which the calling process has set a **semadj** value (see **semop(S)**), that **semadj** value is added to the **semval** of the specified semaphore.

If the process has a process, text, or data lock, an **unlock** is performed (see **plock(S)**).

An accounting record is written on the accounting file if the system's accounting routine is enabled (see **acct(S)**).

If the process ID, tty group ID, and process group ID of the calling process are equal, the **SIGHUP** signal is sent to each process that has a process group ID equal to that of the calling process.

A death of child signal is sent to the parent.

The C function **exit** may cause cleanup actions before the process exits. The function **_exit** circumvents all cleanup.

See also

acct(S), **Intro(S)**, **plock(S)**, **semop(S)**, **signal(S)**, **sigset(S)**, **wait(S)**

Diagnostics

None. There can be no return from an **exit** system call.

Standards conformance

_exit is conformant with:

X/Open Portability Guide, Issue 3, 1989;
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

exit is conformant with:

X/Open Portability Guide, Issue 3, 1989;
ANSI X3.159-1989 Programming Language — C;
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

exp, log, log10, pow, sqrt

exponential, logarithm, power, square root functions

Syntax

```
cc ... -lm

#include <math.h>

double exp (x)
double x;

double log (x)
double x;

double log10 (x)
double x;

double pow (x, y)
double x, y;

double sqrt (x)
double x;
```

Description

The **exp** function returns e^x .

The **log** function returns the natural logarithm of x . The value of x must be positive.

The **log10** function returns the logarithm base ten of x . The value of x must be positive.

The **pow** function returns x^y . If x is zero, y must be positive. If x is negative, y must be an integer.

The **sqrt** function returns the non-negative square root of x . The value of x cannot be negative.

See also

hypot(S), matherr(S), sinh(S)

Diagnostics

The **exp** function returns **HUGE** when the correct value would overflow, or 0 when the correct value would underflow, and sets **errno** to **ERANGE**.

The **log** and **log10** functions return **-HUGE** and set **errno** to **EDOM** when x is non-positive. A message indicating **DOMAIN** error (or **SING** error when x is 0) is printed on the standard error output.

The **pow** function returns 0 and sets **errno** to **EDOM** when x is 0 and y is non-positive, or when x is negative and y is not an integer. In these cases a message indicating **DOMAIN** error is printed on the standard error output. When the correct value for **pow** would overflow or underflow, **pow** returns $\pm$ **HUGE** or 0 respectively, and sets **errno** to **ERANGE**.

The **sqrt** returns 0 and sets **errno** to **EDOM** when x is negative. A message indicating **DOMAIN** error is printed on the standard error output.

These error-handling procedures may be changed with the function **matherr**.

Standards conformance

exp is conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1); and NIST FIPS 151-1.

log, **log10**, **pow** and **sqrt** are conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
ANSI X3.159-1989 Programming Language — C;
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

fclose, fflush

close or flush a stream

Syntax

```
cc ... -lc

#include <stdio.h>

int fclose (stream)
FILE *stream;

int fflush (stream)
FILE *stream;
```

Description

The **fclose** function causes any buffered data for the named *stream* to be written out, and the *stream* to be closed.

The **fclose** function is performed automatically for all open files upon calling **exit**.

fflush causes any buffered data for the named *stream* to be written to that file. The *stream* remains open.

Diagnostics

These functions return 0 for success and EOF if any error (such as trying to write to a file that has not been opened for writing) was detected.

See also

close(S), **exit(S)**, **fopen(S)**, **setbuf(S)**, **stdio(S)**

Standards conformance

fclose and **fflush** are conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

ANSI X3.159-1989 Programming Language — C;

Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

fcntl

file control

Syntax

```
cc ... -lc

#include <fcntl.h>

int fcntl (fildes, cmd, arg)
int fildes, cmd;
```

Description

The **fcntl** system call provides for control over open files. The *fildes* argument is an open file descriptor obtained from a **creat**, **open**, **dup**, **fcntl**, or **pipe** system call. The data type and value of *arg* are specific to the type of command specified by *cmd*. The symbolic names for commands and file status flags are defined by the *<fcntl.h>* header file.

The commands available are:

F_DUPFD	Return a new file descriptor as follows: • Lowest numbered available file descriptor greater than or equal to <i>arg</i>. • Same open file (or pipe) as the original file. • Same file pointer as the original file (that is, both file descriptors share one file pointer). • Same access mode (read, write, or read/write). • Same file status flags (that is, both file descriptors share the same file status flags). • The close-on-exec flag associated with the new file descriptor is set to remain open across exec(S) system calls.
F_GETFD	Get the close-on-exec flag associated with the file descriptor <i>fildes</i> . If the low-order bit is 0, the file remains open across exec ; otherwise the file is closed upon execution of exec .
F_GETFL	Get <i>file</i> status flags (see open(S)).
F_SETFD	Set the close-on-exec flag associated with <i>fildes</i> to the low-order bit of <i>arg</i> (0 or 1 as above).
F_SETFL	Set <i>file</i> status flags to <i>arg</i> . Only certain flags can be set; see fcntl(M) .

The following commands are used for file-locking and record-locking. Locks may be placed on an entire file or segments of a file.

- F_GETLK** Get the first lock that blocks the lock description given by the variable of type `struct flock` pointed to by *arg*. The information retrieved overwrites the information passed to `fcntl` in the `flock` structure. If no lock is found that would prevent this lock from being created, then the structure is passed back unchanged except for the lock type which is set to `F_UNLCK`.
- F_SETLK** Set or clear a file segment lock. According to the variable of type `struct flock` pointed to by *arg* (see `fcntl(M)`), the *cmd* `F_SETLK` is used to establish read (`F_RDLCK`) and write (`F_WRLCK`) locks. It also is used to remove either type of lock (`F_UNLCK`). If a read or write lock cannot be set, `fcntl` returns immediately with an error value of -1.
- F_SETLKW** This *cmd* is the same as `F_SETLK` except that if a read or write lock is blocked by other locks, the process sleeps until the segment is free to be locked.

A read lock prevents any process from write locking the protected area. More than one read lock may exist for a given segment of a file at a given time. The file descriptor on which a read lock is being placed must have been opened with read access.

A write lock prevents any process from read-locking or write-locking the protected area. Only one write lock may exist for a given segment of a file at a given time. The file descriptor on which a write lock is being placed must have been opened with write access.

The structure `flock` defined in the `<fcntl.h>` header file describes a lock. It describes the type (*l_type*), starting offset (*l_whence*), relative offset (*l_start*), size (*l_len*), and process-ID (*l_pid*):

```
short l_type;    /* F_RDLCK, F_WRLCK, F_UNLCK */
short l_whence; /* flag for starting offset */
long l_start;    /* relative offset in bytes */
long l_len;      /* if 0 then until EOF */
short l_pid;     /* returned with F_GETLK */
```

The value of *l_whence* is 0, 1, or 2 to indicate that the relative offset, *l_start* bytes, is measured from the start of the file, current position, or end of file, respectively. The value of *l_len* is the number of consecutive bytes to be locked. The process id is used only with the `F_GETLK` *cmd* to return the values for a blocking lock. Locks may start and extend beyond the current end of a file, but may not be negative relative to the beginning of the file. A lock may be set to always extend to the end of file by setting *l_len* to zero (0). If such a lock also has *l_whence* and *l_start* set to zero (0), the whole file is locked. Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments for either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not

inherited by a child process in a fork system call.

When mandatory file and record locking is active on a file (see `chmod(S)`), read and write system calls issued on the file are affected by the record locks in effect.

The `fcntl` system call fails if one or more of the following is true:

- [EACCES] The *cmd* argument is `F_SETLK`, the type of lock (*l_type*) is a read (`F_RDLCK`) lock, and the segment of a file to be locked is already write locked by another process or the type is a write (`F_WRLCK`) lock and the segment of a file to be locked is already read or write locked by another process.
- [EBADF]
 1. The *fil-des* argument is not a valid open file descriptor.
 2. The *cmd* argument is `F_SETLK` or `F_SETLKW`, the type of lock (*l_type*) is a read-lock (`F_RDLCK`), and *fil-des* is not a valid file-descriptor open for reading.
 3. The *cmd* argument is `F_SETLK` or `F_SETLKW`, the type of lock (*l_type*) is a write-lock (`F_WRLCK`), and *fil-des* is not a valid file-descriptor open for writing.
- [EDEADLK] The *cmd* argument is `F_SETLKW`, the lock is blocked by some lock from another process, and putting the calling-process to sleep, waiting for that lock to become free, would cause a deadlock.
- [EFAULT] The *cmd* argument is `F_SETLK`, *arg* points outside the program address space.
- [EINTR] A signal was caught during the `fcntl` system call.
- [EINVAL]
 1. The *cmd* argument is `F_DUPFD`. The *arg* argument is either negative, or greater than or equal to the configured value for the maximum number of open file descriptors allowed each user.
 2. The *cmd* argument is `F_GETLK`, `F_SETLK`, or `SETLKW` and *arg* or the data it points to is not valid.
- [EMFILE] The *cmd* argument is `F_DUPFD` and file-descriptors are currently open in the calling-process.
- [ENOLCK] The *cmd* argument is `F_SETLK` or `F_SETLKW`, the type of lock is a read or write lock, and there are no more record locks available (too many file segments locked) because the system maximum has been exceeded.
- [ENOLINK] *fil-des* is on a remote machine and the link to that machine is no longer active.

Diagnostics

Upon successful completion, the value returned depends on *cmd* as follows:

<code>F_DUPFD</code>	A new file descriptor.
<code>F_GETFD</code>	Value of flag (only the low-order bit is defined).
<code>F_GETFL</code>	Value of file flags.
<code>F_GETLK</code>	Value other than -1.
<code>F_SETFD</code>	Value other than -1.
<code>F_SETFL</code>	Value other than -1.
<code>F_SETLK</code>	Value other than -1.
<code>F_SETLKW</code>	Value other than -1.

Otherwise, a value of -1 is returned, and `errno` is set to indicate the error.

Warnings

Because in the future the variable `errno` will be set to `EAGAIN` rather than `EACCES` when a section of a file is already locked by another process, portable application programs should expect and test for either value.

See also

`close(S)`, `creat(S)`, `dup(S)`, `exec(S)`, `fcntl(M)`, `fork(S)`, `open(S)`, `pipe(S)`

Standards conformance

`fcntl` is conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

ferror, feof, clearerr, fileno

stream status inquiries

Syntax

```
#include <stdio.h>

int ferror (stream)
FILE *stream;

int feof (stream)
FILE *stream;

void clearerr (stream)
FILE *stream;

int fileno (stream)
FILE *stream;
```

Description

The **ferror** function returns non-zero when an I/O error occurred reading from or writing to the named *stream*, otherwise zero.

feof returns non-zero when EOF previously read the named input *stream*, otherwise zero.

clearerr resets the error indicator and EOF indicator to zero on the named *stream*.

fileno returns the integer file descriptor associated with the named *stream*; see **open(S)**.

Notes

All these functions are implemented as macros; they cannot be declared or redeclared.

See also

fopen(S), **open(S)**, **stdio(S)**

Standards conformance

clearerr, feof and ferror are conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
ANSI X3.159-1989 Programming Language — C;
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C
Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

fileno is conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C
Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

fgetpos

gets and stores the current value of a stream's file position indicator

Syntax

```
cc ... -lc

#include <stdio.h>

int fgetpos(stream, pos)
FILE *stream;
fpos_t *pos;
```

Description

The **fgetpos** function gets the current value of *stream's* file position indicator and stores it in the object to which *pos* points. The **fsetpos** function can later use information stored in *pos* to reset *stream's* pointer to its position when **fgetpos** was called.

Notes

The *pos* value is stored in an internal format and is intended for use only by the **fgetpos** and **fsetpos** functions.

Return value

If successful, the **fgetpos** function returns 0. If it fails, it returns a nonzero value and sets **errno** to one of the following manifest constants (defined in *stdio.h*):

EBADF	The specified stream is not a valid file handle or is not accessible.
EINVAL	The <i>stream</i> value is invalid.

Example

```

#include <stdio.h>

FILE *stream;
fpos_t position[];
int val, i;
char fcontents[13]; /* buffer for full contents of file1 */
char writein1[14]= "Hello world!\n";
char writein2[8]= "there.\n";

main()
{
    if ((stream = fopen("file1","w+")) == NULL) /* open file1 */
    {
        perror("fopen");
        printf ("Trouble opening file\n");
    }
    else /* write in 1st string */
        fwrite(writein1,sizeof(char),13,stream);

    position=0;
    fsetpos(stream, &position); /*reset file to beginning */

    fread(fcontents, sizeof(char), 13, stream); /* read contents */
    printf("%s", fcontents); /* and print it */

    position=0;
    fsetpos(stream, &position); /*reset file to beginning */

    fread(fcontents, sizeof(char), 5, stream); /* read "Hello" */
    if (fgetpos(stream, &position) != 0) /* Save current position */
        perror("fgetpos error");

    fread(fcontents, sizeof(char), 8, stream); /* read some more */
    if (fsetpos(stream, &position) != 0) /* Return to saved position */
        perror("fsetpos error");

    /* Overwrite file from this position with the word "there" */

    fwrite(writein2, sizeof(char), 8, stream);

    position=0;
    fsetpos(stream, &position); /*reset file to beginning */
    fread(fcontents, sizeof(char), 13, stream);
    printf("%s", fcontents); /* print contents */
}

```

This program opens a file named *file1* and prints "Hello world!" It then outputs the file contents on screen. Next, it reads the first five characters in the file *hello* and calls *fgetpos* to find and save the file position pointer. After performing another read, the program calls *fsetpos* to restore the file pointer to the saved position.

See also

fsetpos(S)

Standards conformance

fgetpos is conformant with:

ANSI X3.159-1989 Programming Language — C.

field

FIELD library routines

Syntax

```
cc . . . -lform -lcurses

#include <form.h>

FIELD *dup_field (field, frow, fcol)
FIELD *field;
int frow, fcol;

char *field_arg (field)
FIELD *field;

int field_back (field)
FIELD *field;

char *field_buffer (field, buf)
FIELD *field;
int buf;

int field_fore (field)
FIELD *field;

int field_info (field, rows, cols, frow, fcol, nrow, nbuf)
FIELD *field;
int *rows, *cols, *frow, *fcol, *nrow, nbuf;

int field_just (field)
FIELD *field;

int field_opts_off (field, opts)
FIELD *field;
OPTIONS opts;

int field_opts_on (field, opts)
FIELD *field;
OPTIONS opts;

OPTIONS field_opts (field)
FIELD *field;

int field_pad (field)
FIELD *field;

int field_status (field)
FIELD *field;

FIELDTYPE *field_type (field)
FIELD *field;
```

```

char *field_userptr (field)
FIELD *field;

int free_field (field)
FIELD *field;

FIELD *link_field (field, frow, fcol)
FIELD *field;
int frow, fcol;

FIELD *new_field (r, c, frow, fcol, nrow, nbuf)
int r, c, frow, fcol, nrow, nbuf;

int move_field (field, frow, fcol)
FIELD *field;
int frow, fcol;

int set_field_back (field, back)
FIELD *field;
int back;

int set_field_buffer (field, buf, value)
FIELD *field;
int buf;
char *value;

int set_field_fore (field, fore)
FIELD *field;
int fore;

int set_field_just (field, justification)
FIELD *field;
int justification;

int set_field_opts (field, opts)
FIELD *field;
OPTIONS opts;

int set_field_pad (field, pad)
FIELD *field;
int pad;

int set_field_status (field, status)
FIELD *field;
int status;

int set_field_type (field, type, [arg_1, arg_2, . . . ] )
FIELD *field;
FIELDTYPE *type;

int set_field_userptr (field, userptr)
FIELD *field;
char *userptr;

options:

```

```

O_ACTIVE
O_AUTOSKIP
O_BLANK
O_EDIT
O_NULLOK
O_PUBLIC
O_WRAP

```

Description

These FIELD routines run on any terminal supported by **curses(S)**. Once you compile your program, sourcing the header file *form.h* with the **#include** statement, you should link it with the **form** and **curses** library routines.

Functions

The following is a list of FIELD routines:

dup_field (*field*, *frow*, *fcol*) duplicates the given field at the named location

new_field (*r*, *c*, *frow*, *fcol*, *nrow*, *nbuf*) creates a new field with *r* rows, *c* columns, starting at *frow*, *fcol* in the subwindow of the form to contain the field, with *nrow* offscreen rows and *nbuf* additional work buffers. It returns a pointer to the created field. In general, you should store these field pointers in an array.

link_field (*field*, *frow*, *fcol*) also duplicates the given field at the named location. However, it shares the field buffers between both occurrences of the field and permits the setting of different attributes for each field.

free_field (*field*) frees the storage allocated for the given field

field_arg (*field*) returns a pointer to the field arguments associated with the field type of *field*

field_info (*field*, *rows*, *cols*, *frow*, *fcol*, *nrow*, *nbuf*) returns the size, position, and other named field characteristics to the locations pointed to by the pointer arguments *rows*, *cols*, *frow*, *fcol*, *nrow*, and *nbuf*

move_field (*field*, *frow*, *fcol*) moves the disconnected *field* to the location *frow*, *fcol* in the form subwindow

set_field_type (*field*, *type*, [*arg_1*, *arg_2*, ...]) associates the given field type with *field*. Certain field types take additional arguments. "TYPE_ALNUM", for instance, requires one, the minimum width specification for the field.

field_type (*field*) returns a pointer to the field type of *field*

set_field_just (*field*, *justification*) sets the justification for the given field. Justification may be NO_JUSTIFICATION, JUSTIFY_RIGHT, JUSTIFY_LEFT, or JUSTIFY_CENTER.

field_just (*field*) returns the indicator of the justification for the field

set_field_fore (*field, fore*) sets the foreground attribute of *field*. The foreground attribute is the curses visual display attribute used to display the field characters.

field_fore (*field*) returns the foreground attribute of *field*

set_field_back (*field, back*) sets the background attribute of *field*. The background attribute is the curses visual display attribute used to display the area immediately surrounding the field characters.

field_back (*field*) returns the background attribute of *field*

set_field_pad (*field, pad*) sets the pad (blank) character for *field*

field_pad (*field*) returns the pad character for *field*

set_field_buffer (*field, buf, value*) sets buffer *buf* of *field* to *value*. Buffer 0 stores the displayed value of the field.

field_buffer (*field, buf*) returns the value of *field* buffer *buf*

set_field_status (*field, status*) sets the field's status flag to *status*. Every field has an associated status flag that is set whenever the field's value (field buffer 0) changes.

field_status (*field*) returns the status of *field*. Every field has an associated user pointer that you can use to store pertinent data.

set_field_userptr (*field, userptr*) sets the field's user pointer.

field_userptr (*field*) returns the field's user pointer

set_field_opts (*field, opts*) turns on the named options of the field and turns off its remaining options. Options are boolean values.

field_opts_on (*field, opts*) turns on the named options

field_opts_off (*field, opts*) turns off the named options

field_opts (*field*) returns the field's options setting. To set options, you can apply boolean operators to the value returned by **field_opts()** and let the result be the second argument to **set_field_opts()**.

options:

O_ACTIVE	field visited during processing
O_AUTOSKIP	moves to start of next field when current field full
O_BLANK	whole field erased if first character entered before any other character changed

field(S)

O_EDIT	field can be edited
O_NULLOK	can leave blank field without validating it
O_PUBLIC	field displayed as data entered
O_VISIBLE	field displayed
O_WRAP	words not fitting on field line are wrapped to next line

Diagnostics

The following values are returned by one or more routines that return an integer. For specific information on which routines and values returned, see the *User Interfaces Guide*.

E_BAD_ARGUMENT	argument is incorrect
E_BAD_STATE	called from inappropriate routine
E_CONNECTED	object is connected
E_CURRENT	field is current field
E_INVALID_FIELD	field is invalid
E_NOT_CONNECTED	object is not connected
E_NO_ROOM	form does not fit in subwindow
E_OK	function returned successfully
E_POSTED	form is posted
E_REQUEST_DENIED	recognized request failed
E_SYSTEM_ERROR	system error
E_UNKNOWN_COMMAND	unknown command was given to the form driver

See also

curses(S), fieldtype(S), form(S), item(S), menu(S), panel(S), tam(S)

Standards conformance

field is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

fields

return status based on fields of authentication database

Syntax

```
cc ... -lprot -lx

#include <sys/types.h>
#include <sys/security.h>
#include <sys/audit.h>
#include <prot.h>

int locked_out(pr)
struct pr_passwd *pr;

int read_pw_fields(fld, flg)
struct pr_field *fld;
struct pr_flag *flg;

int store_pw_fields(f, name, fd, fg)
FILE *f;
char *name;
struct pr_field *fd;
struct pr_flag *fg;

void read_tc_fields(fld, flg)
struct t_field *fld;
struct t_flag *flg;

int store_tc_fields(f, name, fd, fg)
FILE *f;
char *name;
struct t_field *fd;
struct t_flag *fg;
```

Description

These routines return various information based on the user, process environment, and the values within the authentication database.

locked_out returns 1 if the user represented in the *pr_passwd* structure referenced by *pr* cannot log in for some reason and returns 0 if the user is able to log in. The reasons include: unconditional lock-out from the system administrator, too many unsuccessful login tries, and/or the password lifetime has past. This routine is used prior to fashioning a session for an account, be it login, at, cron, or su sessions.

read_pw_fields fills in the *pr_field* and *pr_flag* parts of a *pr_passwd* structure with the contents from the current protected password entry. This entry must previously be obtained with a **getprpwent(S)** or **getprpwnam(S)** call.

fields(S)

store_pw_fields is the inverse of **read_pw_fields**. It creates a file entry based on one of the *pr_passwd* structure components *pr_field* and *pr_flag*, and writes them to the protected password database under the key **name** to file **f**.

The **read_tc_fields** and **store_tc_fields** routines perform similar actions for the Terminal Control Database using the appropriate field and flag structures.

The **read_pw_fields**, and **read_tc_fields** routines may be applied to either an entry in their respective databases or to a defaults entry. The *pr_passwd*, and *pr_term*, structures contain extra elements to hold default values.

See also

authcap(F), authcap(S), getprpwent(S), getprtcent(S)

Standards conformance

fields is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

fieldtype

FIELDTYPE library routines

Syntax

```
cc ... -lform -lcurses

#include <form.h>

typedef int (* PTF_int) ();

FIELDTYPE * new_fieldtype (field_check, char_check)
PTF_int field_check;
PTF_int char_check;

int free_fieldtype(fieldtype);
FIELDTYPE * fieldtype;

typedef char * (* PTF_charP) ();
typedef void (* PTF_void) ();

int set_fieldtype_arg (fieldtype, makearg, copyarg, freearg)
FIELDTYPE * fieldtype;
PTF_charP makearg;
PTF_charP copyarg;
PTF_void freearg;

typedef char * (* PTF_charP) ();

int set_fieldtype_choice (fieldtype, next, prev)
FIELDTYPE * fieldtype;
PTF_int next;
PTF_int prev;

int _nextchoice (FIELD * f, char * arg);
int _prevchoice (FIELD * f, char * arg);

FIELDTYPE * link_fieldtype (type1, type2)
FIELDTYPE * type1;
FIELDTYPE * type2;
```

Description

These FIELDTYPE routines run on any terminal supported by the `curses(S)` library. Once you compile your program, sourcing the header file *form.h* with the `#include` statement, you should link it with the `form` and `curses` library routines.

Functions

The following is a list of FIELDTYPE routines:

new_fieldtype (*field_check*, *char_check*) creates a new field type. You must write functions *field_check*, which validates the field values, and *char_check*, which validates each character.

free_fieldtype (*fieldtype*) frees the space allocated for the given field type.

By associating the given function pointers with the field type, **set_fieldtype_arg** (*fieldtype*, *makearg*, *copyarg*, *freearg*) connects to the field type additional arguments necessary for a **set_field_type**() call. Function *makearg* allocates a structure for the field specific parameters to **set_field_type**() and returns a pointer to the saved data. Function *copyarg* duplicates the structure created by *makearg*. Function *freearg* frees any storage allocated by *makearg* or *copyarg*.

Use **REQ_NEXT_CHOICE** or **REQ_PREV_CHOICE** to choose the next or previous value of a field type comprising an ordered set of values. **set_fieldtype_choice** (*fieldtype*, *next_choice*, *prev_choice*) enables you to implement these requests for the given field type. It associates with the given field type application-defined functions that return pointers to the next or previous choice for the field.

link_fieldtype (*type1*, *type2*) returns a pointer to the field type built from the two given types. The constituent types may be any application-defined or curses-defined types.

Diagnostics

The following values are returned by one or more routines that return an integer.

E_BAD_ARGUMENT	argument is incorrect
E_BAD_STATE	called from inappropriate routine
E_CONNECTED	object is connected
E_CURRENT	field is current field
E_INVALID_FIELD	field is invalid
E_OK	function returned successfully
E_NOT_CONNECTED	object is not connected
E_NO_ROOM	form does not fit in subwindow
E_POSTED	form is posted
E_REQUEST_DENIED	recognized request failed
E_SYSTEM_ERROR	system error
E_UNKNOWN_COMMAND	unknown command was given to the form driver

See also

curses(S), field(S), form(S), item(S), menu(S), panel(S), tam(S)

Standards conformance

fieldtype is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

(S)

floor, ceil, fmod, fabs

floor, ceiling, remainder, absolute value functions

Syntax

```
cc ... -lm

#include <math.h>

double floor (x)
double x;

double ceil (x)
double x;

double fmod (x, y)
double x, y;

double fabs (x)
double x;
```

Description

floor returns the largest integer (as a double-precision number) not greater than x .

ceil returns the smallest integer not less than x .

fmod returns the floating-point remainder of the division of x by y : x if y is zero or if x/y would overflow; otherwise the number f with the same sign as x , such that $x = iy + f$ for an integer i , and $|f| < |y|$.

fabs returns the absolute value of x , $|x|$.

See also

abs(S)

Standards conformance

ceil, **fabs**, **floor** and **fmod** are conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
ANSI X3.159-1989 Programming Language — C;
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

fopen, freopen, fdopen

open a stream

Syntax

```
cc ... -lc

#include <stdio.h>

FILE *fopen (filename, type)
const char *filename, *type;

FILE *freopen (filename, type, stream)
const char *filename, *type;
FILE *stream;

FILE *fdopen (fildes, type)
int fildes;
const char *type;
```

Description

The **fopen** function opens the file named by *filename* and associates a *stream* with it. The **fopen** function returns a pointer to the FILE structure associated with the *stream*.

filename points to a character string that contains the name of the file to be opened.

type is a character string with one of the following values:

- r* open for reading
- w* truncate or create for writing
- a* append; open for writing at end of file, or create for writing
- r+* open for update (reading and writing)
- w+* truncate or create for update
- a+* append; open or create for update at end-of-file

freopen substitutes the named file in place of the open *stream*. The original *stream* is closed, regardless of whether the open succeeds. **freopen** returns a pointer to the FILE structure associated with *stream*.

freopen is typically used to attach the preopened *streams* associated with *stdin*, *stdout*, and *stderr* to other files.

fdopen associates a *stream* with a file descriptor. File descriptors are obtained from **open(S)**, **dup(S)**, **creat(S)**, or **pipe(S)**, which open files but do not return pointers to a FILE structure *stream*. Streams are necessary input for many library routines. The *type* of *stream* must agree with the mode of the open file. See Warning below.

When a file is opened for update, both input and output can be done on the resulting *stream*. However, output cannot be directly followed by input without an intervening **fseek** or **rewind**, and input cannot be directly followed by output without an intervening **fseek**, **rewind**, or input operation that encounters end-of-file.

When a file is opened for append (that is, when *type* is "a" or "a+"), it is impossible to overwrite information already in the file. The **fseek** function can be used to reposition the file pointer to any position in the file. However, when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output. If two separate processes open the same file for append, each process can write to the file without destroying output written by the other. The output from the two processes is intermixed in the file in the order in which it is written.

Diagnostics

fopen, **fdopen**, and **freopen** return a NULL pointer on failure.

Warning

If the value of the file descriptor passed to **fdopen** = 60, then **fdopen** will fail and return an error.

See also

creat(S), **dup(S)**, **fclose(S)**, **fseek(S)**, **open(S)**, **pipe(S)**, **stdio(S)**

Standards conformance

fdopen is conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

fopen and **freopen** are conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
ANSI X3.159-1989 Programming Language — C;
Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

(S)

fork

create a new process

Syntax

```
cc ... -lc  
  
int fork ( )
```

Description

The **fork** system call creates a new process. The new process (child process) is an exact copy of the calling process (parent process). The child process inherits the following attributes from the parent process:

- environment: see **putenv(S)**, **getenv(S)**, **environ(S)**, **exec(S)**
- close-on-exec flag: see **fcntl(S)**, **exec(S)**
- signal-handling settings (that is, **SIG_DFL**, **SIG_IGN**, **SIG_HOLD**, function address): see **sigaction(S)**, **signal(S)**
- signal mask: see **sigprocmask(S)**
- profiling on/off status: see **profil(S)**
- nice value: see **nice(S)**
- all attached shared memory segments: see **shmop(S)**
- process group ID: see **setpgid(S)**, **setsid(S)**, **setpgrp(S)**, **getpgid(S)**, **getpgrp(S)**
- tty group ID: see **tcsetpgrp(S)**, **tcgetpgrp(S)**, **exit(S)**
- current working directory: see **chdir(S)**, **getcwd(S)**
- *root* directory: see **chroot(S)**
- file mode creation mask: see **umask(S)**
- file size limit: see **ulimit(S)**
- real, effective, and saved user IDs: see **setuid(S)**, **getuid(S)**, **geteuid(S)**, **exec(S)**
- real, effective and saved group IDs: see **setgid(S)**, **getgid(S)**, **getegid(S)**, **exec(S)**
- login user ID: see **setluid(S)**, **getluid(S)**
- supplemental groups: see **setgroups(S)**, **getgroups(S)**
- controlling terminal

The child process differs from the parent process in the following ways:

- The child process has a unique process ID: see **getpid(S)**
- The child process has a different parent process ID: see **getppid(S)**
- The child process has its own copy of the parent's file descriptors. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.
- No signals are pending in the child: see **sigpending(S)**
- All **semadj** values are cleared: see **semop(S)**
- Process locks, text locks, and data locks are not inherited by the child: see **plock(S)**, **lock(S)**
- The child process' **tms** structure is cleared (setting **utime**, **stime**, **cutime**, and **cstime** to 0): see **times(S)**
- The time left until an alarm clock or itimer signal is reset to 0: see **alarm(S)**, **setitimer(S)**, **getitimer(S)**
- Record locks set by the parent are not inherited by the child: see **fcntl(S)**, **locking(S)**, **lockf(S)**

The **fork** system call fails and no child process is created if one or more of the following is true:

- [EAGAIN] • The system-imposed limit on the total number of processes under execution would be exceeded.
- The system-imposed limit on the total number of processes under execution by a single user would be exceeded.
- The total amount of system memory available when reading via raw I/O is temporarily insufficient.
- [EMFILE] The shared data table would overflow.
- [ENOMEM] The process requires more space than the system can supply.

See also

alarm(S), **exec(S)**, **getitimer(S)**, **nice(S)**, **plock(S)**, **ptrace(S)**, **semop(S)**, **setgid(S)**, **setgroups(S)**, **setluid(S)**, **setuid(S)**, **shmop(S)**, **signal(S)**, **sigset(S)**, **times(S)**, **ulimit(S)**, **umask(S)**, **wait(S)**

Diagnostics

Upon successful completion, **fork** returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of -1 is returned to the parent process; no child process is created, and **errno** is set to indicate the error.

Standards conformance

fork is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

form

FORM library routines

Syntax

```
cc ... -lform -lcurses

#include <form.h>

typedef void (*PTF_void) ();

FIELD *current_field (form)
FORM *form;

int field_count (form)
FORM *form;

int field_index (field)
FIELD *field;

PTF_void field_init (form)
FORM *form;

PTF_void field_term (form)
FORM *form;

int form_driver (form, c)
FORM *form;
int c;

FIELD **form_fields (form)
FORM *form;

PTF_void form_init (form)
FORM *form;

OPTIONS form_opts (form)
FORM *form;

int form_opts_on (form, opts)
FORM *form;
OPTIONS *opts;

int form_opts_off (form, opts)
FORM *form;
OPTIONS *opts;

int form_page (form)
FORM *form;

WINDOW *form_sub (form)
FORM *form;
```

```
PTF_void form_term (form)
FORM *form;

char *form_userptr (form)
FORM *form;

WINDOW *form_win (form)
FORM *form;

int free_form (form)
FORM *form;

FORM *new_form (fields)
FIELD **fields;

int new_page (field)
FIELD *field;

int pos_form_cursor (form)
FORM *form;

int post_form (form)
FORM *form;

int scale_form (form, rows, cols)
FORM *form;
int *rows, cols;

int set_current_field (form, field)
FORM *form;
FIELD *field;

int set_field_init (form, func)
FORM *form;
PTF_void func;

int set_field_term (form, func)
FORM *form;
PTF_void func;

int set_form_fields (form, fields)
FORM *form;
FIELD **fields;

int set_form_init (form, func)
FORM *form;
PTF_void func;

int set_form_opts (form, opts)
FORM *form;
OPTIONS opts;

int set_form_page (form, page)
FORM *form;
int page;
```

```

int set_form_sub (form, window)
FORM *form;
WINDOW *window;

int set_form_term (form, func)
FORM *form;
PTF_void func;

int set_form_userptr (form, userptr)
FORM *form;
char *userptr;

int set_form_win (form, window)
FORM *form;
WINDOW *window;

int set_new_page (field, bool)
FIELD *field;
int bool;

int unpost_form (form)
FORM *form;

```

Description

FORM routines run on any terminal supported by the curses(S) library. Once you compile your program including the FORM header file `<form.h>`, you should link it with the `form` and `curses` library routines.

Functions

The following is a list of FORM routines:

new_form (*fields*) creates a new form connected to the designated fields and returns a pointer to the form.

free_form (*form*) disconnects the form from its associated field pointer array and deallocates the space for the form.

set_new_page (*field, bool*) marks the given field to begin a new page of the form.

new_page (*field*) returns a boolean value indicating whether the given field begins a new page of the form.

set_form_fields (*form, fields*) changes the fields connected to *form* to *fields*.

form_fields (*form*) returns a pointer to the field pointer array connected to *form*.

field_count (*form*) returns the number of fields connected to *form*.

set_form_win (*form, window*) sets *window* as the form window of *form*.

form_win (*form*) returns a pointer to the window associated with *form*.

set_form_sub (*form, window*) sets *window* as the form subwindow of *form*.

form_sub (*form*) returns a pointer to the subwindow associated with *form*.

set_current_field (*form, field*) sets the current field of *form* to *field*.

current_field (*form*) returns a pointer to the current field of *form*.

field_index (*field*) returns the index in the field pointer array to the given *field*.

set_form_page (*form, page*) sets the page number of *form* to *page*.

form_page (*form*) returns the current page number of *form*.

scale_form (*form, rows, cols*) returns the smallest window size necessary for *form*. *rows* and *cols* are pointers to the locations used to return the number of rows and columns for the form.

form_driver (*form, c*) checks if the character *c* is a form request or data. If it is a request, the form driver executes the request and reports the result. If it is data (a printable ASCII character), it enters the data into the current position in the current field. If it is not recognized, the form driver assumes it is an application-defined command and returns `E_UNKNOWN_COMMAND`.

Use the following **set_** functions to establish application routines that execute automatically at initialization and termination points in your form application. You do not have to specify these routines, but they may be helpful for various tasks.

set_form_init (*form, func*) sets an application-defined initialization *func* to be called when the form is posted and just after a page change.

form_init (*form*) returns a pointer to the initialization function, if any, called when the form is posted and just after a page change.

set_form_term (*form, func*) sets an application-defined *func* to be called when the form is unposted and just before a page change.

form_term (*form*) returns a pointer to the termination function, if any, called when the form is unposted and just before a page change.

set_field_init (*form, func*) sets an application-defined *func* to be called when the form is posted and just after the current field changes.

field_init (*form*) returns a pointer to the initialization function, if any, called when the form is posted and just after the current field changes.

set_field_term (*form*, *func*) sets *func* to be called when the form is unposted and just before the current field changes.

field_term (*form*) returns a pointer to the termination function, if any, called when the form is unposted and just before the current field changes.

post_form (*form*) writes the *form* in its associated subwindow.

unpost_form (*form*) erases the *form* from its associated subwindow.

pos_form_cursor (*form*) moves the form window cursor to the location required by the form driver to resume form processing. This is sometimes helpful after you write a message or page number.

set_form_userptr (*form*, *userptr*) sets the form's user pointer. Every form has an associated user pointer that you can use to store pertinent data.

form_userptr (*form*) returns the form's user pointer.

set_form_opts (*form*, *opts*) turns on the named options for the form and turns off all its remaining options. Options are boolean values. Currently, there are two form options, **O_NL_OVERLOAD** and **O_BS_OVERLOAD**.

form_opts (*form*) returns the form's options setting.

form_opts_on (*form*, *opts*) turns on the named options.

form_opts_off (*form*, *opts*) turns off the named options.

See also

curses(S), **field(S)**, **fieldtype(S)**, **item(S)**, **menu(S)**, **panel(S)**, **tam(S)**

Diagnostics

The following values are returned by one or more routines that return an integer.

E_BAD_ARGUMENT	argument is incorrect
E_BAD_STATE	called from inappropriate routine
E_CONNECTED	object is connected
E_CURRENT	field is current field
E_INVALID_FIELD	field is invalid
E_NO_ROOM	form does not fit in subwindow
E_NOT_CONNECTED	object is not connected

form(S)

E_OK	function returned successfully
E_POSTED	form is posted
E_REQUEST_DENIED	recognized request failed
E_SYSTEM_ERROR	system error
E_UNKNOWN_COMMAND	unknown command was given to the form driver

Standards conformance

form is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

fpgetround, fpsetround, fpgetmask, fpsetmask, fpgetsticky, fpsetsticky

IEEE floating point environment control

Syntax

```
cc ... -lc

#include <ieeefp.h>

typedef enum {

    FP_RN=0,      /* round to nearest */
    FP_RM=1,      /* round toward minus infinity */
    FP_RP=2,      /* round toward plus infinity */
    FP_RZ=3       /* round to zero (truncate) */

} fp_rnd;

fp_rnd fpgetround()

fp_rnd fpsetround(rnd_dir)
fp_rnd rnd_dir;

#define fp_except    int
#define FP_X_INV     0x01    /* invalid operation exception*/
#define FP_X_OFL     0x08    /* overflow exception*/
#define FP_X_UFL     0x10    /* underflow exception*/
#define FP_X_DZ      0x04    /* divide-by-zero exception*/
#define FP_X_IMP     0x20    /* imprecise (loss of precision)*/
#define FP_X_DNML    0x02    /* denormalization exception */

fp_except fpgetmask()

fp_except fpsetmask(mask)
fp_except mask;

fp_except fpgetsticky()

fp_except fpsetsticky(sticky)
fp_except sticky;
```

Description

There are six floating point exceptions: divide-by-zero, overflow, underflow, imprecise (inexact) result, denormalization, and invalid operation. When a floating point exception occurs, the corresponding sticky bit is set; if the mask bit is enabled, the trap takes place. These routines let the user change the behavior if any of these exceptions occur, as well as change the rounding mode for floating point operations.

fpgetround	returns the current rounding mode.
fpsetround	sets the rounding mode and returns the previous rounding mode.
fpgetmask	returns the current exception masks.
fpsetmask	sets the exception masks and returns the previous setting.
fpgetsticky	returns the current exception sticky flags.
fpsetsticky	sets (clears) the exception sticky flags and returns the previous setting.

The default environment on the Intel 80386 processor family is:

Rounding mode set to nearest (FP_RN),
Divide-by-zero, Floating point overflow, and
Invalid operation traps enabled.

See also

isnan(S)

Warnings

fpsetsticky modifies all sticky flags. **fpsetmask** changes all mask bits.

C requires truncation (round to zero) for floating point to integral conversions. The current rounding mode has no effect on these conversions.

Notes

You must clear the sticky bit to recover from the trap and to proceed. If the sticky bit is not cleared before the next floating point instruction is executed, a wrong exception type may be signaled.

When you call **fpsetmask** make sure that the sticky bit corresponding to the enabled exception is cleared.

Standards conformance

fpgetround is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

fread, fwrite

binary input/output

Syntax

```
cc ... -lc

#include <stdio.h>
#include <sys/types.h>

size_t fread (ptr, size, nitems, stream)
void *ptr;
size_t size;
int nitems;
FILE *stream;

size_t fwrite (ptr, size, nitems, stream)
void *ptr;
size_t size;
int nitems;
FILE *stream;
```

SVID Syntax

```
int fread (ptr, size, nitems, stream)
void *ptr;
size_t size;
int nitems;
FILE *stream;

int fwrite (ptr, size, nitems, stream)
void *ptr;
size_t size;
int nitems;
FILE *stream;
```

Description

The **fread** function copies *nitems* items of data from the named input *stream* into an array pointed to by *ptr*. An item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. **fread** stops appending bytes if an end-of-file or error condition is encountered while reading *stream*, or if *nitems* items have been read. **fread** leaves the file pointer in *stream*, if defined, pointing to the byte following the last byte read if there is a byte. **fread** does not change the contents of *stream*.

fwrite appends at most *nitems* items of data from the array pointed to by *ptr* to the named output *stream*. **fwrite** stops appending when it has appended *nitems* items of data or if an error condition is encountered on *stream*. **fwrite** does not change the contents of the array pointed to by *ptr*.

The argument *size* is typically `sizeof(*ptr)` where the pseudo-function `sizeof` specifies the length of an item pointed to by *ptr*. If *ptr* points to a data type other than *char*, it should be cast into a pointer to *char*.

Diagnostics

The **fread** and **fwrite** functions return the number of items read or written. If *nitems* is non-positive, no characters are read or written, and 0 is returned by both **fread** and **fwrite**.

See also

fopen(S), **getc(S)**, **gets(S)**, **printf(S)**, **putc(S)**, **puts(S)**, **read(S)**, **scanf(S)**, **stdio(S)**
write(S)

Standards conformance

fread and **fwrite** are conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
ANSI X3.159-1989 Programming Language — C;
Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

frexp, ldexp, modf

manipulate parts of floating-point numbers

Syntax

```
cc ... -lm

#include <math.h>

double frexp (value, eptr)
double value;
int *eptr;

double ldexp (value, exp)
double value;
int exp;

double modf (value, iptr)
double value, *iptr;
```

Description

Every non-zero number can be written uniquely as $x \cdot 2^n$, where the “mantissa” (fraction) x is in the range $0.5 \leq |x| < 1.0$, and the “exponent” n is an integer. **frexp** returns the mantissa of a double *value* and stores the exponent indirectly in the location pointed to by *eptr*. If *value* is zero, both results returned by **frexp** are zero.

ldexp returns the quantity $value \cdot 2^{exp}$.

modf returns the signed fractional part of *value* and stores the integral part indirectly in the location pointed to by *iptr*.

Diagnostics

If **ldexp** would cause overflow, **±HUGE** (defined in *<math.h>*) is returned (according to the sign of *value*), and **errno** is set to **ERANGE**. If **ldexp** would cause underflow, zero is returned and **errno** is set to **ERANGE**.

Standards conformance

frexp, **ldexp** and **modf** are conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
ANSI X3.159-1989 Programming Language — C;
Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

fseek, ftell, rewind

reposition a file pointer in a stream

Syntax

```
cc ... -lc

#include <stdio.h>
#include <unistd.h>

int fseek (stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;

long ftell (stream)
FILE *stream;

void rewind (stream)
FILE *stream;
```

Description

fseek - sets position of next I/O operation on a stream

ftell - returns offset of current byte relative to beginning of file

rewind - sets position of next I/O operation but does not return a value

The **fseek** function sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the position specified by *ptrname*, which can be **SEEK_CUR**, **SEEK_END** or **SEEK_SET**. These positions are defined in the *<unistd.h>* header file as follows:

Name	Action
SEEK_CUR	set position to current location plus <i>offset</i>
SEEK_END	set position to EOF plus <i>offset</i>
SEEK_SET	set position equal to <i>offset</i> bytes

rewind(stream) is equivalent to **fseek(stream, 0L, SEEK_SET)**, except that no value is returned.

fseek and **rewind** undo any effects of **ungetc(S)**.

After **fseek** or **rewind**, the next operation on a file opened for update can be either input or output.

ftell returns the offset of the current byte relative to the beginning of the file associated with the named *stream*.

Diagnostics

The **fseek** function returns zero for correct seeks and non-zero for improper seeks, such as an **fseek** done on a file not opened via **fopen**. **fseek** also cannot be used on a terminal or on a file opened via **popen(S)**.

Warning

On System V/386, an offset returned by **ftell** is measured in bytes, and it is permissible to seek to positions relative to that offset. However, portability to non-UNIX type systems requires that an offset be used by **fseek** directly. You cannot perform arithmetic on such an offset, which is not necessarily measured in bytes.

See also

fopen(S), **lseek(S)**, **popen(S)**, **stdio(S)**, **ungetc(S)**

Standards conformance

fseek, **ftell** and **rewind** are conformant with:

AT&T SVID Issue 2;
 X/Open Portability Guide, Issue 3, 1989;
 ANSI X3.159-1989 Programming Language — C;
 Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);
 IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
 and NIST FIPS 151-1.

fsetpos

sets the file position indicator for a stream

Syntax

```
cc ... -lc

#include <stdio.h>

int fsetpos(stream, pos)
FILE *stream;
const fpos_t *pos;
```

Description

The **fsetpos** function sets the file position indicator for *stream* to the value of *pos*, which is obtained in a prior call to **fgetpos** against *stream*.

The function clears the end-of-file indicator and undoes any effects of the **ungetc(stream)**. After calling **fsetpos**, the next operation on *stream* may be either input or output.

Return value

If successful, the **fsetpos** function returns 0. On failure, the function returns a nonzero value and sets **errno** to one of the following manifest constants (defined in *stdio.h*):

EBADF	The specified stream is not a valid file handle, or is not accessible.
EINVAL	The <i>stream</i> value is invalid.

See also

fgetpos(S)

Standards conformance

fsetpos is conformant with:
ANSI X3.159-1989 Programming Language — C.

Example

```
#include <stdio.h>

FILE *stream;
fpos_t pos[];
int val;
char list[100];

main()
{
    /* open file1 */
    if ((stream = fopen ("file1","rb")) == NULL)
        printf ("Trouble opening file\n");
    else {
        /* Read some data */
        fread (list, sizeof(char), 100, stream);
        /* Save current position */
        if (fgetpos (stream, &pos) != 0)
            perror("fgetpos error");
        /* Read some more */
        fread (list, sizeof(char), 100, stream);
        /* Return to saved position */
        if (fsetpos (stream, &pos) != 0)
            perror ("fsetpos error");
    }
}
```

This program opens the file named *file1* and reads 100 characters. It then uses **fgetpos** to find and save the file position pointer. After performing another read, the program calls **fsetpos** to restore the file pointer to the saved position.

fsync

synchronize changes to a file

Syntax

```
cc ... -lc

int fsync(fildes)
int fildes;
```

Description

The **fsync(S)** routine causes all modified data and attributes of the file referred to by *fildes* to be written to storage media. This routine should be called when a program needs to record to storage media all its modifications to a file before continuing, for example, in updating a record during a transaction.

Return value

Upon successful completion, **fsync(S)** returns zero; otherwise, **fsync(S)** returns -1 and sets **errno**.

Diagnostics

fsync(S) will fail if:

- [EBADF] *fildes* is not a valid file descriptor.
- [EINTR] **fsync(S)** was interrupted by a signal.
- [EINVAL] It is impossible to write to the file referred to by *fildes*.
- [EIO] An I/O error occurred while reading from or writing to the file system.

See also

fflush(S).

Standards conformance

fsync(S) is conformant with:
X/Open Portability Guide, Issue 3, 1989.

ftok

standard interprocess communication package

Syntax

```
cc ... -lc

#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(path, id)
char *path;
char id;
```

Description

All interprocess communication facilities require you to supply a key that the `msgget(S)`, `semget(S)`, and `shmget(S)` system calls use to obtain interprocess communication identifiers. One suggested method for forming a key is to use the `ftok` subroutine. Another way to compose keys is to include the project ID in the most significant byte and to use the remaining portion as a sequence number. There are many other ways to form keys, but it is necessary for each system to define standards for forming them. If a standard is not adhered to, unrelated processes may conflict. Therefore, the most significant byte of a key should refer to a project.

`ftok` returns a key based on *path* and *id* that is usable in subsequent `msgget`, `semget`, and `shmget` system calls. *path* must be the pathname of an existing file that is accessible to the process. *id* is a character which uniquely identifies a project. `ftok` returns the same key for linked files when called with the same *id* and returns different keys when called with the same file name but different *ids*.

Diagnostics

`ftok` returns (key_t) -1 if *path* does not exist or is not accessible to the process.

Warning

If the file whose *path* is passed to `ftok` is removed when keys still refer to the file, future calls to `ftok` with the same *path* and *id* return an error. If the same file is recreated, then `ftok` is likely to return a different key than it did the first time it was called.

See also

Intro(S), msgget(S), semget(S), shmget(S)

Standards conformance

ftok is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

ftw

walk a file tree

Syntax

```
cc ... -lc

#include <sys/stat.h>
#include <ftw.h>

int ftw (path, fn, depth)
char *path;
int (*fn) ();
int depth;
```

Description

The **ftw** function recursively descends the directory hierarchy rooted in *path*. For each object in the hierarchy, **ftw** calls *fn*, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a **stat** structure (see **stat(S)**) containing information about the object, and an integer. Possible values of the integer, defined in the *<ftw.h>* header file, are:

FTW_F	file
FTW_D	directory
FTW_DNR	directory that cannot be read
FTW_NS	object for which stat could not successfully be executed

If the integer is **FTW_DNR**, descendants of that directory are not processed. If the integer is **FTW_NS**, the **stat** structure contains garbage. An example of an object that would cause **FTW_NS** to be passed to *fn* would be a file in a directory with read but without execute (search) permission.

The **ftw** function visits a directory before visiting any of its descendants.

The tree traversal continues until the tree is exhausted; an invocation of *fn* returns a nonzero value; or an error is detected within **ftw** (such as an I/O error). If the tree is exhausted, **ftw** returns zero. If *fn* returns a nonzero value, **ftw** stops its tree traversal and returns whatever value was returned by *fn*. If **ftw** detects an error, it returns -1 and sets the error type in **errno**.

The **ftw** function uses one file descriptor for each level in the tree. The *depth* argument limits the number of file descriptors used. If *depth* is zero or negative, the effect is the same as if it were 1. *depth* must not be greater than the number of file descriptors currently available for use. **ftw** runs faster if *depth* is at least as large as the number of levels in the tree.

Notes

Because **ftw** is recursive, it can terminate with a memory fault when applied to very deep file structures.

The **ftw** function uses **malloc** to allocate dynamic storage during its operation. If **ftw** is forcibly terminated (for example, if **fn** or an interrupt routine executes **longjmp**), **ftw** does not have a chance to free that storage, so it remains permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have **fn** return a nonzero value at its next invocation.

See also

malloc(S), **stat**(S)

Standards conformance

ftw is conformant with:

AT&T SVID Issue 2;
and X/Open Portability Guide, Issue 3, 1989.

gamma

log gamma function

gamma - log gamma function

lgamma - xpg3 log gamma function

signgam - sign of value returned by gamma or lgamma

Syntax

```
cc ... -lm

#include <math.h>

double gamma (x)
double x;

double lgamma (x)
double x;

extern int signgam;
```

Description

The **gamma** and **lgamma** functions both return $\ln(|\Gamma(x)|)$, where

$$\Gamma(x) = \int_0^{\infty} e^{-t} t^{x-1} dt.$$

The sign of $\Gamma(x)$ is returned in the external integer **signgam**. The **lgamma** routine simply calls **gamma**, but is only available when the **cc -xpg3** option is specified to obtain compliant XPG3 code. The argument x may not be a non-positive integer.

The following C program fragment might be used to calculate Γ :

```
if ((y = gamma(x)) > LN_MAXDOUBLE)
    error();
y = signgam * exp(y);
```

where **LN_MAXDOUBLE** is the least value that causes **exp(S)** to return a range error, and is defined in the **<values.h>** header file.

See also

cc(CP), **exp(S)**, **matherr(S)**, **values(M)**

Diagnostics

For non-negative integer arguments **HUGE** is returned, and **errno** is set to **EDOM**. A message indicating **SING** error is printed on the standard error output [for example, **gamma (-5.0)**].

If the correct value would overflow, **gamma** returns **HUGE** and sets **errno** to **ERANGE**.

These error-handling procedures can be changed with the function **matherr(S)**.

Standards conformance

gamma is conformant with:

AT&T SVID Issue 2;

and X/Open Portability Guide, Issue 3, 1989.

getc, getchar, fgetc, getw

get character or word from a stream

Syntax

```
cc ... -lc

#include <stdio.h>

int getc (stream)
FILE *stream;

int getchar ( )

int fgetc (stream)
FILE *stream;

int getw (stream)
FILE *stream;
```

Description

getc - return next character from named input stream

getchar - return next character from *stdin*

fgetc - return next character from named input stream

getw - return next word from named input stream

The **getc** function returns the next character (that is, the next byte) from the named input *stream*, as an integer. It also moves the file pointer, if defined, ahead one character in *stream*. **getchar** is defined as **getc(stdin)**. **getc** and **getchar** are macros.

The **fgetc** function behaves like **getc**, but is a function rather than a macro. **fgetc** runs more slowly than **getc**, but it takes less space per invocation and its name can be passed as an argument to a function.

The **getw** function returns the next word (that is, the next integer) from the named input *stream*. **getw** increments the associated file pointer, if defined, to point to the next word. The size of a word is the size of an integer and varies from machine to machine. **getw** assumes no special alignment in the file.

Diagnostics

These functions return the constant EOF at end-of-file or upon an error. Because EOF is a valid integer, **ferror(S)** should be used to detect **getw** errors.

Warning

If the integer value returned by **getc**, **getchar**, or **fgetc** is stored into a character variable and then compared against the integer constant **EOF**, the comparison may never succeed, because the sign-extension of a character on widening to an integer is machine-dependent.

Notes

Because it is implemented as a macro, **getc** evaluates a *stream* argument more than once. In particular, **getc(*f++)** does not work well. **fgetc** should be used instead.

Because of possible differences in word length and byte ordering, files written using **putw** are machine-dependent, and may not be read using **getw** on a different processor.

Example

The **getchar** function is typically used in a conditional loop to read a string of characters from the standard input. For example, the following function reads *cnt* number of characters from the keyboard:

```
readn (p, cnt)
char  p[ ];
int   cnt;
{
    int i,c;

    i = 0;
    while ( i<cnt )
        if (( p[i++] = getchar()) == EOF ) {
            p[i] = 0;
            return(EOF);
        }
    return(0);
}
```

Note that if **getchar** is reading from the keyboard, it waits for characters to be entered before returning.

See also

fclose(S), **ferror(S)**, **fopen(S)**, **fread(S)**, **gets(S)**, **putc(S)**, **scanf(S)**, **stdio(S)**

Standards conformance

fgetc, **getc** and **getchar** are conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
ANSI X3.159-1989 Programming Language — C;

Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C
Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

getw is conformant with:

AT&T SVID Issue 2;
Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);
and X/Open Portability Guide, Issue 3, 1989.

getcwd

get pathname of current working directory

Syntax

```
cc ... -lc

char *getcwd (buf, size)
char *buf;
int size;
```

Description

The **getcwd** function returns a pointer to the current directory pathname. The value of *size* must be at least two greater than the length of the pathname to be returned.

If *buf* is a NULL pointer, **getcwd** obtains *size* bytes of space using **malloc(S)**. In this case, the pointer returned by **getcwd** may be used as the argument in a subsequent call to **free**.

The function is implemented by using **popen(S)** to pipe the output of the **pwd(C)** command into the specified string space.

Example

The following is an example of the **getcwd** function:

```
void exit(), perror();
.
.
.
if ((cwd = getcwd((char *)NULL, 64)) == NULL) {
    perror ("pwd");
    exit (-1);
}
printf ("%s\n", cwd);
```

Diagnostics

Returns NULL with **errno** set if *size* is not large enough, or if an error occurs in a lower level function.

[EINVAL] If size is zero or not large enough to hold the pathname.

See also

malloc(S), popen(S), pwd(C)

Standards conformance

getcwd is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

getdents

read directory entries and put in a filesystem-independent format

Syntax

```
cc ... -lc

#include <sys/dirent.h>

int getdents (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

Description

The *fildes* argument is a file descriptor obtained from an `open(S)` or `dup(S)` system call.

The `getdents` system call attempts to read *nbyte* bytes from the directory associated with *fildes* and to format them as filesystem-independent directory entries in the buffer pointed to by *buf*. Since the filesystem-independent directory entries are of variable length, in most cases the actual number of bytes returned is strictly less than *nbyte*.

The filesystem-independent directory entry is specified by the `dirent` structure. For a description, see *dirent(F)*.

On devices capable of seeking, `getdents` starts at a position in the file given by the file pointer associated with *fildes*. Upon return from `getdents`, the file pointer is incremented to point to the next directory entry.

This system call was developed to implement the `readdir(S)` routine (see *directory(S)*), and should not be used for other purposes.

The `getdents` system call fails if one or more of the following is true:

- [EBADF] *fildes* is not a valid file descriptor open for reading.
- [EFAULT] *buf* points outside the allocated address space.
- [EINVAL] *nbyte* is not large enough for one directory entry.
- [EIO] An I/O error occurred while accessing the file system.
- [ENOENT] The current file pointer for the directory is not located at a valid entry.
- [ENOLINK] *fildes* points to a remote machine and the link to that machine is no longer active.
- [ENOTDIR] *fildes* is not a directory.

Diagnostics

Upon successful completion, a non-negative integer is returned, indicating the number of bytes actually read. A value of 0 indicates the end of the directory has been reached. If the system call failed, a -1 is returned, and `errno` is set to indicate the error.

See also

`directory(S)`, `dirent(F)`, `dup(S)`, `open(S)`, `readdir(S)`

Standards conformance

`getdents` is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

(S)

getdvagent, getdvagnam, setdvagent, enddvagent, putdvagnam, copydvagent

manipulate device assignment database entry

Syntax

```
cc ... -lprot-lx

#include <sys/types.h>
#include <sys/security.h>
#include <sys/audit.h>
#include <prot.h>

struct dev_asg *getdvagent ( )

struct dev_asg *getdvagnam (name)
char *name;

void setdvagent ( )

void enddvagent ( )

int putdvagnam (name, dv)
char *name;
struct dev_asg *dv;

struct dev_asg *copydvagent(dv)
struct dev_asg *dv;
```

Description

getdvagent - return pointer to first device assignment entry

getdvagnam - search device assignment database for device name

setdvagent - sets device assignment database back to first entry

enddvagent - free memory and close files supporting device assignment routines

putdvagnam - rewrites or adds an entry to the database

copydvagent - copies a device assignment structure and the fields to which it refers

The **getdvagent**, **getdvagnam**, and **copydvagent** routines return pointers to an object with the following structure containing the broken-out fields of an entry in the device assignment database.

Each database entry is returned as a pointer to a `dev_asg` structure, declared in the `<prot.h>` header file:

```

struct dev_asg {
    struct dev_field ufld;
    struct dev_flag uflg;
    struct dev_field sfld;
    struct dev_flag sflg;
};

struct dev_field {
    char      *fd_name;                /* external name */
    char      **fd_devs;               /* device list */
    mask_t     fd_type[AUTH_DEV_TYPE_SIZE]; /* tape, printer, terminal */
    char      **fd_users;              /* list of users */
};

/* bit offsets for fd_type: */

#define AUTH_DEV_PRINTER      0        /* device is a printer (lp) */
#define AUTH_DEV_TERMINAL    1        /* device is a terminal (login) */

#define AUTH_DEV_TAPE        2        /* device can import/export data */

/* this structure tells which of the corresponding */
/* fields in dev_field are valid (filled). */

struct dev_flag {
    unsigned    fg_name : 1,
                fg_devs : 1,
                fg_type : 1,
                fg_users: 1;
};

```

The device assignment database stores the relationship between device pathnames and real devices. Each entry contains a *name*, which is a cross reference to the terminal control database, and a list of *devices*, each of which is a pathname that corresponds to that device. Device drivers typically use different minor device numbers to correspond to different options on the same device, for example, modem control on terminals or densities on tape drives. This list allows the device assignment software of the security management program (SMP) to invalidate all references to a device when reassigning it. The list is a table of character string pointers, whose last entry is a NULL pointer.

When first called, the `getdvagent` routine returns a pointer to the first device assignment entry. Thereafter, it returns a pointer to the next entry in the database. So, successive calls can be used to search the database.

The `getdvagnam` routine searches from the beginning of the database until an entry with a device whose name matches the argument *name* is found, and returns a pointer to that entry.

The **copydvagent** routine copies a device assignment structure and the fields to which it refers to a newly allocated data area. The **dev_asg** structure returned by **copydvagent** may be freed using **free** (see **malloc(S)**).

A call to the **setdvagent** routine sets the device assignment database back to the first entry, to allow repeated searches of the database. The **enddvagent** routine frees all memory and closes all files used to support these routines.

The **putdvagnam** routine rewrites or adds an entry to the database. If there is an entry whose **fd_name** field matches the *name* argument, that entry is replaced with the contents of the **dv** structure. Otherwise, that entry is added to the end of the database.

Return value

getdvagent and **getdvagnam** return a pointer to a static structure on success, or a NULL pointer on failure. This static structure is overwritten by **getdvagent**, **getdvagnam**, and **putdvagnam**. **putdvagnam** returns 1 on success, or 0 on failure. **copydvagent** returns a pointer to the newly allocated structure on success, or a NULL pointer if there was a memory allocation error. There are no return values or errors defined for either the **setdvagent** or the **enddvagent** routines.

Note

Since the **getdvagent**, **getdvagnam**, and **putdvagnam** routines reuse a static structure when accessing the database, the values of any entry must be saved if these routines are used again.

Files

/etc/auth/system/devassign

Standards conformance

getdvagent is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

getenv

return value for environment name

Syntax

```
cc ... -lc

#include <stdlib.h>

char *getenv (name)
char *name;
```

Description

The `getenv` function searches the environment list (see `environ(M)`) for a string of the form *name* = *value* and returns a pointer to the *value* in the current environment if such a string is present. If not, it returns a `NULL` pointer.

See also

`environ(M)`, `exec(S)`, `putenv(S)`

Standards conformance

`getenv` is conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
ANSI X3.159-1989 Programming Language — C;
Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

getgrent, getgrgid, getgrnam, setgrent, endgrent, fgetgrent

get group file entry

Syntax

```
cc ... -lc

#include <grp.h>

struct group *getgrent ( )

struct group *getgrgid (gid)
gid_t gid;

struct group *getgrnam (name)
char *name;

void setgrent ( )

void endgrent ( )

struct group *fgetgrent (f)
FILE *f;
```

Description

getgrent - returns pointer to the first and successive group structures

getgrgid - searches for matching *gid*

getgrnam - searches for matching name

setgrent - rewinds the group file to allow repeated searches

endgrent - closes group file when processing is complete

fgetgrent - returns pointer to next group structure

The **getgrent**, **getgrgid**, and **getgrnam** routines return pointers to an object with the following structure containing the broken-out fields of a line in the */etc/group* file. Each line contains a **group** structure, defined in the *<grp.h>* header file.

```
struct group {
    char    *gr_name;        /* the name of the group */
    char    *gr_passwd;      /* the encrypted group password */
    int     gr_gid;          /* the numerical group ID */
    char    **gr_mem;        /* vector of pointers to member names */
};
```

When the **getgrent** routine is first called, it returns a pointer to the first **group**

structure in the */etc/group* file. Thereafter, it returns a pointer to the next **group** structure in the */etc/group* file. Thus, successive calls to **getgrent** may be used to search the entire file.

The **getgrgid** routine searches from the beginning of the */etc/group* file until a numerical group ID matching the argument *gid* is found. **getgrgid** returns a pointer to the **group** structure in which the argument *gid* was found.

Like **getgrgid**, the **getgnam** routine searches from the beginning of the */etc/group* file until a group whose name matches the argument *name* is found. **getgnam** returns a pointer to the particular **group** structure in which the argument *name* was found.

The **setgrent** routine resets the file pointer to the beginning of the group file to allow repeated searches. The **endgrent** routine closes the group file when processing is complete.

The **fgetgrent** routine returns a pointer to the next **group** structure which matches the format of */etc/group* in the stream argument *f*.

Return value

Upon successful completion, the routines **getgrent**, **getgrgid**, **getgnam**, and **fgetgrent** each returns a pointer to a **group** structure. If an end-of-file or an error is encountered on reading, these routines return a NULL pointer. No errors are defined for the **setgrent** or **endgrent** routines.

Diagnostics

Use **feof()** to test if a null return value is caused by an end-of-file or from a syntax error in the group file. The **feof** macro is described in the **ferror(S)** manual page.

Note

All information is contained in a static area, so it must be copied if it is to be saved. Otherwise, it may be overwritten on subsequent calls to these routines.

Files

/etc/group

See also

getlogin(S), **getpwent(S)**, **group(F)**

Standards conformance

fgetgrent, getgrent, endgrent, and setgrent conform with:

AT&T SVID Issue 2;

these routines have been withdrawn from XPG3.

getgrgid and **getgrnam** conform with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

getgroups

get supplementary group ID's

Syntax

```
cc ... -lc

#include <sys/types.h>

int getgroups (gidsetsize, grouplist)
int gidsetsize;
gid_t *grouplist[];
```

Description

The **getgroups** routine fills in the array argument *grouplist* with the supplementary group IDs of the calling process. This does not include the effective group ID of the calling process itself, unless that is one of the group IDs set by **setgroups(S)**. The argument *gidsetsize* sets the maximum number of elements in the supplied array *grouplist*. The actual number of supplementary group IDs is returned. The values of the array entries with indices larger than or equal to the returned value are undefined since the array *grouplist* will have overflowed.

As a special case, if the argument *gidsetsize* is zero, **getgroups** returns the number of supplemental group IDs associated with the calling process without modifying the array to which the argument *grouplist* points.

Return value

Upon successful completion, the **getgroups** routine returns the number of supplementary group IDs. This value is zero if {NGROUPS_MAX} is zero. If **getgroups** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

If one of the following conditions occurs, **getgroups** returns a value of -1 and **errno** is set to the corresponding value:

[EINVAL] The *gidsetsize* argument is not equal to zero and is less than the number of supplementary group IDs.

See also

setgroups(S)

Standards conformance

getgroups conforms with:

X/Open Portability Guide, Issue 3, 1989;

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

gethz

return the frequency of the system clock in ticks per second

Syntax

```
cc ... -lc
int gethz()
```

Description

gethz returns the frequency of the system clock in ticks per second.

Warning

In the current implementation, **gethz** searches the environment list (see **environ(M)**) for a string of the form **HZ=val**, and returns *val*. If **HZ** cannot be interpreted as an integer, **gethz** returns 0.

If **HZ** is not defined in the environment, **gethz** makes a call to **sysconf(S)** with parameter **_SC_CLK_TCK** to determine the value of **CLK_TCK**. **gethz** returns the value of **CLK_TCK**.

Note

The implementation of **gethz** may change.

See also

environ(M)

Standards conformance

gethz is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

getitimer, setitimer

get and set value of interval timers

Syntax

```
cc ... -lc

#include <sys/select.h>
#include <sys/itimer.h>

getitimer(which, value)
int which;
struct itimerval *value);

setitimer(which, value, ovalue)
int which;
const struct itimerval *value;
struct itimerval *ovalue;
```

Description

getitimer - returns the current value of the specified timer

setitimer - sets the specified interval timer

Each process has three interval timers or “itimers” associated with it. At the expiration of a programmed amount of time, an interval timer signals the process and (optionally) automatically reloads itself to re-interrupt the process at programmed intervals. Each itimer counts a different type of time, and generates a different signal upon expiration:

ITIMER_REAL	Counts in real (wall-clock) time. A SIGALRM is generated when this itimer expires. alarm(S) is a lower-precision non-reloading ITIMER_REAL ; a process may have either an alarm , a ITIMER_REAL , or neither, but never both simultaneously.
ITIMER_VIRT	Counts in process-virtual time. It runs only when the process is executing in user mode. A SIGVTALRM is generated when this itimer expires.
ITIMER_PROF	Counts in both process-virtual time and the time when the system is running on behalf of the process. A SIGPROF is generated when this itimer expires.

Each itimer counts by decrementing the programmed amount towards zero.

The **getitimer** function returns the current value for the itimer specified by the *which* parameter in the structure pointed to by the *value* parameter.

The `setitimer` function programs the *which* itimer to the specified *value*. If the *ovalue* pointer is not NULL, the previous setting is returned in the `itimerval` structure pointed to by *ovalue*.

An itimer is defined by the `itimerval` structure defined in `<sys/itimer.h>` and the `timeval` structure defined in `<sys/select.h>`:

```
struct itimerval {
    struct timeval  it_interval;    /* timer interval */
    struct timeval  it_value;       /* current value */
}

struct timeval {
    long    tv_sec;                /* seconds */
    long    tv_usec;              /* and microseconds */
}
```

If the `it_value` field is nonzero, it indicates the time to the next itimer expiration. This field is decremented as time advances. If the `it_interval` field is nonzero, it specifies a value to be used in reloading `it_value` when the itimer expires.

Setting `it_value` to zero disables an itimer. Setting `it_interval` to zero causes an itimer to be disabled after its next expiration (assuming `it_value` is nonzero).

The child process does not inherit its parent's itimers after a `fork(S)`; that is, all of the child's itimers are disabled. Interval timer settings are not changed by `exec(S)`.

Both the `it_value` and `it_interval` fields have microsecond precision. The `tv_sec` field indicates the number of seconds, and `tv_usec` the millionths of a second. A `timeval` structure is zero only when both `tv_sec` and `tv_usec` are zero.

Time values smaller than the resolution of the system's clock are rounded up to this resolution. The system clock resolution is returned by `gethz(S)` or `sysconf(S)`. Typically, the resolution is ten milliseconds (or ten thousand microseconds). The accuracy of an itimer depends on the system's load; the busier the system, the greater the delay between the generation of the signal and its delivery to the process. Since time values are rounded up, an itimer never expires before it is programmed to expire. On sufficiently busy systems (or with very short itimer intervals) a signal may not be delivered before the subsequent one is generated. Since signals don't queue, only one signal will be delivered. The system compensates and does not allow the itimer to drift; that is, each successive signal is generated when it is programmed to occur and is not dependent on whether or when the prior signal was delivered.

Four macros are defined in `<sys/itimer.h>` for manipulating `timeval` structures:

```
timercopy(
    const struct timeval    source,
    struct timeval          destination
)
```

The `timercopy` macro copies the structure pointed to by *source* to that pointed to by *destination*.

```
timerisset(
    const struct timeval    timvalptr
)
```

The `timerisset` macro is nonzero (true) if the structure pointed to by *timvalptr* is nonzero.

```
timerclear(
    struct timeval    timvalptr
)
```

The `timerclear` macro zeros the structure pointed to by *timvalptr*.

```
timercmp(
    const struct timeval    leftvptr,
    const struct timeval    rightvptr,
    operator
)
```

The `timercmp` macro compares the two intervals defined by the structures pointed to by *leftvptr* and *rightvptr* according to the specified operator (<, ==, !=, or >):

(*leftvptr's interval*) *operator* (*rightvptr's interval*)

The <= and >= operators do not work with the `timercmp` macro.

The `getitimer()` system call fails if one or more of the following is true:

- [EINVAL] *which* is not `ITIMER_REAL`, `ITIMER_VIRT`, or `ITIMER_PROF`.
- [EINVAL] *value* is `NULL`.
- [EFAULT] *value* points outside the allocated address space of the process.
- [ENOSYS] Interval timers are either not installed or not supported on this system; a `SIGSYS` signal is also generated.

The `setitimer()` system call fails if one of more of the following is true, and the current `itimer` setting is not changed:

- [EINVAL] *which* is not `ITIMER_REAL`, `ITIMER_VIRT`, or `ITIMER_PROF`.
- [EINVAL] *value* is `NULL`.
- [EINVAL] Either *value* -> *it_value* or *value* -> *it_interval* is not normalized: Either "*tv_sec*" is negative or over ten million seconds, or "*tv_usec*" is negative or over 999,999 microseconds.

[EFAULT]	Either <i>ovalue</i> or <i>value</i> points outside the allocated address space of the process.
[ENOMEM]	There are insufficient system resources to program the itimer. In this case the current <i>which</i> itimer setting is lost, disabling that itimer.
[ENOSYS]	Interval timers are either not installed or not supported on this system; a SIGSYS signal is also generated.

Diagnostics

Upon successful completion, a value of zero is returned. Otherwise, a value of -1 is returned, and `errno` is set to indicate the error.

Notes

Older systems do not have itimers, and itimers may be deconfigured (not present) in newer systems (by disabling the itimer pseudo-driver). If either the `getitimer()` or `setitimer()` system call is attempted when itimers are not available, a SIGSYS signal results. The itimer pseudo-driver will not function in older systems.

Other vendors' systems may use `<sys/time.h>` instead of `<sys/itimer.h>`.

Any call to `alarm()` disables the `ITIMER_REAL`. Any (successful) call to `setitimer()` disables any `alarm`. `getitimer()` returns information about whichever of `alarm()` and `ITIMER_REAL` is currently enabled.

Older versions of `sleep(S)` may inadvertently convert a `ITIMER_REAL` into an `alarm`, which has only seconds precision and is not reloaded when it expires.

The `ITIMER_PROF` interval timer is traditionally used by interpreters to statistically profile interpreted code. The generated SIGPROF may interrupt in-progress system calls. There is no facility to restart such interrupted system calls, so this itimer may not be very useful for such profiling.

See Also

`alarm(S)`, `exec(S)`, `fork(S)`, `gethz(S)`, `sigaction(S)`, `signal(S)`, `sleep(S)`, `sysconf(S)`

Standards conformance

`getitimer` and `setitimer` are not part of any currently supported standard; they were developed at the University of California at Berkeley and are used by permission.

getlogin

get login name

Syntax

```
cc ... -lc  
  
char *getlogin ( )
```

Description

The **getlogin** routine returns a pointer to the login name as found in the file */etc/utmp*. If **getlogin** is called within a process that is not attached to a terminal, it returns a NULL pointer.

getlogin may be used in conjunction with the **getpwnam** routine to locate the correct password file entry when the same user ID is shared by several login names.

The correct procedure for determining the login name is to call the **cuserid** or the **getlogin** routine. If both fail, one should call the **getpwuid** routine.

Return value

Upon successful completion, the **getlogin** routine returns a pointer to the login name. If the login name is not found or if an error while accessing the */etc/utmp* file occurs, **getlogin** returns the NULL pointer.

Note

The return value points to static data whose content is overwritten by each call. Therefore, the information should be copied if it needs to be saved or used later.

Files

/etc/utmp

See also

cuserid(S), getgrent(S), getpwent(S), utmp(F)

Standards conformance

getlogin conforms with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

(S)

getluid

get login user ID

Syntax

```
cc ... -lprot

#include <sys/types.h>
#include <sys/security.h>
#include <sys/audit.h>
#include <prot.h>

int getluid ()
```

Description

The **getluid** routine returns the login user ID for the process.

The login user ID, or LUID, is set at login or batch start time; once set, it cannot be changed. The LUID, an accurate representation of the user who logged into the system, cannot be altered during the session. The LUID is needed because both the effective and real user IDs can be altered by use of **setuid(S)** or by setting the **setuid** bits on an executable file, and consequently, at times during a session, do not accurately reflect the login user.

Return value

Upon successful completion the LUID is returned. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

If the following condition occurs, **getluid** fails and **errno** is set to the corresponding value:

[EPERM] 10 The login user ID has not yet been set for the process.

See also

getuid(S), **setluid(S)**, **setuid(S)**, **stat(S)**

Standards conformance

The **getluid** routine is an extension of AT&T System V provided by the Santa Cruz Operation.

getmsg

get next message off a stream

Syntax

```
cc ... -lc

#include <sys/stropts.h>

int getmsg (fd, ctrlptr, dataptr, flags)
int fd;
struct strbuf *ctrlptr;
struct strbuf *dataptr;
int *flags;
```

Description

The **getmsg** system call retrieves the contents of a message located at the stream head read queue from a STREAMS file, and places the contents into user-specified buffer(s). The message must contain either a data part, a control part, or both. The data and control parts of the message are placed into separate buffers, as described below. The semantics of each part is defined by the STREAMS module that generated the message.

The *fd* argument specifies a file descriptor referencing an open *stream*. *ctrlptr* and *dataptr* each point to a **strbuf** structure which contains the following members:

```
int maxlen;      /* maximum buffer length */
int len;         /* length of data      */
char *buf;       /* ptr to buffer      */
```

where *buf* points to a buffer in which the data or control information is to be placed, and *maxlen* indicates the maximum number of bytes this buffer can hold. On return, *len* contains the number of bytes of data or control information actually received, or is 0 if there is a zero-length control or data part, or is -1 if no data or control information is present in the message. *flags* may be set to the values 0 or RS_HIPRI and is used as described below.

The *ctrlptr* argument is used to hold the control part from the message and *dataptr* is used to hold the data part from the message. If *ctrlptr* (or *dataptr*) is NULL or the *maxlen* field is -1, the control (or data) part of the message is not processed and is left on the stream head read queue, and *len* is set to -1. If the *maxlen* field is set to 0 and there is a zero-length control (or data) part, that zero-length part is removed from the read queue and *len* is set to 0. If the *maxlen* field is set to 0 and there are more than zero bytes of control (or data) information, that information is left on the read queue and *len* is set to 0. If the *maxlen* field in *ctrlptr* or *dataptr* is less than, respectively, the control or data part of the message, *maxlen* bytes are retrieved. In this case, the remainder of the message is left on the stream head read queue and a non-

zero return value is provided, as described below under “Diagnostics”. If information is retrieved from a priority message, *flags* is set to **RS_HIPRI** on return.

By default, **getmsg** processes the first priority or non-priority message available on the stream head read queue. However, you may choose to retrieve only priority messages by setting *flags* to **RS_HIPRI**. In this case, **getmsg** only processes the next message if it is a priority message.

If **O_NONBLOCK** has not been set, **getmsg** blocks until a message, of the type(s) specified by *flags* (priority or either), is available on the stream head read queue. If **O_NONBLOCK** has been set and a message of the specified type(s) is not present on the read queue, **getmsg** fails and sets **errno** to **EAGAIN**.

If a hangup occurs on the *stream* from which messages are to be retrieved, **getmsg** continues to operate normally, as described above, until the stream head read queue is empty. Thereafter, it returns 0 in the *len* fields of *ctrlptr* and *dataptr*.

The **getmsg** system call fails if one or more of the following is true:

- [EAGAIN] The **O_NONBLOCK** flag is set, and no messages are available.
- [EBADF] *fd* is not a valid file descriptor open for reading.
- [EBADMSG] Queued message to be read is not valid for **getmsg**.
- [EFAULT] *ctrlptr*, *dataptr*, or *flags* points to a location outside the allocated address space.
- [EINTR] A signal was caught during the **getmsg** system call.
- [EINVAL] An illegal value was specified in *flags*, or the *stream* referenced by *fd* is linked under a multiplexer.
- [ENOSTR] A *stream* is not associated with *fd*.

A **getmsg** can also fail if a STREAMS error message had been received at the *stream* head before the call to **getmsg**. The error returned is the value contained in the STREAMS error message.

Diagnostics

Upon successful completion, a non-negative value is returned. A value of 0 indicates that a full message was read successfully. A return value of **MORECTL** indicates that more control information is waiting for retrieval. A return value of **MOREDATA** indicates that more data is waiting for retrieval. A return value of **MORECTL | MOREDATA** indicates that both types of information remain. Subsequent **getmsg** calls retrieves the remainder of the message.

See also

Intro(S), read(S), poll(S), putmsg(S), write(S)

Standards conformance

getmsg is conformant with:
AT&T SVID Issue 2.

(S)

getopt

get option letter from argument vector

Syntax

```
cc ... -lc

int getopt (argc, argv, optstring)
int argc;
char **argv, *optstring;

extern char *optarg;
extern int optind, opterr;
```

Description

The **getopt** function returns the next option letter in *argv* that matches a letter in *optstring*. It supports all the rules of the command syntax standard. Use **getopts(C)** or **getopt** to ensure that all new commands adhere to the command syntax standard. These routines parse positional parameters and check for options that are legal for that command.

optstring must contain the option letters the command using **getopt** recognizes; if a letter is followed by a colon, the option is expected to have an argument, or group of arguments, which must be separated from it by white space.

optarg is set to point to the start of the option-argument on return from **getopt**.

getopt places in *optind* the *argv* index of the next argument to be processed. *optind* is external and is initialized to 1 before the first call to **getopt**.

When all options have been processed (that is, up to the first non-option argument), **getopt** returns -1. The special option "--" may be used to delimit the end of the options; when it is encountered, -1 is returned, and "--" is skipped.

The following rules comprise the standard for command line syntax:

- Command names must be between two and nine characters.
- Command names must only include lowercase letters and digits.
- Option names must be a single character in length.
- All options must be delimited by the "-" character.
- Options with no arguments may be grouped behind one delimiter.
- The first option argument following an option must be preceded by white space.

- Option arguments cannot be optional.
- Groups of option arguments following an option must be separated by commas or separated by white space and quoted.
- All options must precede operands on the command line.
- The characters -- may be used to delimit the end of the options.
- The order of options relative to one another should not matter.
- The order of operands may matter and position-related interpretations should be determined on a command-specific basis.
- The " - " character preceded and followed by white space should be used only to mean standard input.

The function **getopt** is the command line parser that enforces the rules of this command syntax standard.

Diagnostics

getopt prints an error message on standard error and returns a question mark (?) when it encounters an option letter not included in *optstring* or no option-argument after an option that expects one. This error message may be disabled by setting *opterr* to 0.

Warning

Although the following command syntax rule (see **Intro(C)**) relaxations are permitted under the current implementation, they should not be used because they may not be supported in future releases of the system. As in the "Example" section above, **a** and **b** are options, and the option **o** requires an option-argument:

cmd -abxxx file	violation: options with option-arguments must not be grouped with other options
cmd -ab -oxxx file	violation: there must be white space after an option that takes an option-argument

Changing the value of the variable *optind* or calling **getopt** with different values of *argv* may lead to unexpected results.

See also

getopts(C), **Intro(C)**

Standards conformance

getopt is conformant with:

AT&T SVID Issue 2;

Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);

and X/Open Portability Guide, Issue 3, 1989.

Example

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options a and b, and the option o, which requires an option argument:

```
main (argc, argv)
int argc;
char **argv;
{
    int c;
    extern char *optarg;
    extern int optind;
    .
    .
    .
    while ((c = getopt(argc, argv, "abo:")) != -1)
        switch (c) {
            case 'a':
                if (bflg)
                    errflg++;
                else
                    aflg++;
                break;
            case 'b':
                if (aflg)
                    errflg++;
                else
                    bproc( );
                break;
            case 'o':
                ofile = optarg;
                break;
            case '?':
                errflg++;
        }
    if (errflg) {
        (void)fprintf(stderr, "usage: . . . ");
        exit (1);
    }
    for ( ; optind < argc; optind++) {
        if (access(argv[optind], 4)) {
            .
            .
            .
        }
    }
```

getpass

read a password

Syntax

```
cc ... -lc

char *getpass (prompt)
char *prompt;
```

Description

The `getpass` function opens `/dev/tty`, writes to that device the null-terminated string *prompt*, disables echoing, reads a string of characters up to the next newline character or EOF, restores the terminal state, and closes `/dev/tty`.

Return value

Upon successful completion, `getpass` returns a pointer to a null-terminated string of at most `[PASS_MAX]` characters that were read from the terminal device. If an error is encountered, for example if `/dev/tty` cannot be opened, the terminal state is restored and a `NULL` pointer is returned. An interrupt terminates input and sends an interrupt signal to the calling program before returning.

Diagnostics

If the following condition occurs, the `getpass` routine may fail and `errno` is set to the following value:

- [EINTR] The `getpass` routine was interrupted by a signal.
- [ENXIO] The process does not have a controlling terminal.

Note

The return value points to static data whose content is overwritten by each call.



getpass(S)

Files

/dev/tty

Standards conformance

getpass conforms with:

AT&T SVID Issue 2;
and X/Open Portability Guide, Issue 3, 1989.

getpasswd, fgetpasswd, bigcrypt, bigcryptmax

read or clear a password

Syntax

```
cc ... -lprot-lcrypt_i -lx

#include <sys/types.h>
#include <sys/security.h>
#include <sys/audit.h>
#include <prot.h>

char *getpasswd (prompt, max_size)
char *prompt;
int max_size;

char *fgetpasswd (prompt, max_size, file_in, file_out)
char *prompt;
int max_size;
FILE *file_in;
FILE *file_out;

char *bigcrypt (cleartext, salt)
char *cleartext;
char *salt;

char *bigcryptmax (cleartext, salt)
char *cleartext;
char *salt;
```

Description

getpasswd - read or clear a password

fgetpasswd - read or clear a password from a file

bigcrypt - encrypt a short or long password

bigcryptmax - encrypt a long password

The **getpasswd** routine, when given a non-NULL and null-terminated *prompt*, reads up to a newline or an EOF from the file */dev/tty*, after prompting on the standard error output with *prompt* and disabling echoing.

fgetpasswd gets or clears a password from a device file. Set *max_size* to `AUTH_MAX_PASSWD_LENGTH`, defined in *<prot.h>*. If *file_in* is null, information is taken from */dev/tty*; if *file_out* is null, information is written to *stderr*. The value in *prompt* is written to *file_out*. A `\n` character is inserted at the end of the string written to *file_out*. A pointer is returned to the clear-text static area.

The **bigcrypt** routine is used in the same way as **crypt(S)** to encrypt passwords of any length, including passwords longer than eight characters. **bigcrypt** utilizes the setting of the **u_pwseg** in */etc/auth/system/default*, truncating the cleartext password to the number of characters derived from **u_pwseg**. **getprpwent(S)** correspondingly truncates the encrypted password string (**u_pwd**) in the protected password entry, so that a comparison of the value returned by **bigcrypt** succeed if the password would have matched on the first (**u_pwseg** * 8) cleartext characters.

For the purposes of checking passwords, this is transparent to the application. To provide backward compatibility to applications and allow **u_pwseg** to be reduced and then increased again without losing information, it is important that the stored encrypted password (set using **putprpwnam(S)** when the password is changed) not be truncated.

So that the sequence of **getprpwent**, **putprpwnam** does not truncate the stored password when **u_pwseg** is smaller than the password in the entry, the operating system stores the original stored value of **u_pwd**. **putprpwnam** restores **u_pwd** to this saved value if the new value matches the truncated saved value.

bigcryptmax is the same as **bigcrypt** except that in **bigcryptmax** **u_pwseg** is ignored. This routine is used only when a new longer password is set in the protected password entry. Use **bigcrypt** for checking existing passwords.

Return value

Upon successful completion, the **getpasswd** routine returns a pointer to a null-terminated string of at most *max_size* characters. If */dev/tty* cannot be opened, a NULL pointer is returned. An interrupt terminates input and sends an interrupt signal to the calling program before returning.

The **bigcrypt** routine returns an encrypted password.

Note

The library **libcrypt_i.a** is not currently distributed with Development System. Please contact your supplier for information on obtaining this library.

The return value for **getpasswd** and **fgetpasswd** points to static data whose content is overwritten by each call to **getpasswd** or **fgetpasswd**. Therefore, the information should be copied if it needs to be saved or used later. As soon as that password is no longer needed, **getpasswd** or **fgetpasswd** should again be invoked with a NULL prompt string. This clears the password in the static area so that there is no chance of "dirty memory" revealing passwords later on.

Files

/dev/tty

See also

crypt(S), getpass(S), getprwent(S), putprpwnam(S)

Standards conformance

getpasswd, fgetpasswd, bigcrypt, and bigcryptmax are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

(S)

getpid, getpgrp, getppid

get process, process group, and parent process ID

Syntax

```
cc ... -lc  
pid_t getpid ()  
pid_t getpgrp ()  
pid_t getppid ()
```

Description

The **getpid** routine returns the process ID of the calling process.

The **getpgrp** routine returns the process group ID of the calling process.

The **getppid** routine returns the parent process ID of the calling process.

Return value

These routines are always successful and therefore no return value is used to signify an error.

See also

exec(S), fork(S), Intro(S), setpgrp(S), signal(S)

Standards conformance

getpgrp, getpid and **getppid** conform with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

getprdfent, getprdfnam, setprdfent, endprdfent, putprdfnam

manipulate default control database entry

Syntax

```
cc ... -lprot -lx

#include <sys/types.h>
#include <sys/security.h>
#include <sys/audit.h>
#include <prot.h>

struct pr_default *getprdfent ( )

struct pr_default *getprdfnam (name)
char *name;

void setprdfent ( )

void endprdfent ( )

int putprdfnam (name, pr)
char *name;
struct pr_default *pr;
```

Description

getprdfent - returns pointer to first and successive **pr_default** structures

getprdfnam - searches for matching login name

setprdfent - rewinds default control file to allow repeated searches

endprdfent - closes default control file when processing is complete

putprdfnam - puts a new or replaced default control entry

The **getprdfent** and **getprdfnam** routines both return a pointer to an object with the following structure containing the broken-out fields of a line in the default control database. Only entries in the database dealing with users are scanned. Each line in the database contains a **pr_default** structure, declared in the **<prot.h>** header file:

```

struct pr_default {
    char dd_name[20];
    char dg_name;
    struct pr_field prd;
    struct pr_flag prg;
    struct t_field tcd;
    struct t_flag tcg;
    struct dev_field devd;
    struct dev_flag devg;
    struct system_default_fields sfld;
    struct system_default_flags sflg;
};

```

Because this structure is declared in *<prot.h>*, it need not be redeclared.

When the **getprdfent** routine is first called, it returns a pointer to the first user **pr_default** structure in the database. Thereafter, it returns a pointer to the next user **pr_default** structure in the database. So, successive calls can be used to search the entire database.

The **getprdfnam** routine searches from the beginning of the database file until a login name matching the argument *name* is found. **getprdfnam** returns a pointer to the particular **pr_default** structure in which the argument *name* was found.

The **setprdfent** routine resets the file pointer to the beginning of the default control file to allow repeated searches. The **endprdfent** routine closes the default control file when processing is complete.

The **putprdfnam** routine puts a new or replaced default control entry *pr* with key *name* into the default control file. If the `uflg.fg_name` field is 0, the requested entry is deleted from the default control database. The **putprdfnam** routine locks the database for all update operations, and performs an **endprdfent** after the update or failed attempt.

Return value

Upon successful completion, the **getprdfent** and **getprdfnam** routines each returns a pointer to a **pr_default** structure. The **putprdfnam** routine returns a non-zero value on a successful substitution, deletion, or addition and a value of zero upon failure. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer. No errors are defined for the **setprdfent** or **endprdfent** routines.

Diagnostics

Use **feof()** to test if a null return value is caused by an end-of-file or from a syntax error in the default control file. The **feof** macro is described in the **error(S)** manual page.

Note

All information is contained in a static area, so it must be copied if it is to be saved. Otherwise, it may be overwritten on subsequent calls to these routines.

Files

/etc/auth/defaults

See also

**authcap(F), authcap(S) getprdfent(S), getprfient(S), getprpwent(S),
getprtcent(S),**

Standards conformance

endprdfent, getprdfent, getprdfnam, putprdfnam and setprdfent are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

getprfient, getprfinam, setprfient, endprfient, putprfinam

manipulate file control database entry

Syntax

```
cc ... -lprot-lx

#include <sys/types.h>
#include <sys/security.h>
#include <sys/audit.h>
#include <prot.h>

struct pr_file *getprfient ()

struct pr_file *getprfinam (name)
char *name;

void setprfient ()

void endprfient ()

int putprfinam (name, pr)
char *name;
struct pr_file *pr;
```

Description

getprfient - gets next **pr_file** structure
getprfinam - searches for matching login name
setprfient - rewinds file control file to allow repeated searches
endprfient - closes file control file when processing is complete
putprfinam - puts a new or replaced file control entry

The **getprfient** and **getprfinam** routines both return a pointer to an object with the following structure containing the broken-out fields of a line in the file control database. Only entries in the database dealing with users are scanned. Each line in the database contains a **pr_file** structure, declared in the *<prot.h>* header file:

```

struct pr_file {
    struct f_field ufld;
    struct f_flag uflg;
};

struct f_field {
    char    *fd_name;          /* Holds full path name */
    ushort  fd_uid;            /* uid of owner */
    ushort  fd_gid;            /* gid of group */
    ushort  fd_mode;           /* permissions */
    char    fd_type[2];        /* file type (one of r,b,c,d,f,s) */
    off_t    fd_size;          /* size in bytes */
    time_t   fd_mtime;         /* last time file noted as touched */
    time_t   fd_ctime;         /* last time inode noted as touched */
    long     fd_cksum;          /* checksum for file (see sum(C)) */
};

struct f_flag {
    unsigned
        fg_name:1,            /* Is fd_name set? */
        fg_uid:1,              /* Is fd_uid set? */
        fg_gid:1,              /* Is fd_gid set? */
        fg_mode:1,             /* Is fd_mode set? */
        fg_type:1,             /* Is fd_type set? */
        fg_size:1,             /* Is fd_size set? */
        fg_mtime:1,            /* Is fd_mtime set? */
        fg_ctime:1,            /* Is fd_ctime set? */
        fg_cksum:1;            /* Is fd_cksum set? */
};

```

Because this structure is declared in *<prot.h>*, it need not be redeclared.

When the *getprfient* routine is first called, it returns a pointer to the first user *pr_file* structure in the database. Thereafter, it returns a pointer to the next user *pr_file* structure in the database. So, successive calls can be used to search the entire database.

The *getprfinam* routine searches from the beginning of the database file until a login name matching the argument *name* is found. *getprfinam* returns a pointer to the particular *pr_file* structure in which the argument *name* was found.

The *setprfient* routine resets the file pointer to the beginning of the file control file to allow repeated searches. The *endprfient* routine closes the file control file when processing is complete.

The *putprfinam* routine puts a new or replaced file control entry *pr* with key *name* into the file control file. If the *uflg.fg_name* member is 0, the requested entry is deleted from the file control database. The *putprfinam* routine locks the database for all update operations, and performs an *endprfient* after the update or failed attempt.

Return value

Upon successful completion, the **getprfient** and **getprfinam** routines each returns a pointer to a **pr_file** structure. The **putprfinam** routine returns a non-zero value on a successful substitution, deletion, or addition and a value of zero upon failure. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer. No errors are defined for the **setprfient** or **endprfient** routines.

Diagnostics

Use **feof()** to test if a null return value is caused by an end-of-file or from a syntax error in the file control file. The **feof** macro is described in the **ferror(S)** manual page.

Note

All information is contained in a static area, so it must be copied if it is to be saved. Otherwise, it may be overwritten on subsequent calls to these routines.

Files

/etc/auth/files

See also

authcap(F), **authcap(S)**, **getprdfent(S)**, **getprpwent(S)**, **getprtcent(S)**

Standards conformance

endprfient, **getprfient**, **getprfinam**, **putprfinam**, and **setprfient** are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

getpriv

get system privileges associated with this process

Syntax

```
cc ... -lprot

#include <sys/types.h>
#include <sys/macro.h>
#include <sys/security.h>
#include <sys/audit.h>

int getpriv (privtype, privs)
int privtype;
priv_t *privs;
```

Description

The `getpriv` routine returns the system privilege vector for the current process in the vector argument *privs*. *privs* should have at least `SEC_SPRIVVEC_SIZE` entries. The argument *privtype* may only contain the privilege type `SEC_EFFECTIVE_PRIV`.

The system privilege vector contains per-process privileges used by the trusted control database. See `setpriv(S)` for a description of the privileges currently defined.

Return value

Upon successful completion, the `getpriv` routine returns the size, in bytes, of the privilege vector, *privs*. Otherwise, a value of -1 is returned and `errno` is set to indicate the appropriate error.

Diagnostics

If one of the following conditions occurs, `getpriv` fails and `errno` is set to the corresponding value:

- [EFAULT] *privs* points to an invalid address.
- [EINVAL] *privtype* is not `SEC_EFFECTIVE_PRIV`.

getpriv(S)

See also

setpriv(S)

Standards conformance

The **getpriv** routine is an extension of AT&T System V provided by the Santa Cruz Operation.

getprpwent, getprpwuid, getprpwnam, setprpwent, endprpwent, putprpwnam

manipulate protected password database entry

Syntax

```
cc ... -lprot -lx

#include <sys/types.h>
#include <sys/security.h>
#include <sys/audit.h>
#include <prot.h>

struct pr_passwd *getprpwent ( )

struct pr_passwd *getprpwuid (uid)
int uid;

struct pr_passwd *getprpwnam (name)
char *name;

void setprpwent ( )

void endprpwent ( )

int putprpwnam (name, pr)
char *name;
struct pr_passwd *pr;
```

Description

getprpwent - gets pointer to next **pr_passwd** structure
getprpwuid - searches for numerical user ID matching **uid**
getprpwnam - searches for login name matching **name**
setprpwent - rewinds protected password files to allow repeated searches
endprpwent - closes protected password files when processing is complete
putprpwnam - writes new or replaced protected password entry to a file

The **getprpwent**, **getprpwuid** and **getprpwnam** routines each returns a pointer to an object with the following structure containing the broken-out fields of a line in the protected password database. Only entries in the database dealing with users are scanned.

Before using **getprpwent**, call the **set_auth_parameters(S)** routine. This routine is documented on the **identity(S)** manual page.

Each line in the database contains a `pr_passwd` structure, declared in the `<prot.h>` header file:

```

struct pr_passwd {
    /* Fields associated with this user */
    struct pr_field ufld;
    /* Flags associated with this user */
    struct pr_flag uflg;
    /* Fields associated with system */
    struct pr_field sfld;
    /* Flags associated with system */
    struct pr_flag sflg;
};

struct pr_field {
    /* Identity: */
    /* uses 8 character maximum from utmp */
    char    fd_name[9];
    /* uid associated with name above */
    ushort  fd_uid;
    /* Encrypted password, 8 characters max */
    char    fd_encrypt[14];
    /* user type - see user_type */
    mask_t  fd_type[AUTH_TYPEVEC_SIZE];
    /* if a pseudo -user, the user behind it */
    char    fd_owner[9];
    /* nice value with which to login */
    /* command privilege vector */
    int     fd_nice;
    mask_t  fd_cprivs[AUTH_CPRIVVEC_SIZE];
    /* system privilege vector */
    mask_t  fd_sprivs[SEC_SPRIVVEC_SIZE];
    /* audit control */
    mask_t  fd_auditcntl[AUTH_AUDITMASKVEC_SIZE];
    /* audit disposition */
    mask_t  fd_auditdisp[AUTH_AUDITMASKVEC_SIZE];
    /* Password maintenance parameters: */
    /* min time between pswd changes */
    time_t  fd_min;
    /* maximum length of password */
    int     fd_maxlen;
    /* expiration time duration in secs */
    time_t  fd_expire;
    /* account death time duration in secs */
    time_t  fd_lifetime;
    /* last successful change */
    time_t  fd_schange;
    /* last unsuccessful change */
    time_t  fd_uchange;
    /* who can change this user's password */
    ushort  fd_pswduser;
    /* can user pick his own passwords? */
    char    fd_pick_pwd;
    /* can user get pswds generated for him? */
    char    fd_gen_pwd;
    /* should generated pswds be restricted? */

```

```

char    fd_restrict;
        /* is user allowed to have a null password? */
char    fd_nullpw;

/* Login parameters: */
        /* last successful login */
time_t  fd_slogin;
        /* tty of last successful login */
char    fd_suctty[12];
        /* last unsuccessful login */
time_t  fd_ulogin;
        /* tty of last unsuccessful login */
char    fd_unsuctty[12];
        /* consecutive unsuccessful logins */
short   fd_nlogins;
        /* maximum unsuccessful login tries allowed */
ushort  fd_max_tries;
        /* unconditionally lock account? */
char    fd_lock;

/* System parameters: */
        /* require a password on single user shell? */
char    fd_standpswd;
        /* system security class */
mask_t  fd_secclass[AUTH_SECCLASSVEC_SIZE];
};

struct pr_flag {
    unsigned
        /* Identity: */
        fg_name:1,      /* Is fd_name set? */
        fg_uid:1,       /* Is fd_uid set? */
        fg_encrypt:1,   /* Is fd_encrypt set? */
        fg_type:1,      /* Is fd_type set? */
        fg_owner:1,     /* Is fd_owner set? */
        fg_nice:1,      /* Is fd_nice set? */
        fg_cprivs:1,    /* Is fd_sprivs set? */
        fg_sprivs:1,    /* Is fd_sprivs set? */
        fg_auditctl:1,  /* Is fd_auditctl set? */
        fg_auditdisp:1, /* Is fd_auditdisp set? */

        /* Password maintenance parameters: */
        fg_min:1,       /* Is fd_min set? */
        fg_maxlen:1,    /* Is fd_maxlen set? */
        fg_expire:1,    /* Is fd_expire set? */
        fg_lifetime:1,  /* Is fd_lifetime set? */
        fg_schange:1,   /* Is fd_schange set? */
        fg_uchange:1,   /* Is fd_fchange set? */
        fg_pswduser:1,  /* Is fd_pswduser set? */
        fg_pick_pwd:1,  /* Is fd_pick_pwd set? */
        fg_gen_pwd:1,   /* Is fd_gen_pwd set? */
        fg_restrict:1,  /* Is fd_restrict set? */
        fg_nullpw:1,    /* Is fd_nullpw set? */

        /* Login parameters: */
        fg_slogin:1,    /* Is fd_slogin set? */

```

```

        fg_suctty: 1,      /* is fd_suctty set ? */
        fg_unsuctty: 1,   /* is fd_unsuctty set ? */
        fg_ulogin:1,      /* Is fd_ulogin set? */
        fg_nlogins:1,     /* Is fd_nlogins set? */
        fg_max_tries:1,   /* Is fd_max_tries set? */
        fg_lock:1,        /* Is fd_lock set? */

        fg_standpswd:1,   /* Is fd_standpswd set? */
        fg_secclass:1;    /* Is fd_secclass set? */
};

```

Because these structures are declared in *<prot.h>*, it is not necessary to redeclare them.

When the **getprpwent** routine is first called, it returns a pointer to the first user **pr_passwd** structure in the database. Thereafter, it returns a pointer to the next user **pr_passwd** structure in the database. So, successive calls can be used to search the entire database. Note that entries without a corresponding entry in the */etc/passwd* file are skipped. The entries are scanned in the order they appear in */etc/passwd*.

getprpwent(S) truncates the encrypted password string (**u_pwd**) in the protected password entry, so that a comparison of the value returned by **bigcrypt(S)** (described on the **getpasswd(S)** manual page) succeeds if the password would have matched on the first (**u_pwseg** * 8) cleartext characters.

For the purposes of checking passwords, this is transparent to the application. To provide backward compatibility to applications and allow **u_pwseg** to be reduced and then increased again without losing information, it is important that the stored encrypted password (set using **putprpwnam(S)** when the password is changed) not be truncated.

So that the sequence of **getprpwent**, **putprpwnam** does not truncate the stored password when **u_pwseg** is smaller than the password in the entry, the operating system stores the original stored value of **u_pwd**. **putprpwnam** restores **u_pwd** to this saved value if the new value matches the truncated saved value.

The **getprpwuid** routine searches from the beginning of the database until a numerical user ID matching the argument *uid* is found. **getprpwuid** returns a pointer to the particular **pr_passwd** structure in which the argument *uid* was found.

Like **getprpwuid**, the **getprpwnam** routine searches from the beginning of the database file until a login name matching the argument *name* is found. **getprpwnam** returns a pointer to the particular **pr_passwd** structure in which the argument *name* was found.

The **setprpwent** routine resets the file pointer to the beginning of the protected password files to allow repeated searches. The **endprpwent** routine closes the protected password files when processing is complete.

The **putprpwnam** routine puts a new or replaced protected password entry *pr* with key *name* into the protected password database. If the `uflg.fg_name` structure member is 0, the requested entry is deleted from the protected password database. The **putprpwnam** routine locks the database for all update operations, and performs an **endprpwent** after the update or failed attempt.

Return value

Upon successful completion, the routines **getprpwent**, **getprpwuid**, and **getprpwnam** each returns a pointer to a `pr_passwd` structure. The **putprpwnam** routine returns a non-zero value on a successful substitution, deletion, or addition and a value of zero upon failure. If an end-of-file or an error is encountered on reading, these routines return a NULL pointer. No errors are defined for the **setprpwent** or **endprpwent** routines.

Diagnostics

Use `feof()` to test if a null return value is caused by an end-of-file or from a syntax error in the protected password database file. The `feof` macro is described in the **ferror(S)** manual page.

Warning

These routine fail if `set_auth_parameters` has not been called before using **getprpwent**. `set_auth_parameters` is described in the **identity(S)** manual page.

Note

All information is contained in a static area, so it must be copied if it is to be saved. Otherwise, it may be overwritten on subsequent calls to these routines.

These routines assume one name per user ID and one user ID per name. Otherwise, the sequential search loops between the first two instances of a multiple user ID.

Files

```
/etc/passwd
/tcb/files/auth/*/*
```


getprpwent(S)

See also

**authcap(F), authcap(S), bigcrypt(S), getpasswd(S), getprdfent(S),
getprfient(S), getprpwent(S), getprtcent(S), getpwent(S), identity(S)**

Standards conformance

**endprpwent, getprpwent, getprpwnam, getprpwuid, putprpwnam and set-
prpwent** are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

getprtcent, getprtcnam, setprtcent, endprtcent, putprtcnam

manipulate terminal control database entry

Syntax

```
cc ... -lprot -lx

#include <sys/types.h>
#include <sys/security.h>
#include <sys/audit.h>
#include <prot.h>

struct pr_term *getprtcent ( )

struct pr_term *getprtcnam (name)
char *name;

void setprtcent ( )

void endprtcent ( )

int putprtcnam (name, pr)
char *name;
struct pr_term *pr;
```

Description

getprtcent - gets pointer to next **pr_term** structure

getprtcnam - searches for login name matching name

setprtcent - rewinds terminal control file to allow repeated searches

endprtcent - close the terminal control file when processing is complete

putprtcnam - puts a new or replaced terminal control entry

The **getprtcent** and **getprtcnam** routines both return a pointer to an object with the following structure containing the broken-out fields of a line in the terminal control database. Only entries in the database dealing with users are scanned.

Each line in the database contains a `pr_term` structure, declared in the `<prot.h>` header file:

```

struct      pr_term {
    struct t_field ufld;
    struct t_flag uflg;
    struct t_field sfld;
    struct t_flag sflg;
};

struct      t_field {
    char      fd_devname[12]; /* Same size as utmp entry */
    ushort    fd_uid;         /* uid of last successful login */
    time_t    fd_slogin;      /* time stamp of      "      */
    ushort    fd_uuid;        /* uid of last unsuccessful login */
    time_t    fd_ulogin;      /* time stamp of      "      */
    ushort    fd_loutuid;     /* uid of last logout */
    time_t    fd_louttime;    /* time stamp of      "      */
    ushort    fd_nlogins;     /* consecutive failed attempts */
    ushort    fd_max_tries;    /* max unsuc login tries allowed */
    time_t    fd_logdelay;    /* delay between login tries */
    char      fd_label[AUTH_TLSSIZ]; /* terminal label */
    char      fd_lock;        /* terminal locked? */
    ushort    fd_login_timeout; /* login timeout in seconds */
};

struct      t_flag {
    unsigned
        fg_devname:1,        /* Is fd_devname set? */
        fg_uid:1,            /* Is fd_uid set? */
        fg_slogin:1,         /* Is fd_stime set? */
        fg_uuid:1,           /* Is fd_uuid set? */
        fg_ulogin:1,         /* Is fd_ftime set? */
        fg_loutuid:1,        /* Is fd_loutuid set? */
        fg_louttime:1,       /* Is fd_louttime set? */
        fg_nlogins:1,        /* Is fd_nlogins set? */
        fg_max_tries:1,      /* Is fd_max_tries set? */
        fg_logdelay:1,       /* Is fd_logdelay set? */
        fg_label:1,          /* Is fd_label set? */
        fg_lock:1,           /* Is fd_lock set? */
        fg_login_timeout:1;  /* Is fd_login_timeout valid? */
};

```

Because these structures are declared in `<prot.h>`, they need not be redeclared.

When the `getprtcnt` routine is first called, it returns a pointer to the first user `pr_term` structure in the database. Thereafter, it returns a pointer to the next `pr_term` structure in the database. So, successive calls can be used to search the entire database.

The `getprtcnam` routine searches from the beginning of the database file until a login name matching the argument *name* is found. `getprtcnam` returns a pointer to the particular `pr_term` structure in which the argument *name* was found.

The `setprtcnt` routine resets the file pointer to the beginning of the terminal control file to allow repeated searches. The `endprtcnt` routine closes the terminal control file when processing is complete.

The `putprtcnam` routine puts a new or replaced terminal control entry `pr` with key `name` into the terminal control file. If the `uflag.fg_name` member is 0, the requested entry is deleted from the terminal control database. The `putprtcnam` routine locks the database for all update operations, and performs an `endprtcnt` after the update or failed attempt.

Return value

Upon successful completion, the `getprtcnt` and `getprtcnam` routines each returns a pointer to a `pr_term` structure. The `putprtcnam` routine returns a non-zero value on a successful substitution, deletion, or addition and a value of zero upon failure. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer. No errors are defined for the `setprtcnt` or `endprtcnt` routines.

Diagnostics

Use `feof()` to test if a null return value is caused by an end-of-file or from a syntax error in the terminal control file. The `feof` macro is described in the `error(S)` manual page.

Note

All information is contained in a static area, so it must be copied if it is to be saved. Otherwise, it may be overwritten on subsequent calls to these routines.

Files

`/etc/auth/system/ttys`

See also

`authcap(F)`, `authcap(S)`, `getprdfent(S)`, `getprfient(S)`, `getprpwent(S)`

Standards conformance

`endprtcnt`, `getprtcnt`, `getprtcnam`, `putprtcnam` and `setprtcnt` are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

getpw

get user info from UID

Syntax

```
cc ... -lc

int getpw (uid, buf)
int uid;
char *buf;
```

Description

The **getpw** routine searches the */etc/passwd* file for a user ID number that equals the *uid* argument. If the user ID is found, **getpw** copies the line of the password file in which *uid* was found into the array to which the argument *buf* points. Upon a successful match, the **getpw** routine returns a value of zero. If the */etc/passwd* file does not exist, cannot be opened, or if *uid* does not match any ID entry in the password file, **getpw** fails and returns a non-zero value.

This routine is included only for compatibility with prior systems and should not be used; see **getpwent(S)** for routines to use instead.

Return value

Upon successful completion, the **getpw** routine returns a value of zero. **getpw** returns a non-zero value on error.

Files

/etc/passwd

See also

getpwent(S), **passwd(FP)**

Standards conformance

The **getpw** routine is an extension of AT&T System V provided by the Santa Cruz Operation.

getpwent, getpwuid, getpwnam, setpwent, endpwent, fgetpwent

get password file entry

Syntax

```
cc ... -lc

#include <pwd.h>

struct passwd *getpwent ( )

struct passwd *getpwuid (uid)
int uid;

struct passwd *getpwnam (name)
char *name;

void setpwent ( )

void endpwent ( )

struct passwd *fgetpwent (f)
FILE *f;
```

Description

getpwent - gets pointer to next passwd structure
getpwuid - searches for matching numerical user ID
getpwnam - searches for login name matching name
setpwent - rewinds password file to allow repeated searches
endpwent - closes password file when processing is complete
fgetpwent - gets pointer to next matching passwd structure

The **getpwent**, **getpwuid**, and **getpwnam** routines each returns a pointer to an object with a structure containing the broken-out fields of a line in the */etc/passwd* file, a **passwd** structure declared in the *<pwd.h>* header file. Refer to this header file for a listing of the **passwd** structure. Because the elements vary by the type of standards compliance that you are using, it is not included here. Because this structure is in *<pwd.h>*, it does not need to be redeclared. The meanings of the fields in the **passwd** structure are described in *passwd(FP)*.

When the **getpwent** routine is first called, it returns a pointer to the first **passwd** structure in the */etc/passwd* file. Thereafter, it returns a pointer to the next **passwd** structure in the file. So successive calls to **getpwent** can be used

to search the entire file.

The **getpwuid** routine searches from the beginning of the */etc/passwd* file until a numerical user ID matching the argument *uid* is found. **getpwuid** returns a pointer to the particular **passwd** structure in which the argument *uid* was found.

Like **getpwuid**, the **getpwnam** routine searches from the beginning of the */etc/passwd* file until a login name matching the argument *name* is found. **getpwnam** returns a pointer to the particular **passwd** structure in which the argument *name* was found.

The **setpwent** routine resets the file pointer to the beginning of the password file, */etc/passwd*, to allow repeated searches. The **endpwent** routine closes the password file, */etc/passwd*, when processing is complete.

The **fgetpwent** routine returns a pointer to the next **passwd** structure which matches the format of */etc/passwd* in the stream argument *f*.

Return value

Upon successful completion, the routines **getpwent**, **getpwuid**, **getpwnam**, and **fgetpwent** each return a pointer to a **passwd** structure. If an end-of-file or an error is encountered on reading, these routines return a NULL pointer. No errors are defined for the **setpwent** or **endpwent** routines.

Diagnostics

Use **feof()** to test if a null return value is caused by an end-of-file or from a syntax error in the password file. The **feof** macro is described on the **ferror(S)** manual page.

Note

All information is contained in a static area, so it must be copied if it is to be saved. Otherwise, it may be overwritten on subsequent calls to these routines.

Files

/etc/passwd

See also

feof(S), **getlogin(S)**, **getgrent(S)**, **passwd(FP)**

Standards conformance

endpwent, getpwent and setpwent conforms with:

AT&T SVID Issue 2.

getpwnam and getpwuid conforms with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

The **fgetpwent** routine has been withdrawn from XPG3.

(S)

gets, fgets

get a string from a stream

Syntax

```
cc ... -lc

#include <stdio.h>

char *gets (s)
char *s;

char *fgets (s, n, stream)
char *s;
int n;
FILE *stream;
```

Description

gets - reads characters from standard input stream

fgets - reads characters from stream in an array

The **gets** function reads characters from the standard input stream, *stdin*, into the array pointed to by *s*, until a new-line character is read or an end-of-file condition is encountered. The new-line character is discarded and the string is terminated with a null character. See warning below on the use of **gets**.

The **fgets** function reads characters from the *stream* into the array pointed to by *s*, until *n*-1 characters are read, or a new-line character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null character.

Warning

The **gets** function is unsafe to use. It does not limit the number of characters placed into the passed buffer, even though that buffer has some fixed size. For this reason, the **fgets** function should be used instead.

fgets is not, however, a direct substitute for **gets**. With **fgets**, as soon as the buffer is filled, the remainder of the line is left for the next **fgets**, while **gets** always reads the entire line even when it overflows the buffer.

Diagnostics

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a NULL pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a NULL pointer is returned. Otherwise *s* is returned.

See also

ferror(S), fopen(S), fread(S), getc(S), scanf(S), stdio(S)

Standards conformance

fgets and **gets** are conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

ANSI X3.159-1989 Programming Language — C;

Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

(S)

getspent, getspnam, setspent, endspent, fgetspent, lckpwn, ulckpwn

get shadow password file entry

Syntax

```
#include <shadow.h>

struct spwd *getspent (void)

struct spwd *getspnam (name)
char *name;

int lckpwn (void)

int ulckpwn (void)

void setspent (void)

void endspent (void)

struct spwd *fgetspent (fp)
FILE *fp;
```

Description

getspent - get shadow password entry
getspnam - get matching login name shadow password entry
setspent - rewind shadow password file to allow repeated searches
endspent - closes shadow password file when processing is complete
fgetspent - get pointer to next **spwd** structure
lckpwn - lock the shadow password file
ulckpwn - unlock the shadow password file

The **getspent** and **getspnam** routines each return a pointer to an object with the following structure containing the broken-out fields of a line in the */etc/shadow* file. Each line in the file contains a "shadow password" structure, declared in the *<shadow.h>* header file:

```
struct spwd {
    char *sp_namp;
    char *sp_pwdp;
    long sp_lstchg;
    long sp_min;
    long sp_max;
};
```

The **getspent** routine, when first called, returns a pointer to the first **spwd** structure in the file; thereafter, it returns a pointer to the next **spwd** structure in the file, so successive calls can be used to search the entire file.

The **getspnam** routine searches from the beginning of the file until a login name matching *name* is found, and returns a pointer to the particular structure in which it was found. The **getspent** and **getspnam** routines populate the "sp_min", "sp_max" or "sp_lstchg" field with -1 if the corresponding field in */etc/shadow* is empty. If an end-of-file or an error is encountered on reading, or there is a format error in the file, these functions return a NULL pointer and set *errno* to *EINVAL*.

/etc/pwd.lock is the lock file. It is used to coordinate modification access to the password files */etc/passwd* and */etc/shadow*. **lckpwn** and **ulckpwn** are routines that are used to gain modification access to the password files, through the lock file. A process first uses **lckpwn** to lock the lock file thereby gaining exclusive rights to modify the */etc/passwd* or */etc/shadow* password file. Upon completing modifications, a process should release the lock on the lock file via **ulckpwn**. This mechanism prevents simultaneous modification of the password files.

The **lckpwn** routine attempts to lock the file */etc/pwd.lock*. If file */etc/pwd.lock* is already locked, **lckpwn** tries for 15 seconds to lock the file. If **lckpwn** is unsuccessful, then **lckpwn** returns a -1. If **lckpwn** succeeds to lock the file */etc/pwd.lock* within 15 seconds, then a return code other than -1 is returned.

The **ulckpwn** routine attempts to unlock the file */etc/pwd.lock*. If successful, **ulckpwn** returns a 0. If the unlocking failed, as in the case that file */etc/pwd.lock* was not locked initially, then **ulckpwn** returns a -1.

A call to the **setspent** routine has the effect of rewinding the shadow password file to allow repeated searches. The **endspent** routine may be called to close the shadow password file when processing is complete.

The **fgetspent** routine returns a pointer to the next **spwd** structure in the stream *fp*, which matches the format of */etc/shadow*.

getspent(S)

Files

/etc/shadow, /etc/passwd, /etc/pwd.lock

Diagnostics

A NULL pointer is returned on EOF or error.

Notes

If a program not otherwise using standard I/O uses this routine, the size of the program will increase more than might be expected.

These routines are provided for compatibility purposes only, and may not function correctly on all systems. We recommend that the `getprpwent(S)` routines be used as an alternative.

All information is contained in a static area, so it must be copied if it is to be saved.

See also

`putspent(S)`, `getpwent(S)`, `putpwent(S)`, `getprpwent(S)`

Standards conformance

The `getspent`, `getspnam`, `setspent`, `endspent`, `fgetspent`, `lckpwordf`, and `ulckpwordf` routines are conformant with AT&T SVID Issue 3.

getuid, geteuid, getgid, getegid

get real user, effective user, real group, and effective group IDs

Syntax

```
cc ... -lc  
  
unsigned short getuid (  
  
unsigned short geteuid (  
  
unsigned short getgid (  
  
unsigned short getegid (  

```

Description

The **getuid** routine returns the real user ID of the calling process.
The **geteuid** routine returns the effective user ID of the calling process.
The **getgid** routine returns the real group ID of the calling process.
The **getegid** routine returns the effective group ID of the calling process.

Diagnostics

These routines are always successful and therefore no errors defined.

See also

setuid(S)

Standards conformance

getegid, geteuid, getgid and **getuid** are conformant with:
AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

getut: endutent, getutent, getutid, getutline, pututline, setutent, utmpname

access utmp file entry

Syntax

```
cc ... -lc

#include <utmp.h>

void endutent ()

struct utmp *getutent ()

struct utmp *getutid (id)
struct utmp *id;

struct utmp *getutline (line)
struct utmp *line;

struct utmp *pututline (utmp)
struct utmp *utmp;

void setutent ()

void utmpname (file)
char *file;
```

Description

endutent - closes the currently open file
getutent - reads in the next entry from a **utmp** -like file
getutid - searches forward in the **utmp** -like file
getutline - searches for **LOGIN_PROCESS** or **USER_PROCESS** entry
pututline - writes out supplied **utmp** structure to **utmp** -like file
setutent - resets input stream
utmpname - allows the user to change the name of the file examined

The **getutent**, **getutid**, **getutline**, and **pututline** routines each return a pointer to a structure of the following type:

```

struct utmp {
    char    ut_user[8];           /* User login name */
    char    ut_id[4];            /* /etc/inittab id (usually line #) */
    char    ut_line[12];         /* device name (console, lnxx) */
    short   ut_pid;              /* process id */
    short   ut_type;             /* type of entry */
    struct   exit_status {
        short e_termination;     /* Process termination status */
        short e_exit;            /* Process exit status */
    } ut_exit;                  /* The exit status of a process
                                marked as DEAD_PROCESS. */
    time_t  ut_time;            /* time entry was made */
};

```

The **getutent** routine reads in the next **utmp** structure entry from a **utmp**-like file. If the file is not already open, it opens it. If it cannot open a file or if it reaches the end of the file, it fails.

The **getutid** routine searches forward from the current point in the **utmp** file until it finds an entry with a **ut_type** matching the one in the argument **id** if the type specified is **BOOT_TIME**, **NEW_TIME**, **OLD_TIME**, or **RUN_LVL**. If the type specified in the argument **id** is **DEAD_PROCESS**, **INIT_PROCESS**, **LOGIN_PROCESS**, or **USER_PROCESS** then **getutid** returns a pointer to the first entry whose **ut_type** and **ut_id** matches those specified in **id**. If the end of file is reached without a match, it fails.

The **getutline** routine searches forward from the current point in the **utmp** file until it finds an entry of the type **LOGIN_PROCESS** or **USER_PROCESS**, which also has a **ut_line** string matching the **ut_line** string in the argument **line**. If the end of file is reached without a match, it fails.

The **pututline** routine writes out the supplied **utmp** structure specified by the argument **utmp**. into the **utmp**-like file. It uses **getutid** to search forward for the proper place to overwrite if it finds that it is not already at the proper place. It is expected that normally the user of **pututline** has searched for the proper entry using one of the other **getut** routines. If so, **pututline** does not search. If **pututline** does not find a matching slot for the new entry, it adds a new entry to the end of the file.

The **setutent** routine resets the input stream to the beginning of the file. This should be done before each search for a new entry if it is desired that the entire file be examined.

The **endutent** routine closes the currently open file after processing.

The **utmpname** routine allows the user to change the name of the file examined, from */etc/utmp* to any other file. It is most often expected that this other file is */etc/wtmp*. If the file does not exist, this is not made apparent until the first attempt to reference the file. This is because **utmpname** does not open the new file. It just closes the old file if it is currently open and saves the new file name.

Return value

Upon successful completion, the **getutent**, **getutid**, **getutline**, and **pututline** routines return a pointer to a **utmp** structure. These routines return a NULL pointer upon failure. There are no errors or return values defined for the **setutent**, **endutent**, and **utmpname** routines.

Diagnostics

The **getutent**, **getutid**, **getutline**, and **pututline** routines will return the NULL pointer if the **utmp**-like file is of improper format, cannot be open, cannot be created, or if the user does not have the proper permissions to read or write to it. For the first three routines, a NULL pointer is also return if the appropriate entry cannot be found.

Notes

The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made. Each call to either **getutid** or **getutline** routine examines the static structure before performing more I/O. If the contents of the static structure match what it is searching for, it looks no further. For this reason, to use **getutline** to search for multiple occurrences, it would be necessary to zero out the static after each success, or **getutline** would just return the same pointer over and over again.

There is one exception to the rule about removing the structure before further reads are done. The implicit read done by **pututline** (if it finds that it is not already at the correct place in the file) does not hurt the contents of the static structure returned by the **getutent**, **getutid**, or **getutline** routines, if the user has just modified those contents and passed the pointer back to **pututline**.

These routines use buffered standard I/O for input, but **pututline** uses an unbuffered non-standard write to avoid race conditions between processes trying to modify the **utmp** and **wtmp** files.

Files

/etc/utmp
/etc/wtmp

See also

ttyslot(S), utmp(F)

Standards conformance

endutent, getutent, getutid, getutline, pututline, setutent and utmpname conform with:

AT&T SVID Issue 2.

hsearch, hcreate, hdestroy

manage hash search tables

Syntax

```
cc ... -lc

#include <search.h>

ENTRY *hsearch (item, action)
ENTRY item;
ACTION action;

int hcreate (nel)
int nel;

void hdestroy ( )
```

Description

hsearch - searches a hash table

hcreate - allocates sufficient space for the table

hdestroy - destroys the search table

The **hsearch** function is a hash-table search routine generalized from Knuth (6.4) Algorithm D. It returns a pointer into a hash table indicating the location at which an entry can be found. **item** is a structure of type **ENTRY** (defined in the *<search.h>* header file) containing two pointers: **item.key** points to the comparison key, and **item.data** points to any other data to be associated with that key. (Pointers to types other than character should be cast to pointer-to-character.) **action** is a member of an enumeration type **ACTION** indicating the disposition of the entry if it cannot be found in the table. **ENTER** indicates that the item should be inserted in the table at an appropriate point. **FIND** indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a **NULL** pointer.

hcreate allocates sufficient space for the table and must be called before **hsearch** is used. **nel** is an estimate of the maximum number of entries that the table contains. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

hdestroy destroys the search table and may be followed by another call to **hcreate**.

Diagnostics

The **hsearch** function returns a **NULL** pointer if either the action is **FIND** and the item could not be found, or the action is **ENTER** and the table is full.

hcreate returns zero if it cannot allocate sufficient space for the table.

Note

Only one hash search table may be active at any given time.

Warning

hsearch and **hcreate** use **malloc(S)** to allocate space.

See also

bsearch(S), **lsearch(S)**, **malloc(S)**, **string(S)**, **tsearch(S)**

Standards conformance

hcreate, **hdestroy** and **hsearch** are conformant with:

AT&T SVID Issue 2;

and X/Open Portability Guide, Issue 3, 1989.

Example

The following example reads in strings followed by two numbers and stores them in a hash table, discarding duplicates. It then reads in strings and finds the matching entry in the hash table and prints it out.

```
#include <stdio.h>
#include <search.h>

struct info {
    int age, room;
};
#define NUM_EMPL    5000    /* # of elements in search table */

main( )
{
    /* space to store strings */
    char string_space[NUM_EMPL*20];
    /* space to store employee info */
    struct info info_space[NUM_EMPL];
    /* next avail space in string_space */
    char *str_ptr = string_space;
    /* next avail space in info_space */
    struct info *info_ptr = info_space;
    ENTRY item, *found_item, *hsearch( );
    /* name to look for in table */
    char name_to_find[30];
    int i = 0;

    /* create table */
    (void) hcreate(NUM_EMPL);
    while (scanf("%s%d%d", str_ptr, &info_ptr->age,
        &info_ptr->room) != EOF && i++ < NUM_EMPL) {
        /* put info in structure, and structure in item */
        item.key = str_ptr;
        item.data = (char *)info_ptr;
        str_ptr += strlen(str_ptr) + 1;
        info_ptr++;
        /* put item into table */
        (void) hsearch(item, ENTER);
    }
}
```

```
/* access table */
item.key = name_to_find;
while (scanf("%s", item.key) != EOF) {
    if ((found_item = hsearch(item, FIND)) != NULL) {
        /* if item is in the table */
        (void)printf("found %s, age = %d, room = %d\n",
            found_item->key,
            ((struct info *)found_item->data)->age,
            ((struct info *)found_item->data)->room);
    } else {
        (void)printf("no such employee %s\n",
            name_to_find);
    }
}
}
```

(S)

hypot

euclidean distance function

Syntax

```
cc ... -lm

#include <math.h>

double hypot (x, y)
double x, y;
```

Description

hypot returns
 $\text{sqrt}(x * x + y * y)$,
taking precautions against unwarranted overflows.

Diagnostics

When the correct value would overflow, **hypot** returns **HUGE** and sets **errno** to **ERANGE**.

These error-handling procedures may be changed with the function **matherr(S)**.

See also

matherr(S)

Standards conformance

hypot is conformant with:
AT&T SVID Issue 2;
and X/Open Portability Guide, Issue 3, 1989.

identity

get or check uids or gids from program start

Syntax

```
cc ... -lprot

#include <sys/types.h>
#include <sys/security.h>
#include <sys/audit.h>
#include <prot.h>

void check_auth_parameters ( )

int is_starting_egid (gid)
ushort gid;

int is_starting_euid (uid)
ushort uid;

int is_starting_luid (uid)
ushort uid;

int is_starting_ruid (uid)
ushort uid;

int is_starting_rgid (gid)
ushort gid;

void set_auth_parameters (argc, argv)
int argc;
char *argv[];

ushort starting_egid ( )

ushort starting_euid ( )

ushort starting_luid ( )

ushort starting_rgid ( )

ushort starting_ruid ( )
```

Description

check_auth_parameters - verify that **set_auth_parameters** has been previously invoked

is_starting_egid - check current effective GID against retained ID

is_starting_euid - check current effective UID against retained ID

is_starting_luid - check current login UID against retained ID

is_starting_ruid - check current real UID against retained ID

is_starting_rgid - check current real GID against retained ID

set_auth_parameters - retain IDs for future lookup

starting_egid - returns the effective GID

starting_euid - returns the effective UID

starting_luid - returns the login UID

starting_rgid - returns the real GID

starting_ruid - returns the real UID

These routines provide a way to recall the IDs of a process at the time the program started. They are useful when interrogating the invoking environment of a program after any **setuid(S)** or **setgid(S)** calls have been made so that the original environment can be captured.

starting_luid returns the login UID for the process as it was set in the beginning of the program. The login UID is the immutable stamp for the process and accurately denotes the account under which the session is being run, regardless of subsequent **setuid(S)** calls. **starting_ruid** returns the real UID for the process as it was set in the beginning of the program. Similarly, **starting_euid** returns the effective UID, **starting_rgid** returns the real GID, and **starting_egid** returns the effective GID. These IDs may not be the same as those returned by **getluid(S)**, **getuid(S)**, **geteuid(S)**, **getgid(S)**, or **getegid(S)**, respectively, because intervening calls to **setluid(S)**, **setuid(S)** or **setgid(S)** can change the latter set while the former set remains the same.

The routine **is_starting_luid** returns 1 if the argument is the same as the login UID at the time when **set_auth_parameters** was invoked, and 0 otherwise. Similarly, **is_starting_ruid** returns 1 if the argument is the same as the real UID at the time when **set_auth_parameters** was invoked, and 0 otherwise; **is_starting_euid** returns 1 if the argument is the same as the effective UID at the time when **set_auth_parameters** was invoked, and 0 otherwise; **is_starting_rgid** returns 1 if the argument is the same as the real GID at the time when **set_auth_parameters** was invoked, and 0 otherwise; and **is_starting_egid** returns 1 if the argument is the same as the effective GID at the time when **set_auth_parameters** was invoked, and 0 otherwise.

The **set_auth_parameters** routine is used to retain the IDs for future lookup. It also tests the kernel to see if the C2 security features have been loaded. If not, the program exists with an error message. It should be called first in a program or there is a chance that it captures an environment different from the program beginning. The two arguments are the argument count and vector with which the program was called. **check_auth_parameters** verifies that **set_auth_parameters** has been previously invoked. If not, the program exits. If so, nothing happens.

Notes

These routines only work as advertised when `set_auth_parameters` is called as the first item in `main()`.

`set_auth_parameters` resets the umask in the calling process to 077.

See also

`getegid(S)`, `geteuid(S)`, `getgid(S)`, `getuid(S)`

Standards conformance

`identity` is an extension of AT&T System V provided by the Santa Cruz Operation.

(S)

ioctl

IO control command

Syntax

cc ... -lc

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

Description

The **ioctl** system call performs a variety of control functions on devices and **STREAMS**. For non-**STREAMS** files, the functions performed by this call are device-specific control functions. The arguments *request* and *arg* are passed to the file designated by *fildes* and are interpreted by the device driver. Basic input/output functions in **STREAMS** are performed with the **read(S)** and **write(S)** system calls.

fildes is an open file descriptor that refers to a device. *request* selects the control function to be performed and depends on the device being addressed. *arg* represents additional information that is needed by this specific device to perform the requested function. The data type of *arg* depends upon the particular control request, but it is either an integer or a pointer to a device-specific data structure.

In addition to device-specific and **STREAMS** functions, generic functions are provided by more than one device driver, for example, the general terminal interface (see **termio(M)**).

The **ioctl** system call fails for any type of file if one or more of the following is true:

- [EACCES] Future error.
- [EBADF] *fildes* is not a valid open file descriptor.
- [EINTR] A signal was caught during the **ioctl** system call.
- [ENOTTY] *fildes* is not associated with a device driver that accepts control functions.

The **ioctl** system call also fails if the device driver detects an error. In this case, the error is passed through **ioctl** without change to the caller. A particular driver might not have all of the following error cases. Other requests to device drivers fails if one or more of the following is true:

- [EFAULT] *request* requires a data transfer to or from a buffer pointed to by *arg*, but some part of the buffer is outside the process's allocated space.

- [EINVAL] *request* or *arg* is not valid for this device.
- [EIO] Some physical I/O error has occurred.
- [ENOLINK] *files* is on a remote machine and the link to that machine is no longer active.
- [ENXIO] The *request* and *arg* are valid for this device driver, but the service requested cannot be performed on this particular subdevice.

STREAMS errors are described in `streamio(STR)`.

Diagnostics

Upon successful completion, the value returned depends upon the device control function, but must be a non-negative integer. Otherwise, a value of -1 is returned, and `errno` is set to indicate the error.

See also

`termio(M)`

Standards conformance

`ioctl` is conformant with:

AT&T SVID Issue 2;
and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

isaddindex

add an index to an ISAM file

Syntax

```
cc ... -lisam

isaddindex (isfd, keydesc)
int isfd;
struct keydesc *keydesc;
```

Description

The **isaddindex** routine builds an index for an open ISAM file. You can use the following arguments with this routine:

Argument	Description
isfd	File descriptor of the ISAM data file being indexed.
keydesc	Structure defining the index.

The **isaddindex** routine executes only if the file has been opened in exclusive mode.

The number of parts of the index cannot exceed **NPARTS**. The sum of the lengths of the parts of a key may not exceed **MAXKEYSIZE**. **NPARTS** and **MAXKEYSIZE** are defined in the *isam.h* header file.

An ISAM file can have any number of indexes built on it.

This call returns an error if the index is defined not to allow duplicates and the data contains duplicates.

Example

```
isaddindex(bookfd, &key) ;
```

See also

isdelindex(S)

Standards conformance

isaddindex is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

isbuild

create an ISAM file

Syntax

```
cc ... -lisam
```

```
isbuild (filename, recordlength, keydesc, mode)
char *filename;
int recordlength;
struct keydesc *keydesc;
int mode;
```

Description

The **isbuild** function creates a new ISAM file. It creates and initializes the necessary files on the disk to hold the data and index files. You can use the following arguments with this routine:

Argument	Description
filename	Null-terminated character string identifying the name of data file being created. Maximum of 10 characters for the filename. The filename extension should not be specified.
recordlength	Number of bytes in a record. Equals the sum of the field lengths.
keydesc	Pointer to the structure defining the primary index of the file.
mode	Access mode in effect while the file is open as a result of isbuild . Arithmetic sum of a lock mode value and a read/write mode value. The values are described later in this section.

If **isbuild** executes successfully, it returns a file descriptor of the new ISAM file. The file remains open for further processing. Use the **isclose** function to close the file.

The **isbuild** function must include an index structure for the primary index. However, if you set `k_nparts` in the structure `keydesc` to zero, then there is effectively no primary index. Additional indexes may be added with the **isad-index** function.

The mode parameter specifies the lock mode and the read/write mode in effect while the file is open as a result of **isbuild**. The value of the mode parameter is the arithmetic sum of a lock mode value and a read/write mode value.

The lock mode values are defined in *isam.h* as:

Lock Mode	Description
ISAUTOLOCK	automatic record lock
ISMANULOCK	manual lock
ISEXCLLOCK	exclusive file lock

These modes have the following effects:

- When **ISAUTOLOCK** is used, each record is locked when it is accessed for read, write, delete, or update, and unlocked after the next function call is made.
- When **ISMANULOCK** is used, you can perform both optional file locking and record locking. Files may be locked and unlocked multiple times by using the **islock** and **isunlock** functions. You can perform record-locking by **isread** with an **ISLOCK** option. Locked records may be released with an **isrelease** call.
- When **ISEXCLLOCK** is used, the entire file is locked. Other users cannot read, write, or open the file. The file is unlocked when it is closed with **isclose**.

The read/write mode values are defined in *isam.h* as:

Build Option	Description
ISOUTPUT	open for writing only
ISINOUT	open for reading and writing

On success, **isbuild** returns a non-negative value. Otherwise, it returns -1 and sets **iserrno** to indicate the error.

Example

This example creates an ISAM file named *books*, with record length of 52 and a primary key contained in *key*.

```
isbuild("books", 52, &key, ISINOUT + ISEXCLLOCK) ;
```

See also

iserase(S), **isopen(S)**, **isrename(S)**

Standards conformance

isbuild is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

isclose

close an ISAM file

Syntax

```
cc ... -lisam

isclose (isfd)
int isfd;
```

Description

The **isclose** function closes an ISAM file that was opened with **isopen** or **isbuild**. You can use the following argument with this routine:

Argument	Description
isfd	File descriptor identifying the ISAM file to be closed.

All locks on the file are released when it is closed.

Note

ISAM files must be closed when you are through with them. Not doing so may corrupt files in certain environments.

Return Value

On success, **isclose** returns 0. Otherwise, it returns -1 and sets **iserrno** to indicate the error.

Example

```
isclose(bookfd) ;
```

See also

isbuild(S), **isopen(S)**

Standards conformance

isclose is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

isconv: lddbl, ldfloat, ldint, ldlong, stdbl, stfloat, stint, stlong

ISAM data conversion tools

Syntax

```
#include <isam.h>

double lddbl (p)
char *p;

float ldfloat (p)
char *p;

short ldint (p)
char *p;

long ldlong (p)
char *p;

int stdbl (d, p)
double d;
char *p;

int stfloat (f, p)
float f;
char *p;

short stint (i, p)
integer i;
char *p;

int stlong (l, p)
long l;
char *p;
```

Description

lddbl - convert ISAM integer to double
ldfloat - convert ISAM integer to float
ldint - convert ISAM integer to short
ldlong - convert ISAM integer to long
stdbl - convert unaligned ISAM aligned double
stfloat - convert unaligned ISAM aligned float
stint - convert unaligned ISAM aligned short

stlong - convert unaligned ISAM aligned long

The **isconv** routines provide conversion tools for use with ISAM files.

The bytes that make up an integer are always stored in the ISAM file in the order you would expect: most-significant byte first and least-significant byte last. Some machines, however, store integers in the reverse order. Four routines are available to convert integers between ISAM and machine format, and four to convert between the unaligned ISAM format and the aligned machine format.

Converting integers

ldint(p) returns a short integer in machine format. Here *p* is a char pointer to the first byte of the format **INTTYPE**.

ldlong(p) returns a long integer in machine format. Here *p* is a char pointer to the first byte of the format **LONGTYPE**.

stint(i,p) converts a machine-format short integer *i* to **INTTYPE** and stores it at location *p*. Here *p* is a char pointer to the first byte of **INTTYPE**.

stlong(l,p) converts a machine-format long integer *l* to **LONGTYPE** and stores it at location *p*. Here *p* is a char pointer to the first byte of **LONGTYPE**.

ldint and **ldlong** might be used when ISAM records are written to a temporary storage area for further processing by a C program. Any integers being passed from the buffer to the executing program are first converted to machine format with the **ldint** and **ldlong** routines. **stint** and **stlong** might be used when integers produced or processed by an executing C program are to be written to an ISAM file. The **stint** and **stlong** routines convert the machine-format integers to ISAM format and place them in a buffer in preparation for writing them to the ISAM file.

Converting floating point

Integers in ISAM format need not align along word boundaries as must integers in some machine format. On certain machines, floating-point numbers must generally be aligned on word boundaries. ISAM-format numbers need not be. Four routines are available to convert between the unaligned ISAM format and the aligned machine format.

ldfloat(p) returns a float in machine format. Here *p* is a char pointer to the first byte of the format **FLOATTYPE**. **lddbl(p)** returns a double in machine format. Here *p* is a char pointer to the first byte of the format **DOUBLETYPE**.

stfloat(f,p) converts a machine-format float to **FLOATTYPE** and stores it at location *p*. Here *p* is a char pointer to the first byte of **FLOATTYPE**.

stdbl(d,p) converts a machine-format double *d* to **DOUBLETYPE** and stores it at location *p*. Here *p* is a char pointer to the first byte of **DOUBLETYPE**.

ldfloat and **lddbl** can be used when ISAM records are accessed for processing. Any floating-point numbers being passed from the buffer to the executing program must be first converted to machine format with the **ldfloat** and **lddbl** routines. **stfloat** and **stddbl** might be used when floating-point numbers produced or processed by an executing C program are to be written to an ISAM file. The **stfloat** and **stdbl** routines convert the machine-format numbers to ISAM format and place them in a buffer in preparation for writing them to the ISAM file.

See also

isbuild(S)

Standards conformance

lddbl, **ldfloat**, **ldint**, **ldlong**, **stdbl**, **stfloat** and **stint** are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

isdelcurr

delete the current record

Syntax

```
cc ... -lisam

isdelcurr (isfd)
int isfd;
```

Description

The **isdelcurr** function deletes the current record in an ISAM file. It differs from **isdelete**, which deletes a record identified by a key value. You can use the following argument with this routine:

Argument	Description
isfd	File descriptor identifying a currently open ISAM file.

All index entries associated with the current record are also deleted.

You must open the file with the **ISINOUT** option. Otherwise the operation fails. On success, **isdelcurr** returns 0. Otherwise, it returns -1 and sets **iserrno** to indicate the error.

Example

```
isdelcurr(bookfd) ;
```

See also

isdelete(S), **isdelrec(S)**

Standards conformance

isdelcurr is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

isdelete

delete record specified by primary key

Syntax

```
cc ... -lisam

isdelete (isfd, record)
int isfd;
char *record;
```

Description

The **isdelete** function deletes a record from a file using a key value. You can use the following arguments with this routine:

Argument	Description
isfd	File descriptor identifying a currently open ISAM file.
record	Search value pertaining to the current key of the record to be deleted.

If duplicate primary keys are allowed, then **isdelete** should not be used and the **isread** and **isdelcurr** functions should be used instead.

Any index entries associated with the deleted record are also deleted.

The **isrecnum** pointer is set to the record just deleted, and the current record position is unchanged.

You must open the file with the **ISINOUT** option. Otherwise, the operation fails. On success, **isdelete** returns 0. Otherwise, it returns -1 and sets **iserrno** to indicate the error.

Example

```
isdelete(bookfd, buffer) ;
```

See also

isdelcurr(S), **isdelrec(S)**

Standards conformance

isdelete is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

isdelindex

delete an index

Syntax

```
cc ... -lisam

isdelindex (isfd, keydesc)
int isfd;
struct keydesc *keydesc;
```

Description

The **isdelindex** function deletes an existing index from an ISAM file. You can use the following arguments with this routine:

Argument	Description
isfd	File descriptor identifying the ISAM file whose index is being deleted.
keydesc	Name of the structure defining the index to be deleted.

Any index may be removed except the primary index.

The file must be opened in exclusive mode or the operation fails. On success, **isdelindex** returns 0. Otherwise, it returns -1 and sets **iserrno** to indicate the error.

Example

```
isdelindex(bookfd, &key) ;
```

See also

isaddindex(S)

Standards conformance

isdelindex is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

isdelrec

delete record specified by record number

Syntax

```
cc ... -lisam

isdelrec (isfd, recnum)
int isfd;
long recnum;
```

Description

The **isdelrec** function deletes a record based on its record number. The record number is a previously obtained **isrecnum** value.

Argument	Description
isfd	File descriptor identifying the ISAM file containing the record to be deleted.
recnum	The record number of the record to be deleted.

The **isdelrec** function sets the **isrecnum** pointer to *recnum*, and leaves the current record position unchanged.

On success, **isdelrec** returns 0. Otherwise, it returns -1 and sets **iserrno** to indicate the error.

Example

```
isdelrec(bookfd, isrecnum) ;
```

See also

isdelcurr(S), isdelete(S)

Standards conformance

isdelrec is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

iserase

delete an entire ISAM file

Syntax

```
cc ... -lisam
iserase (filename)
char *filename;
```

Description

The `iserase` function deletes an entire ISAM file from the disk. Both the index and data files are deleted. If the files are not found, an error is returned and `iserrno` is set to -1. Use this function with great care.

You can use the following argument with this routine:

Argument	Description
filename	Null-terminated character string identifying the name of the ISAM file to be deleted.

Example

```
iserase(*books*) ;
```

See also

`isbuild(S)`

Standards conformance

`iserase` is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

isindexinfo

obtain information about an ISAM file

Syntax

```
cc ... -lisam

isindexinfo (isfd, buffer, number)
int isfd;
struct ( keydesc | dictinfo ) *buffer;
int number;
```

Description

The information returned by the `isindexinfo` call can be either dictionary information or index information, depending on the *number* parameter. You can use the following arguments with this routine:

Argument	Description
<code>isfd</code>	File descriptor identifying the ISAM file for which information is to be obtained.
<code>buffer</code>	Name of the buffer with structure <code>keydesc</code> or <code>dictinfo</code> into which the information is placed.
<code>keydesc</code>	Structure defined in <i>isam.h</i> to hold index information. Use only if <i>number</i> is nonzero.
<code>dictinfo</code>	Structure defined in <i>isam.h</i> to hold file dictionary information. Use only if <i>number</i> is zero.
<code>number</code>	Non-negative integer determining what information is obtained. If zero, then the file's dictionary information is put into the buffer. Otherwise, information about the index with that number is put into the buffer.

Dictionary information about a file includes the number of indexes defined, the record size, the primary index size, and the number of records in the file. Dictionary information is placed in the buffer by the call when the *number* parameter is set to zero. The buffer should use the `dictinfo` structure defined in *isam.h* as:

```
struct dictinfo {
    short di_nkeys ;           /* number of keys defined */
    short di_recsz ;          /* data record size */
    short di_idxsz ;          /* index key size */
    long di_nrecords ;        /* number of records in file */
};
```

Index information includes flags (duplicates, compression), number of parts in the key, and the length, type, and location of each key part. Index information is placed in the buffer by the call when the *number* parameter is set to the

number of the index. The buffer should use the **keydesc** structure defined in *isam.h* as:

```

struct keypart {
    short kp_start ;           /* starting byte of key part */
    short kp_leng ;           /* length in bytes */
    short kp_type ;           /* type of key part */
};

struct keydesc {
    short k_flags ;           /* flags */
    short k_nparts ;          /* number of parts in key */
    struct keypart k_part[NPARTS] ; /* each key part */
};

```

The number of an index can change when indexes are added or deleted. Use the information in **dictinfo** about the number of indexes to check all indexes.

On success, **isindexinfo** returns 0. Otherwise, it returns -1 and sets **iserrno** to indicate the error.

Example

This example obtains information about the ISAM file:

```

struct dictinfo dictbuffer
isindexinfo(bookfd, dictbuffer, 0) ;

```

This example obtains information about the second index:

```

struct keydesc keybuffer
isindexinfo(bookfd, keybuffer, 2) ;

```

See also

isaddindex(S), isbuild(S)

Standards conformance

isindexinfo is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

islock

lock an ISAM file

Syntax

```
cc ... -lisam

islock (isfd)
int isfd;
```

Description

The **islock** function locks an entire ISAM file so that other processes cannot write to the file. You can use the following argument with this routine:

Argument	Description
isfd	File descriptor identifying the ISAM file to be locked.

For a file to be locked with **islock**, the file must have been opened using the ISMANULOCK value of the mode parameter in **isopen** or **isbuild**.

A file locked with **islock** can be unlocked with the **isunlock** function. On success, **islock** returns 0. Otherwise, it returns -1 and sets **iserrno** to indicate the error.

Example

```
islock(bookfd) ;
```

See also

isunlock(S)

Standards conformance

islock is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

isnan, isnand, isnanf

test for a floating point NaN (Not-A-Number)

Syntax

```
cc ... -lm

#include <nan.h>

int isnan (x)
double x;

cc ... -lc

#include <ieeefp.h>

int isnand (dsrc)
double dsrc;

int isnanf (fsrc)
float fsrc;
```

Description

isnan - test double for Not-a-Number (NaN)

isnand - test double for Not-a-Number (NaN) according to IEEE Standard

isnanf - test float for Not-a-Number (NaN) according to IEEE Standard

The **isnand** and **isnanf** functions return true (1) if the argument *dsrc* or *fsrc* is a NaN; otherwise they return false (0).

Neither routine generates any exception, even for signaling NaNs.

The routine **isnand** is implemented in **libc**, the Standard C Library. **isnanf** is implemented as a macro included in *<ieeefp.h>*.

The **isnan** routine in **libm** calls the NaN macro defined in *<nan.h>*.

isnan(S)

See also

fpgetround(S)

Standards conformance

isnan is conformant with:

Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

isopen

open an ISAM file

Syntax

```
cc ... -lisam

isopen(filename, mode)
char *filename;
int mode;
```

Description

The **isopen** function opens a data file for processing. The call returns the file descriptor that is to be used by subsequent ISAM calls on that file. Use the **isclose** function to close the file.

You can use the following arguments with this routine:

Argument	Description
filename	Null-terminated character string identifying the name of the ISAM file to be opened.
mode	are described below.

The current index is set to the primary key.

The current record pointer is set to the first record in the file based on the order in the primary index. It can be set to the first record in another index by using the **isstart** function.

The mode parameter specifies the lock mode and the read/write mode in effect while the file is open. The value of the mode parameter is the arithmetic sum of a lock mode value and a read/write mode value.

The lock mode values are defined in *isam.h* as:

Lock Mode	Description
ISAUTOLOCK	automatic record lock
ISMANULOCK	manual record lock
ISEXCLLOCK	exclusive isam file lock
ISSHARLOCK	shared isam file lock

These modes have the following effects:

- When **ISAUTOLOCK** is used, each record is locked when it is read, and unlocked after the next function call is made.

- When ISMANULOCK is used, the user must manually lock and unlock each record to perform any operation on that record. A record can be locked by using the ISLOCK mode in the `isread` function call, and unlocked with the `isrelease` function call.
- When ISEXCLLOCK is used, the entire file is locked. Other users can neither read from nor to the file. The file is automatically unlocked when it is closed with `isclose`.
- When ISSHARLOCK is used, the entire file is locked with a shared lock. Other users can read the file and place shared locks on it but may not change it. The file is automatically unlocked when it is closed with `isclose`.

The read/write mode values are defined in *isam.h* as:

Access Modes	Description
ISINPUT	open for reading only
ISOUTPUT	open for writing only
ISINOUT	open for reading and writing

On success, `isopen` returns a non-negative integer. Otherwise, it returns -1 and sets `iserrno` to indicate the error.

Example

```
isopen( "books", ISAUTOLOCK + ISINOUT ) ;
```

See also

`isbuild(S)`, `isclose(S)`

Standards conformance

`isopen` is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

isread

read records in an ISAM file

Syntax

```
cc ... -lisam

isread(isfd, record, mode)
int isfd;
char *record;
int mode;
```

Description

The **isread** function reads a record from an ISAM file into a buffer for further processing. You can use the following arguments with this routine:

Argument	Description
isfd	File descriptor identifying the ISAM file to be read. The file must already be open.
record	Buffer holding the record after the call is executed. For random accesses, record is filled with the search value before the call is executed.
mode	Integer identifying which record is read and whether random or sequential access is used. Also can be used to lock the record manually. The possible values are described below.

If **isread** executes successfully, the current record position pointer and **isrecnum** are set to the record just read.

The mode parameter determines which record is read. The possible values depend on whether you want sequential or random access.

The values of mode for sequential access are specified in the *isam.h* header file as:

Read Mode	Description
ISFIRST	first record
ISLAST	last record
ISNEXT	next record
ISPREV	previous record
ISCURR	current record

During random access, the function compares a value stored in the record parameter to the corresponding data in the records. The mode parameter determines the comparison logic as follows:

Read Mode	Description

ISEQUAL	record equal to search value
ISGREAT	record greater than search value
ISGTEQ	record greater than or equal to search value

The search value is placed in the appropriate byte positions in the record buffer. The record that first satisfies the condition (in the order of the index currently in use) is then copied to the record buffer.

A special use of *isread* allows access by a previously set *isrecnum*. First, *isstart* is called with the *k_nparts* member of the *keydesc* parameter set to zero. This turns off the primary index and causes the file to be read in physical order. Then *isread* is called with the mode parameter set to *ISEQUAL*. This causes *isread* to read the record located at the *isrecnum* position.

The record to be read can also be locked by specifying the *ISLOCK* option for manual locks or *ISSLOCK* for shared locks. The lock is effective only if the *isopen* function was called with a mode parameter that included *ISMANULOCK*. The record stays locked until an *isrelease* call is issued on the file.

On success, *isread* returns 0. Otherwise, it returns -1 and sets *iserrno* to indicate the error.

Example

This example reads a record that matches the search value contained in the buffer:

```
isread( bookfd, buffer, ISEQUAL ) ;
```

See also

isstart(S)

Standards conformance

isread is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

isrelease

unlocks all manually locked records in a file

Syntax

```
cc ... -lisam

isrelease(isfd)
int isfd;
```

Description

The **isrelease** function is used to unlock all records that have been manually locked in a file. You can use the following argument with this routine:

Argument	Description
isfd	File descriptor identifying the ISAM file whose records are being unlocked.

A record is manually locked when:

- The data file was opened with a mode parameter that included **ISMANU-LOCK**, and
- The **isread** function call that read the record included **ISLOCK** in its mode parameter.

On success, **isrelease** returns 0. Otherwise, it returns -1 and sets **iserrno** to indicate the error.

Example

```
isrelease( bookfd ) ;
```

See also

isread(S), **islock(S)**

Standards conformance

isrelease is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

isrename

rename an ISAM file

Syntax

cc ... -lisam

```
isrename(oldname, newname)
char *oldname;
char *newname;
```

Description

The **isrename** function renames both the data and index files, specified by oldname to newname. You can use the following arguments with this routine:

Argument	Description
oldname	Null-terminated string identifying the existing ISAM file that is to be renamed.
newname	Null-terminated string that is the new name of the file.

When you use **isrename**, both the old and new filenames must be specified without any extensions and both files must be in the same directory. Otherwise, the operation can not succeed.

If the file with the oldname does not exist or if the file with newname already exists, the operation can not succeed.

On success, **isrename** returns a positive integer. Otherwise, it returns -1 and sets **iserrno** to indicate the error.

Example

This example changes the name of the *books* file to *titles*:

```
isrename( "books", "titles" );
```

Standards conformance

isrename is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

isrewcurr

rewrite the current record

Syntax

```
cc ... -lisam

isrewcurr(isfd, record)
int isfd;
char *record;
```

Description

The **isrewcurr** function overwrites the current record in an ISAM file. You can use the following arguments with this routine:

Argument	Description
isfd	File descriptor identifying the ISAM file containing the record to be updated.
record	Character buffer whose contents are written into the file.

Each index defined on the data file, including the primary index, is also updated as needed. Unchanged keys are not rewritten.

The record buffer should be loaded with the new data before the call is issued. Care must be taken to ensure that fields are located properly in the buffer.

On return, **isrecnum** is set to the rewritten record and the current record position is unchanged. On success, **isrewcurr** returns a positive integer. Otherwise, it returns -1 and sets **iserrno** to indicate the error.

Example

```
isrewcurr( bookfd, buffer ) ;
```

See also

isrewrite(S), isrewrec(S)

Standards conformance

isrewcurr is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

isrewrec

rewrite the record indicated by record number

Syntax

```
cc ... -lisam

isrewrec(isfd, recnum, record)
int isfd;
long recnum;
char *record;
```

Description

The **isrewrec** function rewrites a record specified by its record number. The record number *recnum* is a previously obtained **isrecnum** value. You can use the following arguments with this routine:

Argument	Description
isfd	File descriptor identifying the ISAM file containing the record to be updated.
recnum	Record number of the record to be updated.
record	Character buffer whose contents are written into the file.

The record buffer should be loaded with the new data before the call is issued. Care must be taken to ensure that fields are located properly in the buffer.

Each index defined on the data file, including the primary index, is updated as needed. On return, **isrecnum** is set to *recnum*, and the current record position is unchanged.

On success, **isrewrec** returns a positive integer. Otherwise, it returns -1 and sets **iserrno** to indicate the error.

Example

```
isrewrec( bookfd, isrecnum, buffer ) ;
```

See also

isrewrite(S), **isrewcurr(S)**

Standards conformance

isrewrec is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

isrewrite

rewrite a record identified by its primary key

Syntax

```
cc ...-lisam

isrewrite(isfd, record)
int isfd;
char *record;
```

Description

The **isrewrite** function rewrites a record specified by its primary key when the primary key does not allow duplicate values. You can use the following arguments with this routine:

Argument	Description
isfd	File descriptor identifying the ISAM file containing the record to be updated.
record	Character buffer whose contents are written into the file. The primary key field identifies the record to be updated.

The record buffer should be loaded with the new data before the **isrewrite** call is issued. The primary key field must match that of the record to be updated. Care must be taken to ensure that fields are located properly in the buffer.

Each index defined on the data file, including the primary index, is updated as needed.

On return, **isrecnum** is set to the rewritten record, and the current record position is unchanged.

On success, **isrewrite** returns a positive integer. Otherwise, it returns -1 and sets **iserrno** to indicate the error.

Example

```
isrewrite( bookfd, buffer ) ;
```

See also

isrewrec(S), isrewcurr(S)

Standards conformance

isrewrite is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

issetunique

set the value of a unique identifier

Syntax

```
cc ... -lisam

issetunique (isfd, uniqueid)
int isfd;
long uniqueid
```

Description

The **issetunique** routine sets the value of a stored unique identifier to the one specified. The identifier can then be incremented as desired. The unique identifier is a long contained in *uniqueid*.

You can use the following arguments with this routine:

Argument	Description
isfd	File descriptor of an open ISAM file.
uniqueid	The long integer specifying the unique identifier.

On success, **issetunique** returns 0. Otherwise, it returns -1 and sets **iserrno** to indicate the error.

See also

isuniqueid(S)

Standards conformance

issetunique is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

isstart

select an index and locate a record

Syntax

cc ... -lisam

```
isstart (isfd, keydesc, length, record, mode)
int isfd;
struct keydesc *keydesc;
int length;
char *record;
int mode;
```

Description

The **isstart** function selects an index for subsequent use. It also locates, but does not read, a record based on the selected index. Use the **isread** function to read this and subsequent records using this index.

You can use the following arguments with this routine:

Argument	Description
isfd	File descriptor identifying the ISAM file that has the index to be selected.
keydesc	Structure describing the index to be selected. The index must already exist.
length	Number of bytes of the index key to use during the search for the record. Zero indicates that the whole key is to be used.
record	Character buffer holding the key value to be used in locating the record if mode is one of ISEQUAL , ISGREAT , or ISGTEQ .
mode	Integer that determines the logic for searching the index for the desired record. The allowed values are described below.

If the mode is set to **ISEQUAL**, **ISGREAT**, or **ISGTEQ**, then prior to calling the function a key value is placed in the record buffer. The value's position in the record buffer must match the position specified in *keydesc*. The length parameter determines how many bytes of the key are used in the search.

The mode parameter tells `isstart` how to find the desired record. The allowed modes as defined in `isam.h` are:

Start Mode	Description
ISFIRST	first record
ISLAST	last record
ISEQUAL	first exact match of search value
ISGREAT	first record greater than search value
ISGTEQ	first record greater than or equal to search value

If the desired record is known to be the first or last in the current ordering, then use the `ISFIRST` or `ISLAST` mode value. In these cases, the record and length parameters are not used.

If `ISFIRST` is used, then the pointer is positioned just before the first record in the selected index order. You should then call the `isread` function using the `ISNEXT` mode parameter to read the first record in the selected order.

If `ISLAST` is used, then the pointer is positioned just after the last record in the selected index order. You should then call the `isread` function using the `ISPREV` mode parameter to read the last record in the selected order.

If the position of the desired record is not first or last, then use one of the `ISEQUAL`, `ISGREAT`, or `ISGTEQ` mode values (and a key value in record) to locate the desired record.

If `ISEQUAL` is used, the function locates the first record with a key value matching the record parameter.

If `ISGREAT` is used, the function locates the first record whose key value is greater than the key field in the record parameter.

If `ISGTEQ` is used, the function locates the first record whose key value is greater than or equal to the key field in the record parameter.

A special case of `isstart` can be used to access a file in physical order. To do this, set the value of the `k_nparts` member of `keydesc` to zero. `isstart` ignores all indexes and position on the first physical record. See `isread(S)` for more information.

If a match is found, then the current record position pointer and `isrecnum` are set to this record. If no match is found, then `iserrno` is set to `ENOREC` and a -1 error code is returned.

Example

This example makes *titlekey* the current index, uses the entire key, and locates a record matching the search value in the buffer:

```
isstart( bookfd, titlekey, 0, buffer, ISEQUAL ) ;
```

See also

isread(S)

Standards conformance

isstart is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

isuniqueid

return a unique identifier

Syntax

```
cc ... -lisam

isuniqueid (isfd, uniqueid)
int isfd;
long *uniqueid
```

Description

The **isuniqueid** routine generates a unique identifier for an ISAM file previously opened by **isopen** or **isbuild**, and identified by *isfd*. The unique identifier is a pointer to a long integer contained in *uniqueid*.

You can use the following arguments with this routine:

Argument	Description
isfd	File descriptor of an open ISAM file.
uniqueid	Pointer to a long integer to receive the unique identifier.

On success, **isuniqueid** returns 0. Otherwise, it returns -1 and sets **iserrno** to indicate the error.

See also

issetunique(S)

Standards conformance

isuniqueid is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

isunlock

unlock an ISAM file

Syntax

```
cc ... -lisam

isunlock (isfd)
int isfd;
```

Description

The **isunlock** function unlocks a file that was locked with **islock**. (The file must have been opened using the **ISMANULOCK** value of the mode parameter in **isopen** or **isbuild**.)

You can use the following argument with this routine:

Argument	Description
isfd	File descriptor identifying the ISAM file to be unlocked.

On success, **isunlock** returns 0. Otherwise, it returns -1 and the error is set to **iserrno**. Once you use **isunlock**, other processes may read and write into the file.

Example

```
isunlock( bookfd ) ;
```

See also

isclose(S), islock(S), isopen(S)

Standards conformance

isunlock is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

iswrcurr

write a new record and make it current

Syntax

```
cc ... -lisam

iswrcurr (isfd, record)
int isfd;
char *record;
```

Description

The **iswrcurr** function writes a new record into an ISAM file. You can use the following arguments with this routine:

Argument	Description
isfd	File descriptor identifying the ISAM file in which the record is being written.
record	Character buffer holding the data to be written into the file.

When you use **iswrcurr**, care must be taken to ensure values are located properly in their fields in the record buffer prior to calling the function.

If the function executes successfully, the current record position pointer and **isrecnum** are both set to the new record.

Any indexes are updated as needed.

If the record contains any duplicate values in fields that are defined with **NODUPS** attribute, an error is produced and no record is written.

On success, **iswrcurr** returns 0. Otherwise, it returns -1 and sets **iserrno** to indicate the error.

Example

```
iswrcurr( bookfd, buffer ) ;
```

See also

iswrite(S)

Standards conformance

iswrcurr is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

iswrite

write a new record into an ISAM file

Syntax

```
cc ... -lisam

iswrite (isfd, record)
int isfd;
char *record;
```

Description

The **iswrite** function writes a new record into an ISAM file. You can use the following arguments with this routine:

Argument	Description
isfd	File descriptor identifying the ISAM file in which the record is being written.
record	Character buffer holding the data to be written into the file.

When using **iswrite**, care must be taken to ensure values are located properly in their fields in the record buffer prior to calling the function.

If the function executes successfully, **isrecnum** is set to the new record. The current record position pointer is left unchanged.

Any indexes are updated as needed.

If the record contains any duplicate values in fields that are defined with **NODUPS** attribute, an error is produced and no record is written.

On success, **iswrite** returns 0. Otherwise, it returns -1 and sets **iserrno** to indicate the error.

Example

```
iswrite( bookfd, buffer ) ;
```

See also

iswrcurr(S)

Standards conformance

iswrite is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

item

CRT menu-item routines

Syntax

```
cc ... -lmenu -lcurses

#include <menu.h>

ITEM *new_item(n, d)
char *n, *d;

int free_item(i)
ITEM * i;

char *item_name(i)
ITEM * i;

char *item_description(i)
ITEM * i;

int set_item_opts(i, o)
ITEM * item;
OPTIONS o;

OPTIONS item_opts(i)
ITEM * i;

int item_opts_on(item, opts)
ITEM * item;
OPTIONS opts;

int item_opts_off(item, opts)
ITEM * item;
OPTIONS opts;

int set_item_value(i, c)
ITEM * item;
int *c;

int item_value(i)
ITEM * item;
```

```

int set_item_userptr(i, n)
ITEM * item;
char *n;

char *item_userptr(i)
ITEM * i;

int item_count(m)
MENU *m;

int item_visible(i)
ITEM * item;

```

Description

These routines allow you to create, display, and access items. Menus can be displayed on any display device supported by the curses(S) library. Once you compile a program that contains a “#include” statement for the ITEM header file *menu.h*, you should link it with the ITEM and curses library routines.

Functions

new_item (*n*, *d*) creates a new item with name *n* and description *d*. It returns a pointer to the new item. In general, you should store these item field pointers in an array.

free_item () frees the storage allocated for the given item. Once an item is freed, you can no longer connect it to a menu.

item_name (*i*) returns a pointer to the given item’s name.

item_description (*i*) returns a pointer to the given item’s description.

set_item_opts (*i*, *o*) turns on the named option(s) for the item and turns off its remaining options, if any. Options are boolean values. Currently, there is one item option O_SELECTABLE which enables your end-user to select the item. The initial current default is to have O_SELECTABLE on for every item.

item_opts (*i*) returns the given item’s option(s) setting. To set options, you can apply boolean operators to the value returned by **item_opts** () and let the result be the second argument to **set_item_opts** ().

item_opts_on (*item*, *opts*) turns on the named options for the item.

item_opts_off (*item*, *opts*) turns off the named options for the item.

Unlike single-valued menus, multi-valued menus enable your end-user to select one or more items from a menu. **set_item_value** (*i*, *c*) sets the given item’s select value—TRUE (selected) or FALSE (not selected). To make a menu multi-valued, you use **set_menu_opts** () or **menu_opts_off** () to turn off option O_ONEVALUE. **set_item_value** () may be used only with multi-valued menus.

item_value (*i*) returns the select value of the given item, either TRUE (selected) or FALSE (unselected).

Every item has an associated user pointer that you can use to store pertinent information. **set_item_userptr** (*i*, *n*) sets the item's user pointer.

item_userptr (*i*) returns the item's user pointer.

item_count (*m*) returns the number of items in the given menu.

A menu item is visible if it currently appears in the subwindow of the posted menu to which it is connected. If an item is visible, **item_visible** (*i*) returns TRUE. If not, it returns FALSE.

Diagnostics

The following values are returned by one or more routines that return an integer.

E_BAD_ARGUMENT	an incorrect argument was passed to the routine
E_BAD_STATE	routine called from an inappropriate routine
E_CONNECTED	one or more items are connected to another menu
E_NO_MATCH	no match occurred
E_NO_ROOM	menu does not fit within its subwindow
E_NOT_CONNECTED	no items are associated with the menu
E_NOT_POSTED	menu has not yet been posted
E_NOT_SELECTABLE	item cannot be selected
E_OK	routine returned normally
E_POSTED	menu is already posted
E_REQUEST_DENIED	menu driver could not process the request
E_SYSTEM_ERROR	system error
E_UNKNOWN_COMMAND	unrecognizable request was given to the driver

See also

curses(S), field(S), fieldtype(S), form(S), menu(S), panel(S), tam(S)

Standards conformance

item is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

kill

send a signal to a process or a group of processes

Syntax

```
cc ... -lc

#include <sys/types.h>
#include <signal.h>

int kill (pid, signo)
pid_t pid;
int signo;
```

Description

The **kill** system call sends a signal to a process or a group of processes. The process or group of processes to which the signal is to be sent is specified by *pid*. The signal that is to be sent is specified by *signo* and is either one from the list given in **signal(S)** or 0. If *signo* is 0 (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

The real or effective user ID of the sending process must match the real or effective user ID of the receiving process, unless the effective user ID of the sending process is super user.

The processes with a process ID of 0 and a process ID of 1 are special processes and are referred to below as **proc0** and **proc1**, respectively.

If *pid* is greater than zero, *signo* is sent to the process whose process ID is equal to *pid*. *pid* may equal 1.

If *pid* is 0, *signo* is sent to all processes excluding **proc0** and **proc1** whose process group ID is equal to the process group ID of the sender.

If *pid* is -1 and the effective user ID of the sender is not super user, *signo* is sent to all processes excluding **proc0** and **proc1** whose real user ID is equal to the effective user ID of the sender.

If *pid* is -1 and the effective user ID of the sender is super user, *signo* is sent to all processes excluding **proc0** and **proc1**.

If *pid* is negative but not -1, *signo* is sent to all processes whose process group ID is equal to the absolute value of *pid*.

The **kill** system call fails and no signal is sent if one or more of the following is true:

[EINVAL]	1. <i>signo</i> is not a valid signal number. 2. <i>signo</i> is SIGKILL and <i>pid</i> is 1 (proc1).
[EPERM]	The user ID of the sending process is not super user, and its real or effective user ID does not match the real or effective user ID of the receiving process.
[ESRCH]	No process can be found corresponding to that specified by <i>pid</i> .

Diagnostics

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

See also

getpid(S), kill(C), setpggrp(S), signal(S), sigset(S)

Standards conformance

kill is conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

l3tol, ltol3

convert between 3-byte integers and long integers

Syntax

```
cc ... -lc

void l3tol (lp, cp, n)
long *lp;
char *cp;
int n;

void ltol3 (cp, lp, n)
char *cp;
long *lp;
int n;
```

Description

l3tol - converts three-byte integers to long integers

ltol3 - converts long integers to three-byte integers

The **l3tol** function converts a list of *n* three-byte integers packed into a character string pointed to by *cp* into a list of long integers pointed to by *lp*.

ltol3 performs the reverse conversion from long integers (*lp*) to three-byte integers (*cp*).

These functions are useful for file-system maintenance where the block numbers are three bytes long.

Note

Because of possible differences in byte ordering, the numerical values of the long integers are machine-dependent.

See also

fs(F)

Standards conformance

l3tol and **ltol3** are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

labs

converts to absolute value

Syntax

```
cc ... -lc

#include <stdlib.h>

long labs (n)
long n;
```

Description

The **labs** function produces the absolute value of its long-integer argument *n*.

Return value

The **labs** function returns the absolute value of its argument. There is no error return.

Example

```
#include <stdlib.h>
#include <stdio.h>

main()
{
    long x,y;

    x = -41567L;
    y = labs(x);
    printf("The labs (%ld) = %ld \n", x, y);
}
```

This program uses **labs** to get and display the absolute value of -41,567.

See also

abs(S), **fabs(S)**

Standards conformance

labs is conformant with:
ANSI X3.159-1989 Programming Language — C.

ldahread

read the archive header of a member of an archive file

Syntax

```
cc ... -lld

#include <filehdr.h>
#include <ldfcn.h>

int ldahread (ldptr, arhead)
LDFILE *ldptr;
ARCHDR *arhead;
```

Description

If **TYPE(ldptr)** is the archive file magic number, **ldahread** reads the archive header of the common object file currently associated with **ldptr** into the area of memory beginning at **arhead**.

ldahread returns **SUCCESS** or **FAILURE**. **ldahread** fails if **TYPE(ldptr)** does not represent an archive file, or if it cannot read the archive header.

See also

ldclose(S), **ldfcn(FP)**

Standards conformance

ldahread is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

ldclose, ldaclose

close a common object file and free memory

Syntax

```
cc ... -lld

#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldclose (ldptr)
LDFILE *ldptr;

int ldaclose (ldptr)
LDFILE *ldptr;
```

Description

The **ldopen** and **ldclose** functions are designed to provide uniform access to both simple object files and object files that are members of archive files. Thus an archive of common object files can be processed as if it were a series of simple common object files.

If **TYPE** (*ldptr*) does not represent an archive file, **ldclose** closes the file and frees the memory allocated to the **LDFILE** structure associated with *ldptr*. If **TYPE** (*ldptr*) is the magic number of an archive file, and if there are any more files in the archive, **ldclose** reinitializes **OFFSET** (*ldptr*) to the file address of the next archive member and returns **FAILURE**. The **LDFILE** structure is prepared for a subsequent **ldopen**(S). In all other cases, **ldclose** returns **SUCCESS**.

ldaclose closes the file and frees the memory allocated to the **LDFILE** structure associated with *ldptr* regardless of the value of **TYPE** (*ldptr*). **ldaclose** always returns **SUCCESS**. The function is often used in conjunction with **ldaopen**.

The program must be loaded with the object file access routine library

See also

ldfcn(FP), **ldopen**(S)

Standards conformance

ldclose and **ldaclose** are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

ldfhread

read the file header of a common object file

Syntax

```
cc ... -lld

#include <filehdr.h>
#include <ldfcn.h>

int ldfhread (ldptr, filehead)
LDFILE *ldptr;
FILHDR *filehead;
```

Description

The **ldfhread** function reads the file header of the common object file currently associated with *ldptr* into the area of memory beginning at *filehead*.

ldfhread returns **SUCCESS** or **FAILURE**. **ldfhread** fails if it cannot read the file header.

In most cases the use of **ldfhread** can be avoided by using the macro **HEADER(*ldptr*)** defined in *ldfcn.h* (see *ldfcn(FP)*). The information in any field, *fieldname*, of the file header may be accessed using **HEADER(*ldptr*),*fieldname***.

See also

ldfcn(FP), **ldahread(S)**

Standards conformance

ldfhread is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

ldgetname

retrieve symbol name for common object file symbol table entry

Syntax

```
cc ... -lld

#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

char *ldgetname (ldptr, symbol)
LDFILE *ldptr;
SYMENT *symbol;
```

Description

The **ldgetname** function returns a pointer to the name associated with *symbol* as a string. The string is contained in a static buffer local to **ldgetname** that is overwritten by each call to **ldgetname**, and therefore must be copied by the caller if the name is to be saved.

The **ldgetname** function can be used to retrieve names from object files without any backward compatibility problems. The **ldgetname** function returns NULL (defined in *stdio.h*) for an object file if the name cannot be retrieved. This situation can occur:

- if the “string table” cannot be found,
- if not enough memory can be allocated for the string table,
- if the string table appears not to be a string table (for example, if an auxiliary entry is handed to **ldgetname** that looks like a reference to a name in a nonexistent string table), or
- if the name’s offset into the string table is past the end of the string table.

Typically, **ldgetname** is called immediately after a successful call to **ldtbread** to retrieve the name associated with the symbol table entry filled by **ldtbread**.

See also

ldfcn(FP), **ldtbindex(S)**, **ldtbread(S)**, **ldtbseek(S)**

Standards conformance

ldgetname is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

ldiv

divides long integers

Syntax

```
cc ... -lc

#include <stdlib.h>

struct ldiv_t
{
    long quot;    /* Quotient */
    long rem;     /* Remainder */
}

ldiv(numerator, denominator)
long numerator;
long denominator;
```

Description

The `ldiv` routine divides the *numerator* by the *denominator*, computing the quotient and the remainder. The sign of the quotient is the same as that of the mathematical quotient. Its absolute value is the largest integer which is less than the absolute value of the mathematical quotient. If the denominator is zero, the program terminates with an error message.

The `ldiv` function is similar to the `div` function, the difference being that the arguments and the members of the returned structure are all of type `long int`.

Return value

The `ldiv` function returns a structure of type `ldiv_t`, comprising both the quotient and the remainder. The structure is defined in *stdlib.h*.

Example

```
#include <stdlib.h>
#include <math.h>

main(argc, argv)
int argc;
char **argv;
{
    long int x,y;
    ldiv_t div_result;

    x = atol(argv[1]);
    y = atol(argv[2]);
    printf("x is %ld, y is %ld\n", x, y);

    div_result = ldiv(x,y);
    printf("The quotient is %ld, and the remainder is %ld\n",
        div_result.quot, div_result.rem);
}
```

This program takes two long integers as command line arguments and displays the results of the integer division.

See also

div(S)

Standards conformance

ldiv is conformant with:

ANSI X3.159-1989 Programming Language — C.

ldlread, ldlnit, ldllitem

manipulate line number entries of a common object file function

Syntax

```
cc ... -lld

#include <filehdr.h>
#include <linenum.h>
#include <ldfcn.h>

int ldlread (ldptr, fcnindx, linenum, linent)
LDFILE *ldptr;
long fcnindx;
unsigned short linenum;
LINENO *linent;

int ldlnit (ldptr, fcnindx)
LDFILE *ldptr;
long fcnindx;

int ldllitem (ldptr, linenum, linent)
LDFILE *ldptr;
unsigned short linenum;
LINENO *linent;
```

Description

ldlread - begins its search with the line number

ldlnit - reads the entry with the smallest line number

ldllitem - retrieves a series of line number entries

The **ldlread** function searches the line number entries of the common object file currently associated with *ldptr*. The **ldlread** function begins its search with the line number entry for the beginning of a function and confines its search to the line numbers associated with a single function. The function is identified by *fcnindx*, the index of its entry in the object file symbol table. The **ldlread** function reads the entry with the smallest line number equal to or greater than *linenum* into the memory beginning at *linent*.

The **ldlnit** and **ldllitem** functions together perform exactly the same function as **ldlread**. After an initial call to **ldlread** or **ldlnit**, **ldllitem** may be used to retrieve a series of line number entries associated with a single function. **ldlnit** simply locates the line number entries for the function identified by *fcnindx*. **ldllitem** finds and reads the entry with the smallest line number equal to or greater than *linenum* into the memory beginning at *linent*.

The **ldlread**, **ldlinit**, and **ldlitem** functions each return either **SUCCESS** or **FAILURE**. **ldlread** fails if there are no line number entries in the object file, if *fcnindx* does not index a function entry in the symbol table, or if it finds no line number equal to or greater than *linenum*. **ldlinit** fails if there are no line number entries in the object file or if *fcnindx* does not index a function entry in the symbol table. **ldlitem** fails if it finds no line number equal to or greater than *linenum*.

The programs must be loaded with the object file access routine library *libld.a*.

See also

ldfcn(FP), **ldlseek(S)**, **ldrseek(S)**, **ldtbindex(S)**

Standards conformance

ldlread, **ldlinit** and **ldlitem** are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

ldlseek, ldnlseek

seek to line number entries of a section of a common object file

Syntax

```
cc ... -lld

#include <filehdr.h>
#include <ldfcn.h>

int ldlseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnlseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

Description

ldlseek - seek to line number entries of section specified by **sectindx**

ldnlseek - seek to line number entries of section specified by **sectname**

The **ldlseek** function seeks to the line number entries of the section specified by **sectindx** of the common object file currently associated with **ldptr**.

The **ldnlseek** function seeks to the line number entries of the section specified by **sectname**.

The **ldlseek** and **ldnlseek** functions return **SUCCESS** or **FAILURE**. **ldlseek** fails if **sectindx** is greater than the number of sections in the object file; **ldnlseek** fails if there is no section name corresponding with ***sectname**. Either function fails if the specified section has no line number entries or if it cannot seek to the specified line number entries.

Note that the first section has an index of one.

See also

ldfcn(FP), **ldlread(S)**, **ldsseek(S)**

Standards conformance

ldlseek and **ldnlseek** are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

ldohseek

seek to the optional file header of a common object file

Syntax

```
cc ... -lld

#include <filehdr.h>
#include <ldfcn.h>

int ldohseek (ldptr)
LDFILE *ldptr;
```

Description

The **ldohseek** function seeks to the optional file header of the common object file currently associated with *ldptr*.

The **ldohseek** function returns **SUCCESS** or **FAILURE**. **ldohseek** fails if the object file has no optional header or if it cannot seek to the optional header.

See also

ldfcn(FP), **ldahread(S)**

Standards conformance

ldohseek is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

ldopen, ldaopen

open a common object file for reading

Syntax

```
cc ... -lld

#include <filehdr.h>
#include <ldfcn.h>

LDFILE *ldopen (filename, ldptr)
char *filename;
LDFILE *ldptr;

LDFILE *ldaopen (filename, oldptr)
char *filename;
LDFILE *oldptr;
```

Description

ldopen - returns a pointer to **LDFILE** structure

ldaopen - allocates and initializes a new **LDFILE** structure

The **ldopen** and **ldclose** functions provide uniform access to both simple object files and object files that are members of archive files. Thus an archive of common object files can be processed as if it were a series of simple common object files.

If *ldptr* has the value **NULL** then **ldopen** opens *filename* and allocates and initializes the **LDFILE** structure, and returns a pointer to the structure to the calling program.

If *ldptr* is valid and if **TYPE**(*ldptr*) is the archive magic number, **ldopen** reinitializes the **LDFILE** structure for the next archive member of *filename*.

The **ldopen** and **ldclose(S)** functions work in concert. **ldclose** returns **FAILURE** only when **TYPE**(*ldptr*) is the archive magic number and there is another file in the archive to be processed. Only then should **ldopen** be called with the current value of *ldptr*. In all other cases, in particular whenever a new *filename* is opened, **ldopen** should be called with a **NULL** *ldptr* argument.

If the value of *oldptr* is not **NULL**, **ldaopen** opens *filename* anew and allocates and initializes a new **LDFILE** structure, copying the **TYPE**, **OFFSET**, and **HEADER** fields from *oldptr*. **ldaopen** returns a pointer to the new **LDFILE** structure. This new pointer is independent of the old pointer, *oldptr*. The two pointers may be used concurrently to read separate parts of the object file. For example, one pointer may be used to step sequentially through the relocation

information, while the other is used to read indexed symbol table entries.

Both **ldopen** and **ldaopen** open *filename* for reading. Both functions return **NULL** if *filename* cannot be opened, or if memory for the **LDFILE** structure cannot be allocated. A successful open does not ensure that the given file is a common object file or an archived object file.

Example

The following is an example for the use of **ldopen** and **ldclose(S)**.

```
/* for each filename to be processed */

ldptr = NULL;
do
{
    if ((ldptr = ldopen(filename, ldptr)) != NULL )
    {
        /* check magic number */
        /* process the file */
    }
} while (ldclose(ldptr) == FAILURE );
```

See also

ldclose(S), ldfcn(FP)

Standards conformance

ldopen and **ldaopen** are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

allocldptr, freeldptr, vldldptr

ldptr structure usage routines

Syntax

```
cc ... -lld

#include <ldfcn.h>

LDFILE * allocldptr ()

int freeldptr (ldptr)
LDFILE *ldptr;

int vldldptr (ldptr)
LDFILE *ldptr;
```

Description

allocldptr - allocate ldptr structure and add to linked list

freeldptr - free allocated ldptr structure from linked list

vldldptr - verify ldptr structure on linked list

The **allocldptr** routine allocates an **LDFILE** structure in memory and attaches it to the end of a linked list of such structures. **allocldptr** returns a pointer to the newly allocated **LDFILE** structure.

freeldptr takes an instance of the **LDFILE** structure pointed to by *ldptr* and deletes it from the linked list.

vldldptr determines if the **LDFILE** structure pointed to by *ldptr* is in the linked list and returns **SUCCESS** if the structure is in the linked list.

Return values

allocldptr returns a pointer to the newly allocated **LDFILE** structure on success; otherwise, **NULL** is returned.

freeldptr and **vldldptr** both return **SUCCESS** on success; otherwise, they return **FAILURE**.

Standards conformance

allocldptr, **freeldptr**, and **vldldptr** are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

ldrseek, ldnrseek

seek to relocation entries of a section of a common object file

Syntax

```
cc ... -lld

#include <filehdr.h>
#include <ldfcn.h>

int ldrseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnrseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

Description

ldrseek - seeks relocation entries of section specified by *sectname*

ldnrseek - seeks relocation entries of section specified by *sectname*

The **ldrseek** function seeks to the relocation entries of the section specified by *sectindx* of the common object file currently associated with *ldptr*.

The **ldnrseek** function seeks to the relocation entries of the section specified by *sectname*.

The **ldrseek** and **ldnrseek** functions return **SUCCESS** or **FAILURE**. **ldrseek** fails if *sectindx* is greater than the number of sections in the object file; **ldnrseek** fails if there is no section name corresponding with *sectname*. Either function fails if the specified section has no relocation entries or if it cannot seek to the specified relocation entries.

Note that the first section has an index of *one*.

See also

ldfcn(FP), **ldshread(S)**, **ldsseek(S)**

Standards conformance

ldrseek and **ldnrseek** are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

ldshread, ldnshread

read an indexed/named section header of a common object file

Syntax

```
cc ... -lld

#include <filehdr.h>
#include <scnhdr.h>
#include <ldfcn.h>

int ldshread (ldptr, sectindx, secthead)
LDFILE *ldptr;
unsigned short sectindx;
SCNHDR *secthead;

int ldnshread (ldptr, sectname, secthead)
LDFILE *ldptr;
char *sectname;
SCNHDR *secthead;
```

Description

ldshread - reads section header specified by *sectindx* into memory

ldnshread - reads section header specified by *sectname* into memory

The **ldshread** function reads the section header specified by *sectindx* of the common object file currently associated with *ldptr* into the area of memory beginning at *secthead*.

The **ldnshread** function reads the section header specified by *sectname* into the area of memory beginning at *secthead*.

The **ldshread** and **ldnshread** functions return **SUCCESS** or **FAILURE**. **ldshread** fails if *sectindx* is greater than the number of sections in the object file; **ldnshread** fails if there is no section name corresponding with *sectname*. Either function fails if it cannot read the specified section header.

Note that the first section header has an index of one.

See also

ldfcn(FP), **ldsseek(S)**

Standards conformance

ldshread and **ldnshread** are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

ldsseek, ldsseek

seek to an indexed/named section of a common object file

Syntax

```
cc ... -lld

#include <filehdr.h>
#include <ldfcn.h>

int ldsseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldsseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

Description

ldsseek - seeks to the section specified by *sectindx*

ldsseek - seeks to the section specified by *sectname*

The **ldsseek** function seeks to the section specified by *sectindx* of the common object file currently associated with *ldptr*.

The **ldsseek** function seeks to the section specified by *sectname*.

The **ldsseek** and **ldsseek** functions return **SUCCESS** or **FAILURE**. **ldsseek** fails if *sectindx* is greater than the number of sections in the object file; **ldsseek** fails if there is no section name corresponding with *sectname*. Either function fails if there is no section data for the specified section or if it cannot seek to the specified section.

Note that the first section has an index of one.

See also

ldfcn(FP), **ldhread(S)**

Standards conformance

ldsseek and **ldsseek** are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

ldtbindex

compute the index of a symbol table entry of a common object file

Syntax

```
cc ... -lld

#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

long ldtbindex (ldptr)
LDFILE *ldptr;
```

Description

The **ldtbindex** function returns the index of the symbol table entry at the current position of the common object file associated with *ldptr*.

The index returned by **ldtbindex** may be used in subsequent calls to **ldtbread(S)**. However, since **ldtbindex** returns the index of the symbol table entry that begins at the current position of the object file, if **ldtbindex** is called immediately after a particular symbol table entry has been read, it returns the index of the next entry.

The **ldtbindex** function fails if there are no symbols in the object file, or if the object file is not positioned at the beginning of a symbol table entry.

Note that the first symbol in the symbol table has an index of zero.

See also

ldfcn(FP), **ldtbread(S)**, **ldtbseek(S)**

Standards conformance

ldtbindex is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

ldtbread

read an indexed symbol table entry of a common object file

Syntax

```
cc ... -lld

#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldtbread (ldptr, symindex, symbol)
LDFILE *ldptr;
long symindex;
SYMENT *symbol;
```

Description

The **ldtbread** function reads the symbol table entry specified by **symindex** of the common object file currently associated with **ldptr** into the area of memory beginning at **symbol**.

The **ldtbread** function returns **SUCCESS** or **FAILURE**. **ldtbread** fails if **symindex** is greater than or equal to the number of symbols in the object file, or if it cannot read the specified symbol table entry.

Note that the first symbol in the symbol table has an index of zero.

See also

ldfcn(FP), **ldgetname(S)**, **ldtbindex(S)**, **ldtbseek(S)**

Standards conformance

ldtbread is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

ldtbseek

seek to the symbol table of a common object file

Syntax

```
cc ... -lld

#include <filehdr.h>
#include <ldfcn.h>

int ldtbseek (ldptr)
LDFILE *ldptr;
```

Description

The **ldtbseek** function seeks to the symbol table of the common object file currently associated with *ldptr*.

The **ldtbseek** function returns **SUCCESS** or **FAILURE**. **ldtbseek** fails if the symbol table has been stripped from the object file, or if it cannot seek to the symbol table.

See also

ldfcn(FP), **ldtbindex(S)**, **ldtbread(S)**

Standards conformance

ldtbseek is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

libwindows

windowing terminal function library

Syntax

cc ... -lwindows

int openagent()

int New (cntlfd, origin_x, origin_y, corner_x, corner_y)
int cntlfd, origin_x, origin_y, corner_x, corner_y;

int Newlayer (cntlfd, origin_x, origin_y, corner_x, corner_y)
int cntlfd, origin_x, origin_y, corner_x, corner_y;

int openchan (chan)
int chan;

int Runlayer (chan, command)
int chan;
char *command;

int Current (cntlfd, chan)
int cntlfd, chan;

int Delete (cntlfd, chan)
int cntlfd, chan;

int Top (cntlfd, chan)
int cntlfd, chan;

int Bottom (cntlfd, chan)
int cntlfd, chan;

int Move (cntlfd, chan, origin_x, origin_y)
int cntlfd, chan, origin_x, origin_y;

int Reshape (cntlfd, chan, origin_x, origin_y, corner_x, corner_y)
int cntlfd, chan, origin_x, origin_y, corner_x, corner_y;

int Exit (cntlfd)
int cntlfd;

Description

This library of routines enables a program running on a host system to perform windowing terminal functions (see [layers\(C\)](#)).

The **openagent** routine opens the control channel of the *xt(HW)* channel group to which the calling process belongs. Upon successful completion, **openagent** returns a file descriptor, *cntlfd*, that can be passed to any of the other

libwindows routines except **openchan** and **Runlayer**. (*cntlfd* can also be passed to **close(S)**.) Otherwise, the value -1 is returned.

The **New** routine creates a new layer with a separate shell. The *origin_x*, *origin_y*, *corner_x*, and *corner_y* arguments are the coordinates of the layer rectangle. If all the coordinate arguments are 0, the user must define the layer's rectangle interactively. The layer appears on top of any overlapping layers. The layer is not made current (that is, the keyboard is not attached to the new layer). Upon successful completion, **New** returns the *xt(HW)* channel number associated with the layer. Otherwise, the value -1 is returned.

The **Newlayer** routine creates a new layer without executing a separate shell. Otherwise it is identical to **New**, described above.

The **openchan** routine opens the channel argument *chan* which is obtained from the **New** or **Newlayer** routine. Upon successful completion, **openchan** returns a file descriptor that can be used as input to **write(S)** or **close(S)**. Otherwise, the value -1 is returned.

The **Runlayer** routine runs the specified *command* in the layer associated with the channel argument *chan*. Any processes currently attached to this layer is killed, and the new process has the environment of the **layers(C)** process.

The **Current** routine makes the layer associated with the channel argument *chan* current (i.e., attached to the keyboard).

The **Delete** routine deletes the layer associated with the channel argument *chan* and kills all host processes associated with the layer.

The **Top** routine makes the layer associated with the channel argument *chan* appear on top of all overlapping layers.

The **Bottom** routine puts the layer associated with the channel argument *chan* under all overlapping layers.

The **Move** routine moves the layer associated with the channel argument *chan* from its current screen location to a new screen location at the origin point (*origin_x*, *origin_y*). The size and contents of the layer are maintained.

The **Reshape** routine reshapes the layer associated with the channel argument *chan*. The arguments *origin_x*, *origin_y*, *corner_x*, and *corner_y* are the new coordinates of the layer rectangle. If all the coordinate arguments are 0, the user is allowed to define the layer's rectangle interactively.

The **Exit** routine causes the **layers(C)** program to exit, killing all processes associated with it.

Return value

Upon successful completion, **Runlayer**, **Current**, **Delete**, **Top**, **Bottom**, **Move**, **Reshape**, and **Exit** return a 0, while **openagent**, **New**, **Newlayer**, and **openchan** return values as described above under each routine. If an error occurs, -1 is returned.

Files

/usr/lib/libwindows.a windowing terminal function library

Note

The values of layer rectangle coordinates are dependent on the type of terminal. This dependency affects the routines that pass layer rectangle coordinates: **Move**, **New**, **Newlayer**, and **Reshape**. Some terminals expects these numbers to be passed as character positions (bytes); others expects the information to be in pixels (bits).

For example, for the AT&T TELETYPE 5620 DMD terminal, **New**, **Newlayer**, and **Reshape** take minimum values of 8 (pixels) for *origin_x* and *origin_y* and maximum values of 792 (pixels) for *corner_x* and 1016 (pixels) for *corner_y*. In addition, the minimum layer size is 28 by 28 pixels and the maximum layer size is 784 by 1008 pixels.

See also

close(S), **jagent(M)**, **write(S)**, **layers(C)**, **xt(HW)**.

Standards conformance

libwindows is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

link

link to a file

Syntax

```
cc ... -lc
```

```
int link (path, newpath)
char *path, *newpath;
```

Description

The *path* argument points to a path name naming an existing file. The *newpath* argument points to a path name naming the new directory entry to be created. The link system call creates a new link (directory entry) for the existing file.

Upon successful completion, link marks for update the "st_ctime" and "st_mtime" fields of the parent directory. Also, the "st_ctime" and "st_mtime" fields of the file are marked for update.

The link system call fails and no link is created if one or more of the following is true:

[EACCES]	1. A component of either path prefix denies search permission. 2. The requested link requires writing in a directory with a mode that denies write permission.
[EEXIST]	The link named by <i>newpath</i> exists.
[EFAULT]	<i>path</i> points outside the allocated address space of the process.
[EINTR]	A signal was caught during the link system call.
[EMLINK]	The maximum number of links to a file would be exceeded.
[EMULTIHOP]	Components of <i>path</i> require hopping to multiple remote machines.
[ENOENT]	1. A component of either path prefix does not exist. 2. The file named by <i>path</i> does not exist. 3. <i>newpath</i> points to a null path name.
[ENOLINK]	<i>path</i> points to a remote machine and the link to that machine is no longer active.
[ENOSPC]	The directory containing the link cannot be extended.
[ENOTDIR]	A component of either path prefix is not a directory.

[EPERM]	The file named by <i>path</i> is a directory and the effective user ID is not super user.
[EROFS]	The requested link requires writing in a directory on a read-only file system.
[EXDEV]	The link named by <i>newpath</i> and the file named by <i>path</i> are on different logical devices (file systems).

Diagnostics

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

See also

stat(S), unlink(S)

Standards conformance

link is conformant with:

AT&T SVID Issue 2;
 X/Open Portability Guide, Issue 3, 1989;
 IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
 and NIST FIPS 151-1.

llog

library logging package

Syntax

```
cc ... -lmmdf

#include ll_log.h

ll_close (loginfo)
struct ll_struct *loginfo;

ll_err (loginfo, verbosity, printf args . . . )
char verbosity;
struct ll_struct *loginfo;

ll_hdinit (loginfo, pref)
struct ll_struct *loginfo;
char *pref;

ll_init (llloginfo)
struct ll_struct *loginfo;

ll_log (loginfo, verbosity, printf args . . . )
struct ll_struct *loginfo;
char verbosity;

ll_open (loginfo)
struct ll_struct *loginfo;
```

```

struct ll_struct
{
char *ll_file;      /* Path name to logging file */
char *ll_hdr;       /* Text to put in opening line */
char ll_level;      /* Max verbosity level */
    LLOGBST         /* Basic statistics */
    LLOGBTR         /* Trace of more basic activities */
    LLOGFAT         /* Fatal error */
    LLOGFST         /* Full statistics */
    LLOGFTR         /* Full program trace */
    LLOGGEN         /* General information */
    LLOGMAX         /* Maximum possible value */
    LLOGPTR         /* Trace of program phases */
    LLOGTMP         /* Temporary (minor?) error */
char ll_msize;      /* Max log size, in 100's of blocks */
char ll_stat;       /* Assorted switches */
    LLOGCLS         /* Close after each write */
    LLOGCYC         /* Cycle to beginning when log full */
    LLOGSOME        /* CLS and WAT */
    LLOGTIMED       /* Reopen periodically */
    LLOGWAT         /* Wait, if log locked and LLOGCLS */
char ll_fd;         /* char holds file descriptor */
int ll_timax;       /* Time fd can be in use before reopen */
time_t ll_timer;    /* Time of last open of log file */
FILE *ll_fp;        /* Standard I/O stream pointer */
};

```

Description

ll_close - closes the MMDF logging file and resets ll_fd to zero

ll_err - returns **errno** and the specified MMDF string

ll_hdinit - sets the prefix of the MMDF header string

ll_init - accesses the opened MMDF logging file

ll_log - makes an MMDF log entry

ll_open - opens the MMDF logging file

This package is intended to perform standardized information logging for programs. The time of an entry is recorded in addition to its text.

Format of Entries

Depending on the mode of logging (whether the log is left open between **ll_log** calls or is re-opened to make each entry, as indicated by the **LLOGCLS** flag) the date, hour and loginfo.llhdr string are recorded on a header line when the log is opened (if **LLOGCLS** and **LLOGTIMED** are off) or on the logging line (if **LLOGCLS** is on). In addition, the current minute and second are recorded, followed by the actual log text. Each log entry is started on a new line. The text of the entry is entered with a **printf**, called with parameters passed to the **ll_log** function.

Access to Log Files

Multiple logs can be accessed simultaneously by the same process, because all of the state information for a single log is kept in the user-maintained structure, `llogstruct`, which is included as a parameter when calling `llog` functions. While opened by one process, a single log cannot be shared with another process, since logging files are opened with exclusive access (the specific locking mechanism depends on the version of the UNIX operating system that you are running). The `LLOGCLS` flag is intended to help circumvent this protection. Also, the `LLOGWAT` flag may be used to cause `ll_open` to repeatedly retry opening the log, if the failure to open it was due to its being used by another process. `ll_open` tries up to five times, waiting five seconds between tries; after that, it returns. The failure is treated as temporary if `LLOGCLS` is used (implying infrequent logging expected); otherwise the failure is treated as permanent. In either case, the log is marked as "broken" and no further attempts are made to use it until either `ll_close` or `ll_open` is called. Exclusive opening of the log file is not currently used. As a result, the `LLOGWAT` flag does not have any effect. Exclusive opening could be re-added but seeking to the end of the log and flushing after writes works most of the time.

Limiting Size of Log Files

The package can impose a size limit to log files. If `lloginfo.llmsize` is zero, size-limiting is not imposed, otherwise `llog` does not allow a log to grow larger (`25 * lloginfo.llmsize`) blocks. If the log is being held open between entries and cycling is allowed (that is, `LLOGCYC` is set and `LLOGCLS` is not) then when `ll_init` is called (by `ll_open` or by the user) during successive uses, entries are made starting from the beginning of the file. If the `LLOGCLS` bit is set or if the `LLOGCYC` bit is not set, then the log is marked as unusable.

The `ll_open` routine attempts to open the logging file. It returns the return value from its `open` call. It does not attempt to create the file if the file does not exist. The result of the `open` call is placed in the `ll_fd` member of the `logstruct` structure. The using program should ensure that `ll_fd` initially has the value zero. `ll_log` simply returns, if `ll_fd` is -1; therefore the caller can ignore failures to open the log. Also, the existence of the logging file can be used as a convenient external switch for turning logging on and off.

Entries are appended to the end of any old entries in the file, except when cycling occurs. If the `LLOGCLS` bit is set in the `lloginfo.ll_stat` field, then the file is opened only long enough to make an entry. This action makes it more reasonable for multiple processes to share a log into which relatively few entries are made, such as when only error messages are recorded.

The `ll_init` routine is to be used when the logging file is already opened (for example, when the file descriptor has been inherited from a parent process) instead of calling `ll_open`.

The `ll_log` routine actually makes the log entries. If the log is not already opened and `lloginfo.ll_fd` is 0, `ll_log` calls `ll_open`. (Note that this means that the calling program usually does not need to call `ll_open`, unless there is a concern about locking the file.) The `ll_level` value indicates how much

logging the caller desires. When called, `ll_log` does not make the log entry if the value of the verbosity argument is greater than the value of `ll_level`. This allows, for example, distinction between mandatory and debugging information, with many levels of (256) information possible. As a guide, the list shown in the synopsis can be used.

The `ll_log` routine uses `printf` to make the entries. There is a limit of ten (10) arguments that may be passed to it. The third argument to `ll_log` becomes the first argument to `printf`. The `flush` routine is then called so that the log is always up-to-date; this is deemed more important than fully minimizing write calls, and in any event, is required to allow `ll_log` to use the `printf` mechanism in a way that does not interfere with its use elsewhere in the process.

The `ll_close` routine closes the logging file and marks `ll_fd` as clear (that is, it resets it to be zero). Note that this normally need not be called, since the file is automatically closed when the process exits. However, it is necessary to close the log, during execution, if a child or some other process is expected to open the same log independently. Calling `ll_close` also always clears the file descriptor, making it possible for a new try at opening the log, if it failed previously.

Diagnostics

The `ll_open` routine returns the value of its open call, if that fails; otherwise it returns the value of its call to `ll_init`. The `ll_log` routine returns the value of its `flush` call, or zero if `ll_fd` is -1; if `flush` returns less than zero, then `ll_log` closes the log and returns NOTOK. The `ll_close` routine returns the value of its close call.

See also

`ml_send(S)`, `mmdf(S)`, `phs(S)`, `tai(S)`

Standards conformance

The `ll` package is not part of any currently supported standard; it was developed at the University of Delaware and is used by permission.

lock

locks a process in primary memory

Syntax

```
cc ... -lx  
  
int lock (flag);  
int flag;
```

Description

If the *flag* argument is nonzero, the process executing this call is not swapped except if it is required to grow. If the argument is zero, the process is unlocked. This call may only be executed by the super user.

Notes

Locked processes interfere with the compaction of primary memory and can cause deadlock. Systems with small memory configurations should avoid using this call. It is best to lock a process soon after booting to ensure that it is locked into memory.

This feature is a XENIX specific enhancement and may not be present in all UNIX implementations.

Standards conformance

lock is conformant with:
X/Open Portability Guide, Issue 3, 1989.

lockf

record locking on files

Syntax

```
cc ... -lc

#include <unistd.h>

int lockf (fildes, function, size)
int fildes, function;
long size;
```

Description

The **lockf** command allows sections of a file to be locked; advisory or mandatory write locks are used depending on the mode bits of the file [see **chmod(S)**]. Locking calls from other processes which attempt to lock the locked file section either return an error value or are put to sleep until the resource becomes unlocked. All the locks initiated by a process are removed when the process terminates or when the process closes the file. See **fcntl(S)** for more information about record locking.

fildes is an open file descriptor. The file descriptor must have **O_WRONLY** or **O_RDWR** permission in order to establish a lock with this function call.

function is a control value which specifies the action to be taken. The permissible values for **function** are defined in **<unistd.h>** as follows:

```
#define F_ULOCK 0    /* Unlock a previously locked section */
#define F_LOCK  1    /* Lock a section for exclusive use */
#define F_TLOCK 2    /* Test and lock a section for exclusive use */
#define F_TEST  3    /* Test section for other processes locks */
```

All other values of **function** are reserved for future extensions and will result in an error return if not implemented.

F_TEST is used to detect if a lock by another process is present on the specified section. **F_LOCK** and **F_TLOCK** both lock a section of a file if the section is available. **F_ULOCK** removes locks from a section of the file.

size is the number of contiguous bytes to be locked or unlocked. The section to be locked starts at the current offset in the file and extends forward for a positive size and backward for a negative size (the preceding bytes up to but not including the current offset). If **size** is zero, the section from the current offset through the largest file offset is locked (that is, from the current offset through the present or any future end-of-file). An area need not be allocated to the file in order to be locked as such locks may exist past the end-of-file.

The sections locked with **F_LOCK** or **F_TLOCK** may, in whole or in part, contain or be contained by a previously locked section for the same process. When this occurs, or if adjacent sections occur, the sections are combined into a single section. If the request requires that a new element be added to the table of active locks and this table is already full, an error is returned, and the new section is not locked.

F_LOCK and **F_TLOCK** requests differ only by the action taken if the resource is not available. **F_LOCK** causes the calling process to sleep until the resource is available. **F_TLOCK** causes the function to return a -1 and set **errno** to **[EACCES]** if the section is already locked by another process.

F_ULOCK requests may, in whole or in part, release one or more locked sections controlled by the process. When sections are not fully released, the remaining sections are still locked by the process. Releasing the center section of a locked section requires an additional element in the table of active locks. If this table is full, an **[EDEADLK]** error is returned, and the requested section is not released.

A potential for deadlock occurs if a process controlling a locked resource is put to sleep by accessing another process's locked resource. Thus calls to **lockf** or **fcntl** scan for a deadlock prior to sleeping on a locked resource. An error return is made if sleeping on the locked resource would cause a deadlock.

Sleeping on a resource is interrupted with any signal. The **alarm(S)** function may be used to provide a timeout facility in applications which require this facility.

The **lockf** utility fails if one or more of the following is true:

[EBADF]	<i>fildes</i> is not a valid open descriptor.
[EACCES]	cmd is F_TLOCK or F_TEST and the section is already locked by another process.
[EDEADLK]	cmd is F_LOCK and a deadlock would occur. Also the cmd is either F_LOCK , F_TLOCK , or F_ULOCK and the number of entries in the lock table would exceed the number allocated on the system.
[ECOMM]	<i>fildes</i> is on a remote machine and the link to that machine is no longer active.

Diagnostics

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

Warnings

Unexpected results may occur in processes that do buffering in the user address space. The process may later read/write data which is/was locked. The standard I/O package is the most common source of unexpected buffering.

Because in the future the variable `errno` will be set to `EAGAIN` rather than `EACCES` when a section of a file is already locked by another process, portable application programs should expect and test for either value.

See also

`chmod(S)`, `close(S)`, `creat(S)`, `fcntl(S)`, `Intro(S)`, `open(S)`, `read(S)`, `write(S)`

Standards conformance

`lockf` is conformant with:
AT&T SVID Issue 2

locking

locks or unlocks a file region for reading or writing

Syntax

```
cc ... -lx

#include <sys/types.h>
#include <sys/locking.h>

int locking (fildes, mode, size);
int fildes, mode;
long size;
```

Description

locking allows a specified number of bytes in a file to be controlled by the locking process. Other processes which attempt to read or write a portion of the file containing the locked region may sleep until the area becomes unlocked depending upon the mode in which the file region was locked.

A file must be open with read or read/write permission for a read lock to be performed. Write or read/write permission is required for a write lock. If either of these conditions are not met, the lock fails with the error **EINVAL**.

A process that attempts to write to or read a file region that has been locked against reading and writing by another process (using the **LK_LOCK** or **LK_NBLCK** mode) sleeps until the region of the file has been released by the locking process.

A process that attempts to write to a file region that has been locked against writing by another process (using the **LK_RLCK** or **LK_NBRLOCK** mode) sleeps until the region of the file has been released by the locking process, but a read request for that file region proceeds normally.

A process that attempts to lock a region of a file that contains areas that have been locked by other processes sleeps if it has specified the **LK_LOCK** or **LK_RLCK mode** in but returns with the error **EACCES** if it specified **LK_NBLCK** or **LK_NBRLOCK**.

fildes is the value returned from a successful **creat**, **open**, **dup**, or **pipe** system call.

mode specifies the type of lock operation to be performed on the file region. The available values for *mode* are:

LK_UNLCK 0

Unlocks the specified region. The calling process releases a region of the file it had previously locked.

LK_LOCK 1

Locks the specified region. The calling process sleeps until the entire region is available if any part of it has been locked by a different process. The region is then locked for the calling process and no other process may read or write in any part of the locked region. (lock against read and write).

LK_NBLCK 2

Locks the specified region. If any part of the region is already locked by a different process, EACCES is returned instead of waiting for the region to become available for locking (nonblocking lockrequest).

LK_RLCK 3

Same as LK_LOCK except that the locked region may be read by other processes (read permitted lock).

LK_NBRLOCK 4

Same as LK_NBLCK except that the locked region may be read by other processes (nonblocking, read permitted lock).

The locking utility uses the current file pointer position as the starting point for the locking of the file segment. So a typical sequence of commands to lock a specific range within a file might be as follows:

```
fd=open('datafile',O_RDWR);
lseek(fd, 200L, 0);
locking(fd, LK_LOCK, 200L);
```

Accordingly, to lock or unlock an entire file a seek to the beginning of the file (position 0) must be done and then a locking call must be executed with a size of 0.

size is the number of contiguous bytes to be locked or unlocked. The region to be locked starts at the current offset in the file. If *size* is 0, the entire file (up to a maximum of 2 to the power of 30 bytes) is locked or unlocked. *size* may extend beyond the end of the file, in which case only the process issuing the lock call may access or add information to the file within the boundary defined by *size*.

The potential for a deadlock occurs when a process controlling a locked area is put to sleep by accessing another process' locked area. Thus calls to locking, read, or write scan for a deadlock prior to sleeping on a locked region. An EDEADLK (or EDEADLOCK) error return is made if sleeping on the locked region would cause a deadlock.

Lock requests may, in whole or part, contain or be contained by a previously locked region for the same process. When this occurs, or when adjacent regions are locked, the regions are combined into a single area if the mode of the lock is the same (that is, either read permitted or regular lock). If the mode of the overlapping locks differ, the locked areas are assigned assuming that the most recent request must be satisfied. Thus if a read only lock is applied to a region, or part of a region, that had been previously locked by the same process against both reading and writing, the area of the file specified by the new lock is locked for read only, while the remaining region, if any, remains locked against reading and writing. There is no arbitrary limit to the number of regions that may be locked in a file. There is, however, a system-wide limit on the total number of locked regions. This limit is 200 for UNIX style systems.

Unlock requests may, in whole or part, release one or more locked regions controlled by the process. When regions are not fully released, the remaining areas are still locked by the process. Release of the center section of a locked area requires an additional locked element to hold the separated section. If the lock table is full, an error is returned, and the requested region is not released. Only the process which locked the file region may unlock it. An unlock request for a region that the process does not have locked, or that is already unlocked, has no effect. When a process terminates, all locked regions controlled by that process are unlocked.

If a process has done more than one open on a file, all locks put on the file by that process are released on the first close of the file.

Although no error is returned if locks are applied to special files or pipes, read/write operations on these types of files will ignore the locks. Locks may not be applied to a directory.

Diagnostics

locking returns the value (int) -1 if an error occurs. If any portion of the region has been locked by another process for the LK_LOCK and LK_RLCK actions and the lock request is to test only, *errno* is set to EAGAIN when used with UNIX System V binaries. If the binary using this routine is a UNIX 3.0 binary, this *errno* is set to EACCES. If the file specified is a directory, *errno* is set to EACCES. If locking the region would cause a deadlock, *errno* is set to EDEADLK (or EDEADLOCK). If there are no more free internal locks, *errno* is set to EDEADLK (or EDEADLOCK).

See also

close(S), *creat(S)*, *dup(S)*, *lseek(S)*, *open(S)*, *read(S)*, *write(S)*

Standards conformance

locking is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

localeconv

get lconv structure pointer

Syntax

```
cc ... -lc

#include <locale.h>

struct lconv *localeconv ( )
```

Description

The **localeconv** routine returns a pointer to a statically allocated **lconv** structure. When **localeconv** is called, **setlocale(S)** is implicitly called and sets locale-specific values for **LC_ALL**, **LC_NUMERIC**, or **LC_MONETARY** automatically into the structure. Refer to **lconv(FP)** for information about the **lconv** structure.

Return value

A pointer to the static **lconv** structure is returned. No errors occur from calling this routine.

See also

lconv(FP), **setlocale(S)**

Standards conformance

The **localeconv** routine conforms to:
ANSI X3.159-1989 Programming Language — C and IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1).

logname

return login name of user

Syntax

```
cc ... -lPW
char *logname ()
```

Description

The **logname** function returns a pointer to the null-terminated login name; it extracts the LOGNAME environment variable from the user's environment.

Files

/etc/profile

Notes

The return values point to static data whose content is overwritten by each call.

This method of determining a login name is subject to forgery.

See also

env(C), environ(M), getenv(S), login(M), profile(M)

Standards conformance

logname is conformant with:

AT&T SVID Issue 2, but has been withdrawn from XPG3.

lsearch, lfind

linear search and update

Syntax

```
cc ... -lc

#include <stdio.h>
#include <search.h>

void *lsearch (key, base, nelp, width, compar)
void *key *base
size_t width, *nelp;
int (*compar)();

void *lfind (key, base, nelp, width, compar)
void *key *base
size_t width, *nelp;
int (*compar)();
```

Description

lsearch - performs linear search of table

lfind - searches for object and adds to table

The **lsearch** function is a linear search routine generalized from Knuth (6.1) Algorithm S. It returns a pointer into a table indicating where an entry may be found. If the entry does not occur, it is added at the end of the table. The *key* argument points to the entry to be sought in the table. The *base* argument points to the first element in the table. The *nelp* argument points to an integer containing the current number of elements in the table. The *width* argument is the size of an element in bytes. The integer is incremented if the entry is added to the table. The *compar* argument is the name of the comparison function which the user must supply (**strcmp**, for example). It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal and non-zero otherwise.

The **lfind** function is the same as **lsearch** except that if the entry is not found, it is not added to the table. Instead, a NULL pointer is returned.

Example

This fragment reads in less than TABSIZE strings of length less than ELSIZE and stores them in a table, eliminating duplicates.

```
#include <stdio.h>
#include <search.h>

#define TABSIZE 50
#define ELSIZE 120

char line[ELSIZE], tab[TABSIZE][ELSIZE], *lsearch( );
size_t nel = 0;
int strcmp( );

. . .
while (fgets(line, ELSIZE, stdin) != NULL &&
      nel < TABSIZE)
    (void) lsearch((void *)line, (tab *)tab, &nel,
                  ELSIZE, strcmp);
. . .
```

Diagnostics

If the searched-for entry is found, both **lsearch** and **lfind** return a pointer to it. Otherwise, **lfind** returns **NULL** and **lsearch** returns a pointer to the newly added element.

Notes

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

Undefined results can occur if there is not enough room in the table to add a new item.

See also

bsearch(S), **hsearch(S)**, **string(S)**, **tsearch(S)**

Standards conformance

lfind and **lsearch** are conformant with:

AT&T SVID Issue 2;
and X/Open Portability Guide, Issue 3, 1989.

lseek

move read/write file pointer

Syntax

```
cc ... -lc

#include <unistd.h>

long lseek (fildes, offset, whence)
int fildes;
long offset;
int whence;
```

Description

The *fildes* argument is a file descriptor returned from a **creat**, **open**, **dup**, or **fcntl** system call. The **lseek** system call sets the file pointer associated with *fildes* as follows:

- If *whence* is 0, the pointer is set to *offset* bytes.
- If *whence* is 1, the pointer is set to its current location plus *offset*.
- If *whence* is 2, the pointer is set to the size of the file plus *offset*.

Symbolic constants for *whence* are defined in the *<unistd.h>* header file:

Name	Description
SEEK_CUR	Set file-pointer to current location plus <i>offset</i> .
SEEK_END	Set file-pointer to EOF plus <i>offset</i> .
SEEK_SET	Set file-pointer equal to <i>offset</i> bytes.

Upon successful completion, the resulting pointer location, as measured in bytes from the beginning of the file, is returned. Note that if *fildes* is a remote file descriptor and *offset* is negative, **lseek** returns the file pointer even if it is negative.

lseek fails and the file pointer remains unchanged if one or more of the following is true:

[EBADF] *fildes* is not an open file descriptor.

- [EINVAL] 1. (with SIGSYS signal is present): The *whence* argument is not 0, 1, or 2.
 2. *fildes* is not a remote file descriptor, and the resulting file pointer would be negative.

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

[ESPIPE] *fildes* is associated with a pipe or fifo.

Diagnostics

Upon successful completion, a non-negative integer indicating the file pointer value is returned. Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

See also

creat(S), dup(S), fcntl(S), open(S)

Standards conformance

lseek is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

malloc, free, realloc, calloc, cfree

allocates main memory

Syntax

```
cc ... -lc

#include <stdlib.h>

void *malloc (size)
size_t size;

void free (ptr)
void *ptr;

void *realloc (ptr, size)
void *ptr;
size_t size;

void *calloc (nelem, elsize)
size_t nelem, elsize;
```

SVID Syntax

```
char *malloc (size)
size_t size;

char *realloc (ptr, size)
void *ptr;
size_t size;

char *calloc (nelem, elsize)
size_t nelem, elsize;

void cfree (p, num, size)
char *p;
unsigned num, size;
```

Description

malloc - allocates space for an object
free - deallocates space
realloc - changes the size of memory object
calloc - allocates unused space for an array
cfree - deallocates space

This is the first of two **malloc(S)** manual pages. This manual page describes the version found in the *libc.a* archive, and the next **malloc** manual page describes the *libmalloc.a* version.

This version of **malloc** provides a general-purpose memory allocation package. This routine allocates the first contiguous reach of free space found in a circular search from the last block allocated or freed, coalescing adjacent blocks as it searches. **malloc** calls **sbrk** to get more memory from the system when there is no suitable space already free. allocates space for an object whose size in bytes is specified by *size* and whose value is indeterminate.

The **free** function causes the space pointed to by *ptr* to be deallocated so that it is made available for further allocation; however, its contents are left undisturbed. If *ptr* is null, no action occurs. Undefined results occur if space assigned by **malloc** is overrun or if some random number is handed to **free**. The **cfree** routine is provided for compliance to the iBCSe2 standard and simply calls **free**. The *num* and *size* arguments to **cfree** are not used.

The **realloc** function changes the size of the memory object pointed to by *ptr* to the size specified by *size*. The contents of the object are unchanged up to the lesser of the new and old sizes.

The **calloc** function allocates unused space for an array of *nelem* elements each of whose size in bytes is *elsize*. The space is initialized to zero.

Errors

These functions fail if **errno** is set to:

[ENOMEM] Insufficient memory is available.

Diagnostics

malloc, **realloc**, and **calloc** return a null pointer (0) if there is no available memory or if the area is corrupted by storing data outside a block boundary. When **realloc** returns 0, the block pointed to by *ptr* is left intact.

Notes

As noted, **malloc** calls **sbrk** to allocate memory. Since **sbrk** takes a signed integer as its argument, **malloc** fails if an attempt is made to allocate more memory than a signed integer holds (2Gb -1).

Search time increases when many objects have been allocated; that is, if a program allocates but never frees, then each successive allocation takes longer.

The **sbrk(S)** routine has been withdrawn from XPG3.

See also

brkctl(S), malloc(S), sbrk(S)

Standards conformance

malloc, free, realloc, and calloc are conformant with:

X/Open Portability Guide, Issue 3, 1989;
and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

The **cfree** routine conforms to:

Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

(S)

malloc, free, realloc, calloc, malloc, mallinfo

allocates main memory quickly

Syntax

```
cc ... -lmalloc

#include <malloc.h>

void *malloc (size)
unsigned size;

void free (ptr)
char *ptr;

void *realloc (ptr, size)
void *ptr;
unsigned size;

char *calloc (nelem, elsize)
unsigned nelem, elsize;

int malloc (cmd, value)
int cmd, value;

struct mallinfo mallinfo ( )
```

SVID Syntax

```
char *malloc (size)
unsigned size;

char *realloc (ptr, size)
void *ptr;
unsigned size;
```

Description

malloc - allocates space
free - frees allocated space
realloc - changes the size of the allocated space
calloc - allocates space for an array of elements
malloc - controls the space allocation algorithm
mallinfo - reports allocated space usage

There are two versions of the **malloc(S)** package. This is the library version which provides a simple general-purpose memory allocation package, that runs considerably faster than the other **malloc(S)** package. Both versions are documented in these **malloc(S)** manual pages; the description of the standard default package starts on page 1.

This **malloc(S)** package is found in the library “**malloc**” and is loaded when the option **-lmalloc** is used with **cc(CP)** or **ld(CP)**.

malloc returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to **free** is a pointer to a block previously allocated by **malloc**; after **free** is performed this space is made available for further allocation, and its contents destroyed (see **mallopt** below for a way to change this behavior).

Undefined results occur if the space assigned by **malloc** is overrun or if some random number is handed to **free**.

realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents are unchanged up to the lesser of the new and old sizes.

calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

mallopt provides control over the allocation algorithm. The available values for *cmd* are:

M_GRAIN	Set <i>grain</i> to <i>value</i> . The sizes of all blocks smaller than <i>maxfast</i> are considered to be rounded up to the nearest multiple of <i>grain</i> . <i>grain</i> must be greater than 0. The default value of <i>grain</i> is the smallest number of bytes that allow alignment of any data type. <i>value</i> is rounded up to a multiple of the default when <i>grain</i> is set.
M_KEEP	Preserve data in a freed block until the next malloc , realloc , or calloc . This option is provided only for compatibility with the old version of malloc and is not recommended.
M_MXFAST	Set <i>maxfast</i> to <i>value</i> . The algorithm allocates all blocks below the size of <i>maxfast</i> in large groups and then does them out very quickly. The default value for <i>maxfast</i> is 0.
M_NLBLKS	Set <i>numlblks</i> to <i>value</i> . The above mentioned “large groups” each contain <i>numlblks</i> blocks. <i>numlblks</i> must be greater than 0. The default value for <i>numlblks</i> is 100.

These values are defined in the **<malloc.h>** header file.

mallopt may be called repeatedly, but may not be called after the first small block is allocated.

mallinfo returns a structure that provides information about allocation. The **mallinfo** structure is returned:

```
struct mallinfo {
    int arena;      /* total space in arena          */
    int ordblks;    /* number of ordinary blocks     */
    int smlblks;    /* number of small blocks        */
    int hblks;      /* number of holding blocks      */
    int hblkhd;     /* space in holding block headers */
    int usmlblks;   /* space in small blocks in use  */
    int fsmblks;    /* space in free small blocks    */
    int uordblks;   /* space in ordinary blocks in use */
    int fordblks;   /* space in free ordinary blocks */
    int keepcost;   /* space penalty if keep option  */
                  /* is used                      */
};
```

This structure is defined in the `<malloc.h>` header file.

Here is an example program code segment for the **mallinfo** function:

```
#include <stdio.h>
#include <malloc.h>

main()
{
    char *malloc(), *cp;
    struct mallinfo minfo;

    if ((cp = malloc(1024)) == NULL)

        perror("malloc");
        exit(1);
    }
    minfo = mallinfo();
    printf("%d %d %d\n", minfo.arena, minfo.ordblks, minfo.uordblks);
}
```

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

Diagnostics

malloc, **realloc** and **calloc** return a null pointer if there is not enough available memory. When **realloc** returns null, the block pointed to by **ptr** is left intact. If **mallopt** is called after any allocation or if **cmd** or **value** are invalid, non-zero is returned. Otherwise, it returns zero.

Warnings

This package usually uses more data space than the other **malloc(S)**.

The code size is also bigger than the other **malloc(S)**.

Note that unlike the other **malloc(S)**, this package does not preserve the contents of a block when it is freed, unless the **M_KEEP** option of **mallopt** is used.

Undocumented features of the other **malloc(S)** have not been duplicated.

These routines must be linked with the **-lmalloc** linker option.

See also

brkctl(S), **malloc(S)**, **sbrk(S)**

Standards conformance

calloc, **free**, **malloc** and **realloc** conform to:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

ANSI X3.159-1989 Programming Language — C;

Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

mallinfo and **mallopt** conform to:

AT&T SVID Issue 2, but have been withdrawn from XPG3.

The *malloc.h* header file has been withdrawn from XPG3.

matherr

error-handling function

Syntax

```
cc ... -lm

#include <math.h>

int matherr (x)
struct exception *x;
```

Description

The **matherr** function is invoked by functions in the Math Library when errors are detected. Users may define their own procedures for handling errors by including a function named **matherr** in their programs. The **matherr** function must be of the form described above. When an error occurs, a pointer to the exception structure **x** is passed to the user-supplied **matherr** function. This structure, which is defined in the *<math.h>* header file, is as follows:

```
struct exception {
    int type;
    char *name;
    double arg1, arg2, retval;
};
```

The element **type** is an integer describing the type of error that has occurred, from the following list of constants (defined in the header file):

DOMAIN	argument domain error
OVERFLOW	overflow range error
PLOSS	partial loss of significance
SING	argument singularity
TLOSS	total loss of significance
UNDERFLOW	underflow range error

The element **name** points to a string containing the name of the function that incurred the error. The variables **arg1** and **arg2** are the arguments with which the function was invoked. **retval** is set to the default value that is returned by the function unless the user's **matherr** sets it to a different value.

If the user's **matherr** function returns non-zero, an error message is not printed, and **errno** is not set.

If `matherr` is not supplied by the user, the default error-handling procedures, described with the `math` functions involved, are invoked upon error. These procedures are also summarized in the table below. In every case, `errno` is set to `EDOM` or `ERANGE` and the program continues.

Example

```
#include <math.h>

int
matherr(x)
register struct exception *x;
{
    switch (x->type) {
    case DOMAIN:
        /* change sqrt to return sqrt(-arg1), not 0 */
        if (!strcmp(x->name, "sqrt")) {
            x->retval = sqrt(-x->arg1);
            return (0); /* print message and set errno */
        }
    case SING:
        /* all other domain or sing errors,*/
        /* print message and stop further processing */
        fprintf(stderr, "domain error in %s\n", x->name);
        abort();
    case PLOSS:
        /* print detailed error message */
        fprintf(stderr, "loss of significance in %s(%g) = %g\n",
            x->name, x->arg1, x->retval);
        return (C); /* take no other action */
    }
    return (0); /* all other errors, execute default procedure */
}
```

(S)

Default Error Handling Procedures						
	Types of Errors					
type ->	DOMAIN	SING	OVERFLOW	UNDERFLOW	TLOSS	PLOSS
errno ->	EDOM	EDOM	ERANGE	ERANGE	ERANGE	ERANGE
BESSEL:	-	-	-	-	M,0	*
y0, y1, yn (arg ≤ 0)	M,-H	-	-	-	-	-
EXP:	-	-	H	0	-	-
LOG, LOG10:						
(arg < 0)	M, -H	-	-	-	-	-
(arg = 0)	-	M, -H	-	-	-	-
POW:	-	-	±H	0	-	-
neg ** non-int	M,0	-	-	-	-	-
0 ** non-pos						
SQRT:	M,0	-	-	-	-	-
GAMMA:	-	M, H	H	-	-	-
HYPOT:	-	-	H	-	-	-
SINH:	-	-	±H	-	-	-
COSH:	-	-	H	-	-	-
SIN, COS, TAN: -	-	-	-	M,0	*	
ASIN, ACOS, ATAN2: M, 0	-	-	-	-	-	

ABBREVIATIONS:

- * As much as possible of the value is returned.
- M Message is printed (EDOM error).
- H HUGE is returned.
- H -HUGE is returned.
- ±H HUGE or -HUGE is returned.
- 0 0 is returned.

Standards conformance

matherr is conformant with:

AT&T SVID Issue 2, but has been withdrawn from XPG3.

mblen, mbtowc, mbstowcs, wctomb, wcstombs

multibyte character routines

Syntax

```
cc ... -lc

#include <stdlib.h>

int mblen (s, len)
const char *s;
size_t len;

int mbtowc (pwc, s, len)
wchar_t *pwc;
const char *s;
size_t len;

size_t mbstowcs (pwcs, s, len)
wchar_t *pwcs;
const char *s;
size_t len;

int wctomb (s, wchar)
char *s;
wchar_t wchar;

size_t wcstombs (s, pwcs, len)
char *s;
const wchar_t *pwcs;
size_t len;
```

Description

- mblen** - Get length of multibyte character
- mbtowc** - Convert single multibyte character to wide character
- mbstowcs** - Convert multibyte string to wide string
- wctomb** - Convert wide character to multibyte character
- wcstombs** - Convert wide string to multibyte string

These routines handle multibyte characters. A multibyte character is simply a character spread across more than one byte. Wide characters contain more than 8 bits in a character. The `wchar_t` type defines a wide character. In routines specifying such, the `s` argument points to an array of bytes. The `len` argument is the length of the array. If `s` is null, a value of zero is always returned.

The **mblen** routine returns the length of a multibyte character.

The **mbtowc** routine translates a single multibyte character pointed to by *s* to a multibyte character, storing the result in the array pointed to by *pwc*. The number of bytes stored in *pwc* is returned.

The **mbstowcs** routine translates a multibyte character pointed to by *s* to a wide character string, storing the result in the string pointed to by *pwc*. The number of wide characters stored in *pwc* is returned.

The **wctomb** routine translates a wide character pointed to by *wchar* into a multibyte character array pointed to by *s*. The number of bytes stored in *s* is returned.

The **wcstombs** routine translates a wide character string pointed to by *pwcs* into a multibyte character string pointed to by *s*. The number of bytes stored in *s* is returned (not counting the null byte terminating the string).

Return values

mblen returns the length if successful; otherwise, a value of zero is returned if the string pointer is null, or -1 is returned if the length is zero bytes.

mbtowc returns the length if successful; otherwise, a value of zero is returned if the string pointer is null. When zero is returned, *pwc* is also set to null.

mbstowcs returns the length if successful; otherwise, a value of zero is returned if the string pointer is null. If an invalid character is encountered in the string, processing stops and -1 is returned.

wctomb returns the length if successful; otherwise, a value of zero is returned if the string pointer is null. If the wide character length is zero bytes, zero is returned.

Standards conformance

These routines conform to:
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1).

memmove

copies characters between objects

Syntax

```
cc ... -lc

#include <string.h>

void *memmove(dest, src, count)
void *dest;
const void *src;
size_t count;
```

Description

The **memmove** function copies *count* characters from *src* to *dest*. If some regions of *src* and *dest* overlap, **memmove** ensures that the original *src* bytes in the overlapping region are copied before being overwritten.

Return value

The value of *dest*, the destination object.

Example

```
#include <stdio.h>
#include <string.h>

char Source[] = ">>>>>>>>><<<<<<<<<";
char Target[] = "Copy the characters in here: "
               "to see if memmove correctly moves "
               "in the string";
char *ToPrint;

main()
{
    printf("Target Before: %s\n\n", Target);
    ToPrint = memmove(&Target[32], Source, sizeof(Source));
    printf("Target After:  %s\n", Target);
}
```

Using **memmove**, string **Source** is copied into string **Target**. *sizeof* returns the size of the string, including the end-of-string character, effectively shortening **Target**.

See also

memcpy(S), **memcpy**(S)

Standards conformance

memmove is conformant with:

ANSI X3.159-1989 Programming Language — C.

memory: memccpy, memchr, memcmp, memcpy, memset

memory operations

Syntax

```
cc ... -lc

#include <memory.h>

char *memccpy (s1, s2, c, n)
char *s1, *s2;
int c, n;

void *memchr (s, c, n)
const void *s;
int c;

size_t n;
int memcmp (s1, s2, n)
const void *s1, *s2;
size_t n;

void *memcpy (s1, s2, n)
void *s1;
const *s2;
size_t n;

void *memset (s, c, n)
void *s;
int c;
size_t n;
```

SVID Syntax

```
char *memcpy (s1, s2, n)
void *s1;
const *s2;
size_t n;
```

Description

memccpy - copies characters from memory area

memchr - returns a pointer to the first occurrence of a character

memcmp - compares arguments

memcpy - copies *n* characters

memset - sets the first *n* characters in memory area

These functions operate as efficiently as possible on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

memccpy copies characters from memory area *s2* into *s1*, stopping after the first occurrence of character *c* has been copied, or after *n* characters have been copied, whichever comes first. It returns a pointer to the character after the copy of *c* in *s1*, or a NULL pointer if *c* was not found in the first *n* characters of *s2*.

memchr returns a pointer to the first occurrence of character *c* in the first *n* characters of memory area *s*, or a NULL pointer if *c* does not occur.

memcmp compares its arguments, looking at the first *n* characters only, and returns an integer less than, equal to, or greater than 0, according as *s1* is lexicographically less than, equal to, or greater than *s2*.

memcpy copies *n* characters from memory area *s2* to *s1*. It returns *s1*.

memset sets the first *n* characters in memory area *s* to the value of character *c*. It returns *s*.

For user convenience, all these functions are declared in the optional *<memory.h>* header file.

Notes

memcmp is implemented by using the most natural character comparison on the machine. Thus the sign of the value returned when one of the characters has its high order bit set is not the same in all implementations and should not be relied upon.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

Standards conformance

memcpy is conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

memchr, **memcmp**, **memcpy** and **memset** are conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
ANSI X3.159-1989 Programming Language — C;
and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

The `<memory.h>` header file has been withdrawn from XPG3.

(S)

menu

CRT menu routines

Syntax

```
cc ... -lmenu -lcurses

#include <menu.h>

typedef void (*PTF_void)();

ITEM * current_item (m)
MENU *m;

int free_menu (m)
MENU *m;

int item_index (i)
ITEM * i;

PTF_void item_init (m)
MENU *m;

PTF_void item_term (m)
MENU *m;

int menu_back (m)
MENU *m;

int menu_driver (m, c)
MENU *m;
int c;

int menu_fore (m)
MENU *m;

void menu_format (m, rp, cp)
MENU *m;
int *rp, *cp;

int menu_grey (m)
MENU *m;

PTF_void menu_init (m)
MENU *m;

ITEM * menu_items (m)
MENU * m;
```

```

char *menu_mark (m)
MENU *m;

OPTIONS menu_opts (m)
MENU *m;

int menu_opts_off (m, o)
MENU *m;
OPTIONS o;

int menu_opts_on (m, o)
MENU *m;
OPTIONS o;

int menu_pad (m)
MENU *m;

char *menu_pattern (m)
MENU *m;

WINDOW *menu_sub (m)
MENU *m;

PTF_void menu_term (m)
MENU *m;

char *menu_userptr (m)
MENU *m;

WINDOW *menu_win (m)
MENU *m;

MENU *new_menu (ip)
ITEM * ip;

int pos_menu_cursor (m)
MENU *m;

int post_menu (m)
MENU *m;

int scale_menu (m, rp, cp)
MENU *m;
int *rp, *cp;

int set_current_item (m, i)
MENU *m;
ITEM * item;

int set_item_init (m, f)
MENU *m;
ITEM * item;
PTF_void f;

```

(S)

```
int set_item_term (m, f)
MENU *m;
PTF_void f;

int set_menu_back (m, c)
MENU *m;
int c;

int set_menu_fore (m, c)
MENU *m;
int c;

int set_menu_format (m, c, r)
MENU *m;
int c, r;

int set_menu_grey (m, c)
MENU *m;
int c;

int set_menu_init (m, f)
MENU *m;
PTF_void f;

int set_menu_items (m,i)
MENU * m;
ITEM * i;

int set_menu_mark (m, n)
MENU *m;
char *n;

int set_menu_opts (m, o)
MENU *m;
OPTIONS o;

int set_menu_pad (m, c)
MENU *m;
int c;

int set_menu_pattern (m, n)
MENU *m;
char *n;

int set_menu_sub (m, w)
MENU *m;
WINDOW *w;

int set_menu_term (m, f)
MENU *m;
PTF_void f;

int set_menu_userptr (m, n)
MENU *m;
char *n;
```

```

int set_menu_win (m, w)
MENU *m;
WINDOW *w;

int set_top_row (m, c)
MENU *m;
int *c;

int top_row (m)
MENU *m;

int unpost_menu (m)
MENU *m;

```

Description

These routines allow you to create, display, and access menus. Menus can be displayed on any display device supported by the `curses(S)` library. Once you compile your program including the `MENU` header file `menu.h`, you should link it with the `MENU` and `curses` library routines.

Functions

The following is a list of `MENU` routines.

new_menu (*ip*) creates a new menu connected to the given item pointer array and returns a pointer to the new menu.

free_menu (*m*) disconnects the menu from its associated item pointer array and free the storage allocated for the menu.

set_menu_items (*m*, *i*) changes the item pointer array connected to the given menu to item pointer array *i*.

menu_items (*n*) returns a pointer to the item pointer array connected to menu *m*.

set_menu_format (*m*, *c*, *r*) sets the maximum number of rows and columns of items that may be displayed at one time on a menu.

menu_format (*m*, *rp*, *cp*) returns the maximum number of rows and columns that may be displayed at one time on a menu. *rp* and *cp* are pointers to the values used to return these numbers.

The mark string distinguishes selected items in a multi-valued menu and the current item in a single-valued menu. **set_menu_mark** (*m*, *n*) sets the menu's mark string to *n*.

menu_mark (*m*) returns a pointer to the menu's mark string.

scale_menu (*m*, *rp*, *cp*) returns the minimum window size necessary for the given menu. *rp* and *cp* are pointers to the locations used to return the number of rows and columns for the menu.

set_menu_win (*m*, *w*) sets window *w* as the window of menu *m*.

menu_win (*m*) returns a pointer to the menu's window.

set_menu_sub (*m*, *w*) sets window *w* as the subwindow of menu *m*.

menu_sub (*m*) returns a pointer to the menu's subwindow.

set_menu_fore (*m*, *c*) sets the menu's foreground attribute—the display attribute for the current item (if selectable) on single-valued menus and for selected items on multi-valued menus. This display attribute is a **curses** visual attribute. By default, this attribute is **A_STANDOUT**.

menu_fore (*m*) returns the menu foreground attribute.

set_menu_back (*m*, *c*) sets the menu's background attribute—the display attribute for unselected, yet selectable, items. This display attribute is a **curses** visual attribute. By default, this attribute is **A_NORMAL**.

menu_back (*m*) returns the menu background attribute.

set_menu_grey (*m*, *c*) sets the menu's grey attribute—the display attribute for nonselectable items in multi-valued menus. This display attribute is a **curses** visual attribute. By default, this attribute is **A_UNDERLINE**.

menu_grey (*m*) returns the menu's grey attribute.

The pad character is the character that fills the space between a menu item's name and description, if any. **set_menu_pad** (*m*, *c*) sets the pad character for menu *m* to *c*.

menu_pad (*m*) returns the menu's pad character.

post_menu (*m*) writes the menu in the menu's subwindow.

unpost_menu (*m*) erases the menu from its associated subwindow.

The workhorse of the menu subsystem, **menu_driver** (*m*, *c*) checks if the character *c* is a menu request or data. If it is a request, the menu driver executes the request and reports the result. If it is data (a printable ASCII character), it enters the data into the current position in the current field. If the character is not recognized, the menu driver assumes it is an application-defined command and returns **E_UNKNOWN_COMMAND**.

The following **set_** functions enable you to establish application routines to be executed automatically at initialization and termination points in your form application. You need not specify any application-defined initialization or termination routines at all, but they may be helpful for displaying messages or page numbers and other chores.

set_item_init (*m*, *f*) sets the application-defined function *f* to be called when the menu is posted and just after the current item changes.

item_init (*m*) returns a pointer to the item initialization routine, if any, called when the menu is posted and just after the current item changes.

set_item_term (*m*, *f*) sets function *f* to be called when the menu is unposted and just before the current item changes.

item_term (*m*) returns a pointer to the termination function, if any, called when the menu is unposted and just before the current item changes.

set_menu_init (*m*, *f*) sets the application-defined function *f* to be called when the menu is posted and just after the top row changes on a posted menu.

menu_init (*m*) returns a pointer to the menu's initialization routine, if any, called when the menu is posted and just after the top row changes on a posted menu.

set_menu_term (*m*, *f*) sets the application-defined function *f* to be called when the menu is unposted and just before the top row changes on a posted menu.

menu_term (*m*) returns a pointer to the menu's termination routine, if any, called when the menu is unposted and just before the top row changes on a posted menu.

The current item is the item where the cursor is currently positioned. **set_current_item** (*m*, *i*) sets the current menu item to the given item.

current_item (*m*) returns a pointer to the current item.

item_index (*i*) returns the index to the given item in the item pointer array.

set_top_row (*m*, *c*) sets the top of the menu to the named row. The leftmost item on the new top row becomes the current item.

top_row (*m*) returns the number of the menu row currently displayed at the top of the given menu.

pos_menu_cursor (*m*) moves the menu window's cursor to the correct position to resume menu processing.

Every menu has a pattern buffer to match entered data with menu items. **set_menu_pattern** (*m*, *p*) sets the pattern buffer to the given pattern and tries to find the first item that matches the pattern. If it does, the matching item becomes the current item. If not, the current item does not change.

menu_pattern (*m*) returns the string in the pattern buffer of the given menu.

Every menu has an associated user pointer that you can use to store pertinent information.

set_menu_userptr (*m*, *n*) sets the menu's user pointer.

menu_userptr (*m*) returns the menu's user pointer.

set_menu_opts (*m*, *o*) turns on the named options for the menu and turns off all its remaining options. Options are boolean values. Menu options are O_ONEVALUE, O_SHOWDESC, O_ROWMAJOR, O_IGNORECASE, and O_SHOWMATCH.

menu_opts (*m*) returns the menu's option setting.

menu_opts_on (*m*, *opts*) turns on the named options for the menu.

menu_opts_off (*m*, *opts*) turns off the named options for the menu.

Diagnostics

The following values are returned by one or more routines that return an integer.

E_BAD_STATE	routine called from an inappropriate routine
E_BAD_ARGUMENT	an incorrect argument was passed to the routine
E_CONNECTED	one or more items are connected to another menu
E_OK	routine returned normally
E_NO_MATCH	no match occurred
E_NO_ROOM	menu does not fit within its subwindow
E_NOT_CONNECTED	no items are associated with the menu
E_NOT_POSTED	menu has not yet been posted
E_NOT_SELECTABLE	item cannot be selected
E_POSTED	menu is already posted
E_REQUEST_DENIED	menu driver could not process the request
E_SYSTEM_ERROR	system error
E_UNKNOWN_COMMAND	unrecognizable request was given to the driver

See also

curses(S), field(S), fieldtype(S), form(S), item(S), panel(S), tam(S)

Standards conformance

menu is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

(S)

mkdir

make a directory

Syntax

```
cc ... -lc

#include <sys/types.h>
#include <sys/stat.h>

int mkdir (path, mode)
char *path;
mode_t mode;
```

Description

The routine **mkdir** creates a new directory with the name **path**. The argument **mode** specifies the initial mode of the new directory. The protection bits of the argument **mode** are modified by the process file mode creation mask (see **umask(S)**). The value of the argument **mode** should be the logical OR of the values of the desired permissions:

Name	Description
S_IEXEC	Execute (search) by owner
S_IREAD	Read by owner
S_IRGRP	Read by group
S_IROTH	Read by others (that is, anyone else)
S_IWGRP	Write by group
S_IWOTH	Write by others
S_IWRITE	Write by owner
S_IXGRP	Execute (search) by group
S_IXOTH	Execute (search) by others

The directory's owner ID is set to the process's effective user ID. The directory's group ID is set to the process's effective group ID. The newly created directory is empty with the possible exception of entries for "." and "..". **mkdir** fails and no directory is created if one or more of the following is true:

[EACCES]	Either a component of the path prefix denies search permission, or write permission is denied on the parent directory of the directory to be created.
[EEXIST]	The named file already exists.
[EFAULT]	<i>path</i> points outside the allocated address space of the process.
[EIO]	An I/O error has occurred while accessing the file system.
[EMLINK]	The maximum number of links to the parent directory would be exceeded.
[EMULTIHOP]	Components of <i>path</i> require hopping to multiple remote machines.
[ENOENT]	1. A component of the path prefix does not exist. 2. The path is longer than the maximum allowed.
[ENOLINK]	<i>path</i> points to a remote machine and the link to that machine is no longer active.
[ENOTDIR]	A component of the path prefix is not a directory.
[EROFS]	The path prefix resides on a read-only file system.

Diagnostics

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

Standards conformance

mkdir is conformant with:

X/Open Portability Guide, Issue 3, 1989;
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

mkfifo

make a FIFO special file

Syntax

```
cc ... -lc

#include <sys/types.h>
#include <sys/stat.h>

int mkfifo (path, mode)
char *path;
mode_t mode;
```

Description

The **mkfifo** routine creates a new FIFO special file named by the pathname specified by *path*. The permissions of the new FIFO are initialized from *mode*. The file permission bits of the *mode* argument are modified by the process's file creation mask (see **umask(S)**). When bits in *mode* other than the file permission bits are set, the effect is implementation-defined.

The FIFO's owner ID is set to the process's effective user ID. The FIFO's group ID is set to the group ID of the directory in which the FIFO is being created or to the process's effective group ID.

Upon successful completion, the **mkfifo** function marks the *st_atime*, *st_ctime*, and *st_mtime* fields of the file for update. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the new entry are marked for update.

Upon successful completion a value of zero is returned. Otherwise, a value of -1 is returned, no FIFO is created, and **errno** is set to indicate the error.

Diagnostics

If any of the following conditions occur, the **mkfifo** function returns -1 and sets **errno** to the corresponding value:

[EACCES]	A component of the path prefix denies search permission.
[EEXIST]	The named file already exists.
[ENAMETOOLONG]	The length of the <i>path</i> string exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

[ENOENT]	A component of the path prefix does not exist or the <i>path</i> argument points to an empty string.
[ENOSPC]	The directory that would contain the new file cannot be extended or the file system is out of file allocation resources.
[ENOTDIR]	A component of the path prefix is not a directory.
[EROFS]	The named file resides on a read-only file system.

See also

chmod(S), exec(S), pipe(S), stat(S), umask(S)

Standards conformance

mkfifo is conformant with:

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

mknod

make a directory or a special or ordinary file or a FIFO

Syntax

```
cc ... -lc

#include <sys/types.h>
#include <sys/stat.h>

int mknod (path, mode, dev)
char *path;
int mode, dev;
```

Description

The **mknod** system call creates a new file named by the pathname pointed to by *path*. The mode of the new file is initialized from *mode*. Where the value of *mode* is interpreted as follows:

0170000 file type; one of the following:

- 0010000 fifo special
- 0020000 character special
- 0040000 directory
- 0060000 block special
- 0100000 ordinary file
- 0000000 out-of-service inode

0004000 set user ID on execution

00020#0 set group ID on execution if # is 7, 5, 3, or 1
enable mandatory file/record locking if # is 6, 4, 2, or 0

0001000 save text image after execution

0000777 access permissions; constructed from the following:

- 0000400 read by owner
- 0000200 write by owner
- 0000100 execute (search on directory) by owner
- 0000070 read, write, execute (search) by group
- 0000007 read, write, execute (search) by others

Symbolic constants defining the value of the argument *mode* are in the `<sys/stat.h>` header file and should be used to construct *mode*. The value of the argument *mode* should be the logical OR of the values of the desired permissions:

<i>Name</i>	<i>Description</i>
S_ENFMT	record-locking enforced
S_IFBLK	block-special
S_IFCHR	character-special
S_IFDIR	directory node
S_IFIFO	FIFO-special
S_IFMT	file type; one of the following:
S_IFREG	ordinary-file
S_IRGRP	read by group
S_IROTH	read by others (that is, anyone else)
S_IRUSR	read by owner
S_ISGID	set group-ID on execution
S_ISUID	set user-ID on execution
S_ISVTX	(reserved)
S_IWGRP	write by group
S_IWOTH	write by others
S_IWUSR	write by owner
S_IXGRP	execute (search) by group
S_IXUSR	execute (search) by owner
S_IXOTH	execute (search) by others

The owner ID of the file is set to the effective user ID of the process. The group ID of the file is set to the effective group ID of the process.

Values of *mode* other than those above are undefined and should not be used. The low-order 9 bits of *mode* are modified by the process's file mode creation mask: all bits set in the process's file mode creation mask are cleared (see `umask(S)`). If *mode* indicates a block or character special file, *dev* is a configuration-dependent specification of a character or block I/O device. If *mode* does not indicate a block special or character special device, *dev* is ignored.

The **mknod** routine may be invoked only by the super user for file types other than FIFO special.

The **mknod** routine fails and the new file is not created if one or more of the following is true:

[EEXIST]	The named file exists.
[EFAULT]	<i>path</i> points outside the allocated address space of the process.
[EINTR]	A signal was caught during the mknod system call.
[EMULTIHOP]	Components of <i>path</i> require hopping to multiple remote machines.
[ENOENT]	A component of the path prefix does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOLINK]	<i>path</i> points to a remote machine and the link to that machine is no longer active.
[ENOSPC]	No space is available.
[EPERM]	The effective user ID of the process is not super user.
[EROFS]	The directory in which the file is to be created is located on a read-only file system.

Diagnostics

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

Warning

If **mknod** is used to create a device in a remote directory (Remote File Sharing), the major and minor device numbers are interpreted by the server.

See also

chmod(S), **exec(S)**, **fs(F)**, **mkdir(C)**, **umask(S)**

Standards conformance

mknod is conformant with:

AT&T SVID Issue 2, but has been withdrawn from XPG3

mktemp

make a unique filename

Syntax

```
cc ... -lc

char *mktemp (template)
char *template;
```

Description

The **mktemp** function replaces the contents of the string pointed to by *template* by a unique filename, and returns the address of *template*. The string in *template* should look like a filename with six trailing "X"s; **mktemp** replaces the "X"s with a letter and the current process ID. The letter is chosen so that the resulting name does not duplicate an existing file.

Diagnostics

The **mktemp** function assigns to *template* the null string if it cannot create a unique name.

See also

getpid(S), **tmpfile(S)**, **tmpnam(S)**

Standards conformance

mktemp is conformant with:

AT&T SVID Issue 2, but has been withdrawn from XPG3;
and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

mktime

converts local time to calendar time

Syntax

```
cc ... -lc

#include <time.h>

time_t mktime (timeptr)
struct tm *timeptr;           /* Local time structure */
```

Description

The **mktime** function converts the local time into a calendar value. The *timeptr* argument points to a structure that contains the local time. The structure is described in the reference page for **asctime**. The converted time has the same encoding as the values returned by the *time* function. The original values of the `tm_wday` and `tm_yday` components of the *timeptr* structure are ignored, and the original values of the other components are not restricted to their normal ranges.

If successful, **mktime** sets the values of `tm_wday` and `tm_yday` appropriately, and sets the other components to represent the specified calendar time, but with their values forced to the normal ranges; the final value of `tm_mday` is not set until `tm_mon` and `tm_year` are determined.

Notes

Note that the **gmtime**, **mktime**, and **localtime** functions use a single statically allocated buffer for the conversion. Each call to one of these routines destroys the result of the previous call.

Return value

The **mktime** function returns the specified calendar time encoded as a value of type `time_t`. If the calendar time cannot be represented, the function returns the value -1 (`time_t`).

See also

ctime(S), **difftime(S)**, **nl_cxtime(S)**, **stime(S)**, **time(S)**

Standards conformance

mktime is conformant with:

ANSI X3.159-1989 Programming Language — C.

Example

```
#include <time.h>
#include <stdio.h>

struct tm when;
time_t now;
time_t result;
int days;
main()
{
    printf("How many days to look ahead? ");
    scanf("%d", &days);

    time(&now);
    when = *localtime(&now);
    when.tm_mday = when.tm_mday + days;
    if ((result = mktime(&when)) != (time_t)-1)
        printf("\n%d days from now the time will be %s",
            days, asctime(&when));
    else
        perror("mktime failed");
}
```

The example above takes a number of days as input and returns the future time and date on the specified number of days ahead.

(S)

ml_send

Simple mail submission (ml_)

Syntax

```
cc ... -lmmdf

#include <util.h>
#include <stdio.h>

ml_ladr (return, sender, from, subject, address)
    int return, sender;
    char *from, *subject, *address;

ml_adr (address)
    char *address;

ml_aend ()

ml_cc ()

ml_end (end-stat)
    int end-stat;

ml_file (fil-text)
    FILE *file-text;

ml_init (return, sender, from, subject)
    int return, sender;
    char *from, *subject;

ml_tinit ()

ml_to ()

ml_txt (text)
    char *text;
```

Description

These procedures may be used for simple mail submission, providing a simple interface to the `mm_` package described in `mmdf(S)`. The basic sequence for using these calls is:

```
# include <util.h>

ml_init (YES, NO, "My Name", "The Subject");
ml_adr ("destination address 1");
ml_adr ("destination address 2");
...
ml_aend ();
ml_tinit ();
ml_txt ("Some opening text");
ml_txt ("maybe some more text");
ml_file (stdio-file-stream-descriptor-of-file-to-include);
if (ml_end (OK)) != OK
{   error-handling code }
```

Arguments that are to be defaulted should be zero. All routines return a status value of OK or NOTOK.

`ml_init`'s arguments specify:

- a) Whether return-mail (to the sender) should be allowed (1=yes, 0=no).
- b) Whether a Sender field should be used to specify the correct sender (contingent on next argument).
- c) Text for the From field.
- d) Text for the Subject field.

If (b) is NO, then (c)'s text must contain the correct sender information, or else just contain the "name" portion of the address.

The `ml_to` and `ml_cc` calls are used to switch between "To:" and "cc:" addresses. Normally, only "To:" addresses are used and, for this, no `ml_to` call is needed.

The `ml_adr` call specifies the text of one address. An "address" is whatever is valid for your system as if you were typing it to the `checkaddr(ADM)` command. `ml_aend` signals the end of address specification, and `ml_tinit` signals the start of message text submission.

The `ml_txt` call directly enters raw text. The `ml_file` call passes a stdio file pointer, to be used for file text submission; all of the (remaining) contents of the file are submitted. Any number of `ml_txt` and `ml_file` calls may be made and they may be freely mixed. They just append text to the message. The text must contain its own newlines.

The `ml_1adr` call is provided as a convenient way to initiate mail when there is only one addressee. Its arguments are the same as for `ml_init`, except there is an extra argument for the address string. It may be used, with the earlier example, to replace the `ml_init`, `ml_addr`, `ml_aend`, and `ml_tinit` calls.

See also

`checkaddr(ADM)`, `llog(S)`, `mmdf(S)`, `phs(S)`, `submit(ADM)`, `tai(S)`

Standards conformance

The `ml_send` procedures are not part of any currently supported standard; they were developed at the University of Delaware and are used by permission.

mmdf

M MDF mail submission and pickup

Syntax

cc ... -lmmdf

```
#include <util.h>
#include <mmdf.h>
```

```
mm_end (type)                /* Done with mail */
int type;                    /* OK or NOTOK */

mm_init()                    /* Get ready */

mm_pkend ()                  /* Done with pickup */

mm_pkinit (chname)           /* Initialize for pickup */
char *chname;                /* Channel being picked up */

mm_radr (host, adr)          /* Read an address */
char *host, *adr;

mm_rinit (info, retadr)      /* Message initialization */
char *info, *retadr;

mm_rrec (linebuf, len)       /* Read one "record" */
char *linebuf;               /* sizeof buf on call, */
int *len;                    /* nread on return */

mm_rrply (valstr, len)       /* Get a reply */
struct rp_bufstruct *valstr;
int *len;

mm_rstm (buffer, len)        /* Buffered text read */
char *buffer;                /* sizeof buf on call, */
int *len;                    /* nread on return */

mm_rtxt (buffer, len)        /* Buffered msg text read */
char *buffer;                /* sizeof buf on call, */
int *len;                    /* nread on return */

mm_sbend ()                  /* Done with submission */

mm_sbinit ()                 /* Initialize submission */

mm_wadr (host, adr)          /* Write one address */
char *host, *adr;            /* adr@host, if host is null */
```



```

mm_waend ( )                                /* End of address list */

mm_winit (vianet, info, retadr)             /* Initialize for one message */
char *vianet, *info, *retadr;              /* if == 0, use 'r' or 's' in info */

mm_wrec (linebuf, len)                      /* Write a record/packet */
char *linebuf;
int len;

mm_wrply (valstr, len)                      /* Send a reply */
struct rp_bufstruct *valstr;
int len;

mm_wstm (buffer, len)                      /* Buffered text write */
char *buffer;
int len;

mm_wtend ( )                               /* End of message text */

mm_wtxt (buffer, len)                      /* Buffered msg text write */
char *buffer;
int len;

```

Description

mm_end - indicates success or failure of MMDF conversation
mm_init - initializes conversation with MMDF mail system
mm_pkend - ends MMDF pickup
mm_pkinit - initializes an MMDF pickup conversation
mm_radr - reads an MMDF address
mm_rinit - reads initialization information for MMDF message
mm_rrec - reads an MMDF conversation record
mm_rrply - reads a reply from the other MMDF process
mm_rstm - reads a buffered block of MMDF text
mm_rtxt - reads MMDF message text
mm_sbend - ends MMDF submission
mm_sbinit - initializes for an MMDF submission
mm_wadr - writes an MMDF address
mm_waend - ends MMDF address list
mm_winit - sends MMDF initialization information for a message
mm_wrec - writes MMDF record
mm_wrply - writes MMDF reply
mm_wstm - writes buffered MMDF text

mm_wtend - signals end of MMDF message text

mm_wtxt - writes block of MMDF message text

The **mm** package is intended to simplify a user program's task of interacting with MMDF. This is primarily used for message submission (posting). The basic sequence of calls is to initialize the package, initialize for the submission of one message, indicate the addresses for the message and receive validations for them, send the message text, terminate the message and receive a final validation for the message, and then optionally repeat submission from the point of initializing for submission.

Note that the **mm** package depends on certain variables having been initialized. Call **mmdf_init** before using this package.

It is important to note the relationship between the info and sender arguments to **mm_winit**. If the info strings contains either 'r' or 's', then the sender must be a (char *)0.

The following **mmdf** routines allow the user to carry on a conversation with the mail system to do mail pickup and submission:

mm_end	Indicates success or failure of a mail conversation.
mm_init	Initializes for a conversation with the mail system. This is needed before mm_sbinit or mm_pkinit system calls, and may include multiple submissions or pickups.
mm_pkend	Indicates end of pickup.
mm_pkinit	Initializes a pickup conversation.
mm_radr	Reads an address. This is the "response" to mm_wadr .
mm_rinit	Reads initialization information for a message. This is the "response" to mm_winit .
mm_rrec	Reads a conversation record.
mm_rrply	Reads a reply from the other processes.
mm_rstm	Reads a buffered block of text.
mm_rtxt	Reads message text.
mm_sbend	Ends submission.
mm_sbinit	Initializes for submission of message(s). mm_init must be called first.
mm_wadr	Writes an address.
mm_waend	Ends address list.
mm_winit	Sends initialization information for a message.
mm_wrec	Writes record.
mm_wrply	Writes a reply which will be read by mm_reply .

mm_wstm Writes buffered text.
mm_wtend Signifies end of message text.
mm_wtxt Writes a block of message text.

See also

llog(S), ml_send(S), phs(S), tai(S)

Standards conformance

The **mm** package is not part of any currently supported standard; it was developed at the University of Delaware and is used by permission.

Example

The following depicts a typical use of calls to **mm**. (No attempt is made to provide for the way a user might choose to acquire the address and text information needed for the variables **sender**, **host**, **adr**, or **buffer**.)

```
send_manage ()
{
    /* submit a batch of messages */
    if (rp_isbad (mm_init()) ||
        rp_isbad (mm_sbinit()))
        abort-submit;
    /* initialize for submission */

    while (more-messages)
    {
        /* process each message */
        if (rp_isbad (do_a_message ()))
            abort-submit;
    }
    return (RP_OK);
}
```

```

do_a_message ()
{
    /* process a single message */
    struct rp_bufstruct thereply;
    int len;

    if (rp_isbad (mm_winit("", "mv", sender)) ||
        rp_isbad (mm_rrply(&thereply, &len) ||
        rp_isbad (thereply.rp_val))
        abort-submit;
        /* initialize the message */

    while (more_addresses)
    {
        /* step through address list */
        if (rp_isbad (do_an_address ()))
            abort-submit;
    }
    if (rp_isbad (mm_waend ()))
        abort-submit;
        /* indicate no more addresses */

    return (do_text ());
}

```

(S)

```

do_an_address ( )
{
    /* process a single address */
    struct rp_bufstruct thereply;
    int len;

    /* somehow begin with a host and adr */

    if (rp_isbad (mm_wadr(host, adr)) ||
        rp_isbad (mm_rrply (&thereply, &len)))
        { abort-submit; }
    /* send address & get status */

    switch (rp_gval (thereply.rp_val))
    {
        /* was address acceptable? */
        case RP_AOK:      /* yes */
        case RP_DOK:
            { note-the-acceptance }
            break;

        case RP_NO:      /* not acceptable */
        case RP_USER:
        case RP_NDEL:
        case RP_AGN:
        case RP_NS:
        case RP_NOOP:
        case RP_PARM:
            { note-failure; }
            break;

        default:          /* unexpected reply value */
            { probably-should-treat-as-illegal-value; }
    }
    return (RP_OK);
}

```

```

do_text ()
{
    struct rp_bufstruct thereply;
    int len;

    len = sizeof buffer;
    while (more-text)
        if (rp_isbad (mm_wtxt (buffer, len)))
            abort-submit;
        len = sizeof buffer;
    }      /* send a chunk of text */

    if (rp_isbad (mm_wtend ()))
        abort-submit;

    if (rp_isbad (mm_rrply (&thereply, &len)))
        abort-submit;

    switch (rp_gval (thereply.rp_val))
    { /* was text acceptable? */
        case RP_OK:
        case RP_MOK:
            { note-acceptance; }
            break;

        case RP_NO:
        case RP_NDEL:
        case RP_AGN:
        case RP_NS:
        case RP_NOOP:
            { note-failure; }
            break;

        default:
            { treat-as-illegal-value }
            break;
    }
    return (RP_OK);
}

```

(S)

monitor

prepare execution profile

Syntax

```
cc ... -lc

#include <mon.h>

void monitor (lowpc, highpc, buffer, bufsize, nfunc)
int (*lowpc)( ), (*highpc)( );
WORD *buffer;
int bufsize, nfunc;
```

Description

An executable program created by **cc -p** automatically includes calls for **monitor** with default parameters; **monitor** need not be called explicitly except to gain fine control over profiling.

The **monitor** function is an interface to **profil(S)**. *lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a user-supplied array of *bufsize* WORDs (defined in the *<mon.h>* header file). **monitor** records a histogram of two items: periodically sampled values of the program counter and counts of calls to certain functions. This histogram is recorded in the buffer. The lowest address sampled is that of *lowpc* and the highest is just below *highpc*. *lowpc* may not equal 0 for this use of **monitor**. At most *nfunc*, call counts can be kept; only calls of functions compiled with the profiling option **-p** of **cc(CP)** are recorded.

For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use

```
extern etext;
...
monitor ((int (*)())2, &etext, buf, bufsize, nfunc);
```

etext lies just above all the program text; see **end(S)**.

To stop execution monitoring and write the results, use

```
monitor ((int (*)())0, 0, 0, 0, 0);
```

The **prof(CP)** command can then be used to examine the results.

The name of the file written by **monitor** is controlled by the environment variable **PROFDIR**. If **PROFDIR** does not exist, *mon.out* is created in the current directory. If **PROFDIR** exists but has no value, **monitor** does not do any

profiling and creates no output file. Otherwise, the value of `PROFDIR` is used as the name of the directory in which to create the output file. If `PROFDIR` is *dirname*, then the file written is in the format of:

dirname/pid.mon.out

where *pid* is the program's process ID. (When `monitor` is called automatically by compiling via `cc -p`, the file created is "*dirname/pid.progname*" where *progname* is the name of the program.)

Files

`mon.out`

Notes

The *dirname/pid.mon.out* form does not work; the *dirname/pid.progname* form (automatically called via `cc -p`) does work.

See also

`cc(CP)`, `end(S)`, `prof(CP)`, `profil(S)`

Standards conformance

monitor is conformant with:

AT&T SVID Issue 2, but has been withdrawn from XPG3. The `<mon.h>` header file has also been withdrawn from XPG3.

mount

mount a filesystem

Syntax

```
cc ... -lc

#include <sys/types.h>
#include <sys/mount.h>

int mount (spec, dir, mflag, fstyp, dataptr, datalen)
char *spec, *dir;
int mflag, fstyp;
char *dataptr;
int datalen;
```

Description

mount requests that a removable filesystem contained on the block special file identified by *spec* be mounted on the directory identified by *dir*. *spec* and *dir* are pointers to path names. *fstyp* is the filesystem type number. The **sysfs(S)** system call can be used to determine the filesystem type number. Note that if both the MS_DATA and MS_FSS flag bits of *mflag* are off, the filesystem type defaults to the root filesystem type. Only if either flag is on is *fstyp* used to indicate the filesystem type.

If the MS_DATA flag is set in *mflag*, the system expects the *dataptr* and *datalen* arguments to be present. Together they describe a block of file-system specific data at address *dataptr* of length *datalen*. This is interpreted by file-system specific code within the operating system and its format depends upon the filesystem type. A particular filesystem type may not require this data, in which case *dataptr* and *datalen* should both be zero. Note that MS_FSS is obsolete and is ignored if MS_DATA is also set, but if MS_FSS is set and MS_DATA is not, *dataptr* and *datalen* are both assumed to be zero.

Upon successful completion, references to the file *dir* refer to the root directory on the mounted filesystem.

The low-order bit of *mflag* is used to control write permission on the mounted filesystem; if 1, writing is forbidden, otherwise writing is permitted according to individual file accessibility.

mount may be invoked only by the super user. It is intended for use only by the **mount(ADM)** utility.

mount fails if one or more of the following is true:

[EBUSY]	1. <i>dir</i> is currently mounted on, is someone's current working directory, or is otherwise busy. 2. The device associated with <i>spec</i> is currently mounted. 3. There are no more mount table entries.
[EFAULT]	<i>spec</i> or <i>dir</i> points outside the allocated address space of the process.
[EINVAL]	The super block has an invalid magic number or the <i>fstyp</i> is invalid or <i>mflag</i> is not valid.
[EMULTIHOP]	Components of <i>path</i> require hopping to multiple remote machines.
[ENOENT]	Any of the named files does not exist.
[ENOLINK]	<i>path</i> points to a remote machine and the link to that machine is no longer active.
[ENOSPC]	The filesystem state in the super block is not FsOKAY and <i>mflag</i> requests write permission.
[ENOTBLK]	<i>spec</i> is not a block special device.
[NOTDIR]	1. A component of a path prefix is not a directory. 2. <i>dir</i> is not a directory.
[ENXIO]	The device associated with <i>spec</i> does not exist.
[EPERM]	The effective user ID is not super user.
[EREMOTE]	<i>spec</i> is remote and cannot be mounted.
[EROFS]	<i>spec</i> is write-protected and <i>mflag</i> requests write permission.

Diagnostics

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned, and *errno* is set to indicate the error.

See also

filesystem(FP), *fsname*(ADM), *labelit*(ADM), *mount*(ADM), *sysfs*(S), *umount*(S)

Standards conformance

mount is conformant with:

AT&T SVID Issue 2, but has been withdrawn from XPG3.

msgctl

message control operations

Syntax

```
cc ... -lc

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl (msqid, cmd, buf)
int msqid, cmd;
struct msqid_ds *buf;
```

Description

The **msgctl** system call provides a variety of message control operations as specified by *cmd*. The following *cmds* are available:

IPC_RMID Remove the message queue identifier specified by *msqid* from the system and destroy the message queue and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of super user, or to the value of *msg_perm.cuid* or *msg_perm.uid* in the data structure associated with *msqid*.

IPC_SET Set the value of the following members of the data structure associated with *msqid* to the corresponding value found in the structure pointed to by *buf*:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode /* only low 9 bits */
msg_qbytes
```

This *cmd* can only be executed by a process that has an effective user ID equal to either that of super user, or to the value of *msg_perm.cuid* or *msg_perm.uid* in the data structure associated with *msqid*. Only super user can raise the value of *msg_qbytes*.

IPC_STAT Place the current value of each member of the data structure associated with *msqid* into the structure pointed to by *buf*. The contents of this structure are defined in **Intro(S)**.

The **msgctl** system call fails if one or more of the following is true:

[EACCES]	The <i>cmd</i> argument is equal to IPC_STAT and (READ) operation permission is denied to the calling process.
[EFAULT]	The <i>buf</i> argument points to an illegal address.
[EINVAL]	1. The <i>msqid</i> argument is not a valid message queue identifier. 2. The <i>cmd</i> argument is not a valid command.
[EPERM]	1. The <i>cmd</i> argument is equal to IPC_RMID or IPC_SET. The effective user ID of the calling process is not equal to that of super user, or to the value of msg_perm.cuid or msg_perm.uid in the data structure associated with <i>msqid</i> . 2. The <i>cmd</i> argument is equal to IPC_SET, an attempt is being made to increase to the value of msg_qbytes, and the effective user ID of the calling process is not equal to that of super user.

Diagnostics

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned, and *errno* is set to indicate the error.

See also

Intro(S), msgget(S), msgop(S)

Standards conformance

msgctl is conformant with:

AT&T SVID Issue 2;
and X/Open Portability Guide, Issue 3, 1989.

msgget

get message queue

Syntax

```
cc ... -lc

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget (key, msgflg)
key_t key;
int msgflg;
```

Description

The **msgget** system call returns the message queue identifier associated with *key*.

A message queue identifier and associated message queue and data structure are created for *key* if one of the following is true:

The *key* argument is equal to **IPC_PRIVATE**.

The *key* argument does not already have a message queue identifier associated with it, and (*msgflg* & **IPC_CREAT**) is “true”.

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

msg_perm.cuid, *msg_perm.uid*, *msg_perm.cgid*, and *msg_perm.gid* are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of *msg_perm.mode* are set equal to the low-order 9 bits of *msgflg*.

msg_qnum, *msg_lspid*, *msg_lrpid*, *msg_stime*, and *msg_rtime* are set equal to 0.

msg_ctime is set equal to the current time.

msg_qbytes is set equal to the system limit.

The *msgget* system call fails if one or more of the following is true:

- [EACCES] A message queue identifier exists for *key*, but operation permission as specified by the low-order 9 bits of *msgflg* would not be granted.
- [EEXIST] A message queue identifier exists for *key* but $[(msgflg \& IPC\_CREAT) \& (msgflg \& IPC\_EXCL)]$ is "true".
- [ENOENT] A message queue identifier does not exist for *key* and $(msgflg \& IPC\_CREAT)$ is "false".
- [ENOSPC] A message queue identifier is to be created but the system-imposed limit on the maximum number of allowed message queue identifiers system wide would be exceeded.

Diagnostics

Upon successful completion, a non-negative integer, namely a message queue identifier, is returned. Otherwise, a value of -1 is returned, and *errno* is set to indicate the error.

See also

Intro(S), *msgctl*(S), *msgop*(S)

Standards conformance

msgget is conformant with:

AT&T SVID Issue 2;
and X/Open Portability Guide, Issue 3, 1989.

msgop

message operations

Syntax

```
cc ... -lc

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (msqid, msgp, msgsz, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz, msgflg;

int msgrcv (msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz;
long msgtyp;
int msgflg;
```

Description

msgsnd - sends a message

msgrcv - receives a message

The **msgsnd** system call is used to send a message to the queue associated with the message queue identifier specified by *msqid*. (WRITE) *msgp* points to a structure containing the message. This structure is composed of the following members:

```
long    mtype;          /* message type */
char    mtext[];        /* message text */
```

The *mtype* integer is positive and can be used by the receiving process for message selection (see **msgrcv** below). The array *mtext* is any text of length *msgsz* bytes. The *msgsz* argument can range from 0 to a system-imposed maximum.

msgflg specifies the action to be taken if one or more of the following is true:

- The number of bytes already on the queue is equal to *msg_qbytes*.
- The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are:

- If (*msgflg* & *IPC_NOWAIT*) is "true", the message is not sent and the calling process returns immediately.
- If (*msgflg* & *IPC_NOWAIT*) is "false", the calling process suspends execution until one of the following occurs:
 - The condition responsible for the suspension no longer exists, in which case the message is sent.
 - The *msgid* argument is removed from the system (see *msgctl*(S)). When this occurs, *errno* is set equal to *EIDRM*, and a value of -1 is returned.
 - The calling process receives a signal that is to be caught. In this case the message is not sent and the calling process resumes execution in the manner prescribed in *signal*(S).

msgsnd fails and no message is sent if one or more of the following is true:

- [EACCES] Operation permission is denied to the calling process (see *Intro*(S)).
- [EAGAIN] The message cannot be sent for one of the reasons cited above and (*msgflg* & *IPC_NOWAIT*) is "true".
- [EFAULT] *msgp* points to an illegal address.
- [EINVAL] *msgid* is not a valid message queue identifier.
mtype is less than 1.
msgsz is less than zero or greater than the system-imposed limit.

Upon successful completion, the following actions are taken with respect to the data structure associated with *msgid* (see *Intro*(S)):

- *msg_qnum* is incremented by 1.
- *msg_lspid* is set equal to the process ID of the calling process.
- *msg_stime* is set equal to the current time.

msgrcv reads a message from the queue associated with the message queue identifier specified by *msgid* and places it in the structure pointed to by *msgp*.
 (READ) This structure is composed of the following members:

```

long    mtype;          /* message type */
char    mtext[];        /* message text */

```

mtype is the received message's type as specified by the sending process.
mtext is the text of the message. *msgsz* specifies the size in bytes of *mtext*.
 The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and (*msgflg* & *MSG_NOERROR*) is "true". The truncated part of the message is lost and no indication of the truncation is given to the calling process.

msgtyp specifies the type of message requested as follows:

- If *msgtyp* is equal to 0, the first message on the queue is received.
- If *msgtyp* is greater than 0, the first message of type *msgtyp* is received.
- If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

msgflg specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:

- If (*msgflg* & IPC_NOWAIT) is "true", the calling process returns immediately with a return value of -1 and *errno* set to ENOMSG.
- If (*msgflg* & IPC_NOWAIT) is "false", the calling process suspends execution until one of the following occurs:
 - A message of the desired type is placed on the queue.
 - *msqid* is removed from the system. When this occurs, *errno* is set equal to EIDRM, and a value of -1 is returned.
 - The calling process receives a signal that is to be caught. In this case a message is not received and the calling process resumes execution in the manner prescribed in *signal(S)*.

msgrcv fails and no message is received if one or more of the following is true:

- [E2BIG] *mtext* is greater than *msgsz* and (*msgflg* & MSG_NOERROR) is "false".
- [EACCES] Operation permission is denied to the calling process.
- [EFAULT] *msgp* points to an illegal address.
- [EINVAL] *msqid* is not a valid message queue identifier.
msgsz is less than 0.
- [ENOMSG] The queue does not contain a message of the desired type and (*msgtyp* & IPC_NOWAIT) is "true".

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid*. *msg_qnum* is decremented by 1.

- *msg_lrp*id is set equal to the process ID of the calling process.
- *msg_rtime* is set equal to the current time.

Diagnostics

If `msgsnd` or `msgrcv` return due to the receipt of a signal, a value of -1 is returned to the calling process and `errno` is set to `EINTR`. If they return due to removal of *msgid* from the system, a value of -1 is returned and `errno` is set to `EIDRM`.

Upon successful completion, the return value is as follows:

- `msgsnd` returns a value of 0.
- `msgrcv` returns a value equal to the number of bytes actually placed into *mtext*.

Otherwise, a value of -1 is returned, and `errno` is set to indicate the error.

See also

`msgctl(S)`, `msgget(S)`, `signal(S)`

Standards conformance

`msgrcv` and `msgsnd` are conformant with:

AT&T SVID Issue 2;
and X/Open Portability Guide, Issue 3, 1989.

nap

suspends execution for a short interval

Syntax

```
cc ... -lx  
  
long nap (period)  
long period;
```

Description

The current process is suspended from execution for at least the number of milliseconds specified by *period*, or until a signal is received.

Return value

On successful completion, a long integer indicating the number of milliseconds actually slept is returned. If the process received a signal while napping, -1 is returned, and *errno* is set to *EINTR*.

Notes

This function is driven by the system clock, which in most cases has a granularity of 10 millisecond (1/100 second) units.

See also

`sleep(S)`

Standards conformance

`nap` is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

nice

change priority of a process

Syntax

```
cc ... -lc

int nice (incr)
int incr;
```

Description

The `nice` system call adds the value of *incr* to the `nice` value of the calling process. A process's `nice` value is a non-negative number for which a more positive value results in lower CPU priority.

A maximum `nice` value of 39 and a minimum `nice` value of 0 are imposed by the system. (The default `nice` value is 20.) Requests for values above or below these limits result in the `nice` value being set to the corresponding limit.

[EPERM] The `nice` system call fails and does not change the `nice` value if *incr* is negative or greater than 39, and the effective user ID of the calling process is not super user.

Diagnostics

Upon successful completion, `nice` returns the new `nice` value minus 20. Otherwise, a value of -1 is returned, and `errno` is set to indicate the error.

See also

`exec(S)`, `nice(C)`

Standards conformance

`nice` is conformant with:

AT&T SVID Issue 2;
and X/Open Portability Guide, Issue 3, 1989.

nl_cxtime, nl_ascxtime

format date and time

Syntax

```
cc ... -lintl

#include <time.h>

char *nl_cxtime (clock, format)
long *clock;
char *format;

char *nl_ascxtime (tm, format)
struct tm *tm;
char *format;
```

Description

nl_cxtime - converts UNIX epoch time to local time

nl_ascxtime - converts a **tm** structure to a 26-character string

The routines **nl_cxtime** and **nl_ascxtime** provide an alternative interface to the **ctime(S)** routines for programs written to the *X/OPEN Portability Guide* standard. However, both **nl_cxtime** and **nl_ascxtime** have been withdrawn from XPG3.

They provide the same functions as the **ctime(S)** routines **ctime** and **asctime**, except that a **strftime(S)** format string is provided to determine the layout of the resulting string.

The *X/OPEN Portability Guide* permits the following field descriptors only:

n	insert a newline character
t	insert a tab character
m	month of year - 01 to 12
d	day of month - 01 to 31
y	last 2 digits of year - 00 to 99
D	date as mm/dd/yy
H	hour - 00 to 23
M	minute - 00 to 59
S	second - 00 to 61
T	time as HH:MM:SS
j	day of year - 001 to 366
w	day of week - 0 to 6 (Sunday = 0)
a	abbreviated weekday - Sun to Sat
h	abbreviated month - Jan to Dec
r	time in AM/PM notation

If *format* is the empty (zero length) string, the date and time format (D_T_FMT) of the currently selected LC_TIME locale is used.

The maximum length of the resultant string is defined by the constant NL_TEXTMAX in the *limits.h* file.

See also

ctime(S), difftime(S), mktime(S), limits(FP), nl_init(S), stime(S), time(S), timtbl(M)

Standards conformance

nl_cxtime and nl_ascxtime are not part of any currently supported standard; they are extensions of AT&T System V provided by The Santa Cruz Operation, Inc.

(S)

nl_init

initializes native language support operation

Syntax

```
cc ... -lintl

int nl_init (lang)
char * lang;
```

Description

The `nl_init` routine is provided as an alternative interface to the `setlocale(S)` routine, for programs written to the *X/OPEN Portability Guide* standard. However, `nl_init` has been withdrawn from XPG3.

The `nl_init` routine initializes native language support operation for the language identified by *lang*, which is a pointer to a string containing settings of *language*, *territory* and *codeset* as defined for the LANG environment variable (see `environ(M)`).

Typically, `nl_init` is used to bind program operation to the user's specified language requirements, for example:

```
nl_init(getenv("LANG"));
```

A call to `nl_init` fails if the string pointed to by *lang* does not identify a valid *language/territory/codeset* combination. In this case, operation continues for the language identified on the last successful call.

Return value

If successful, `nl_init` returns 0 (zero). Otherwise, -1 is returned.

Notes

Calls to `nl_init` can be used to switch operation from one supported language to another. It is not necessary to call `nl_init` to set the initial language for an application; this is done by the automatic `setlocale` call at program startup.

The `nl_init` routine is not as flexible an interface as that provided by `setlocale`.

See also

environ(M), getenv(S), locale(M), setlocale(S)

Standards conformance

nl_init is an extension of AT&T System V provided by the Santa Cruz Operation.

nl_langinfo

language information

Syntax

```
cc ... -lintl

#include <nl_types.h>
#include <langinfo.h>

char *nl_langinfo (item)
nl_item item;
```

Description

The **nl_langinfo** routine returns a pointer to a null terminated string that contains information relevant to a particular language or cultural area identified by the last successful call to **setlocale** or **nlinit**. The constant names and values of *item* are defined in the file *langinfo.h*.

Example

The following statement sets a character pointer called *day* to point to the string **Dom** if the identified language is Portuguese or **Sun** if the identified language is English.

```
day = nl_langinfo(ABDAY_1)
```

If the parameter is not a valid *item*, **nl_langinfo** returns a pointer to an empty (null) string.

See also

environ(M),
langinfo(FP),
nl_init(S),
nl_types(FP), **setlocale**(S)

Standards conformance

nl_langinfo is an extension of AT&T System V provided by the Santa Cruz Operation.

nl_printf, nl_fprintf, nl_sprintf

formats native language output

Syntax

```
cc ... -lintl
```

```
int nl_printf (format [, arg...])
char *format;
```

```
int nl_fprintf (stream, format [, arg...])
FILE *stream;
char *format;
```

```
int nl_sprintf (s, format [, arg...])
char *s, *format;
```

Description

The functions `nl_printf`, `nl_fprintf` and `nl_sprintf` provide similar functionality to the standard `printf`, `fprintf` and `sprintf` routines (see `printf(S)`), with the difference that the conversion character `%` in the format string is replaced by the sequence `%digit$`, where `digit` is a decimal digit `n` in the range `(1-{ NL_ARGMAX })` (see `limits(FP)`). Conversions are applied to the `n`th argument in the argument list, rather than to the next unused argument.

The format passed to these functions can contain either form of conversion specification, namely `%` or `%digit$`, although the two forms cannot be mixed within a single format string. The format string should contain values for all the arguments specified in the argument list. The asterisk, `*`, cannot be used to indicate a field width in the format string.

Examples

The following `nl_printf` statement could be used to print the language independent date and time format:

```
nl_printf(format, weekday, month, day, hour, min);
```

For American usage, format could be a pointer to the string:

```
"%1$s, %2$s %3$d, %4$d:%5$.2d0
```

This would produce the following message:

Sunday, July 3, 10:02

For German usage, **format** could be a pointer to the string:

"%1\$s, %3\$d. %2\$s, %4\$d:%5\$.2d0

This would produce the message:

Sonntag, 3. Juli, 10:02

See also

limits(FP), nl_init(S), nl_scanf(S), printf(S)

Standards conformance

nl_fprintf, nl_printf and **nl_sprintf** are extensions of AT&T System V provided by the Santa Cruz Operation.

nl_scanf, nl_fscanf, nl_sscanf

converts formatted native language input

Syntax

```
cc ... -lintl

int nl_scanf (format [, pointer...])
char *format;

int nl_fscanf (stream, format [, pointer...])
FILE *stream;
char *format;

int nl_sscanf (s, format [, pointer...])
char *s, *format;
```

Description

nl_scanf - reads from the standard input stream

nl_fscanf - reads from the named input

nl_sscanf - reads from the character string

The functions **nl_scanf**, **nl_fscanf** and **nl_sscanf** provide similar functionality to the standard **scanf**, **fscanf** and **sscanf** routines (see **scanf(S)**), with the difference that the conversion character **%** in the format string is replaced by the sequence **%digit\$**, where **digit** is a decimal digit *n* in the range (1-{**NL_ARGMAX**}) (see **limits(FP)**). Conversions are applied to the *n*th argument in the argument list, rather than to the next unused argument.

The *format* passed to these functions can contain either form of conversion specification, such as **%** or **%digit\$**, although the two forms cannot be mixed within a single *format* string.

See also

limits(FP), **nl_init(S)**, **nl_printf(S)**, **scanf(S)**

Standards conformance

nl_fscanf, **nl_scanf** and **nl_sscanf** are extensions of AT&T System V provided by The Santa Cruz Operation.

nl_strcmp, nl_strncmp

compare native language strings

Syntax

```
cc ... -lintl

int *nl_strcmp (s1, s2)
char *s1, *s2;

int *nl_strncmp (s1, s2, n)
char *s1, *s2;
int n;
```

Description

nl_strcmp - compare native language strings

nl_strncmp - compare native *n* language strings

The routines **nl_strcmp** and **nl_strncmp** are provided as an alternative interface to the **strcoll(S)** routine, for programs written to the *X/OPEN Portability Guide* standard. However, these routines have been withdrawn from XPG3. They provide the same functions as the **string(S)** routines **strcmp** and **strncmp**, except that the language dependent collating information defined for the current **LC_COLLATE** locale is used to rank the strings, instead of the binary values of the machine character set.

In this way, the strings are compared according to the dictionary order of data, taking into account case and accent priority, 1-to-1, 1-to-2, 2-to-1 and don't care character mappings.

See also

collation(S), **coltbl(M)**, **nl_init(S)**, **string(S)**

Standards conformance

nl_strcmp and **nl_strncmp** are extensions of AT&T System V provided by the Santa Cruz Operation.

nlist

get entries from name list

Syntax

```
cc ... -lc

#include <nlist.h>

int nlist (filename, nl)
char *filename;
struct nlist *nl;
```

Description

The **nlist** function examines the name list in the executable file whose name is pointed to by *filename*, and selectively extracts a list of values and puts them in the array of **nlist** structures pointed to by *nl*. The name list *nl* consists of an array of structures containing names of variables, types, and values. The list is terminated with a null name; that is, a null string is in the name position of the structure. Each variable name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. The type field is set to 0 unless the file was compiled with the **-g** option. If the name is not found, both entries are set to 0. See **a.out(FP)** for a discussion of the symbol table structure.

This function is useful for examining the system name list kept in the file */unix*. In this way programs can obtain system addresses that are up to date.

Diagnostics

All value entries are set to 0 if the file cannot be read or if it does not contain a valid name list.

The **nlist** function returns -1 upon error; otherwise it returns 0.

Notes

The `<nlist.h>` header file is automatically included by `<a.out.h>` for compatibility. However, if the only information needed from `<a.out.h>` is for use of *nlist*, then including `<a.out.h>` is discouraged. If `<a.out.h>` is included, the line `"#undef n_name"` may need to follow it.

See also

a.out(FP)

Standards conformance

nlist is conformant with:
AT&T SVID Issue 2.

open

open for reading or writing

Syntax

```
cc . . . -lc

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open (path, oflag [, mode] )
char *path;
int oflag, mode;
```

Description

path points to a path name naming a file. The `open` system call opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*. For non-STREAMS files, *oflag* values are constructed by OR-ing flags from the following list (only one of the first three flags below can be used):

- O_RDONLY** Open for reading only.
- O_WRONLY** Open for writing only.
- O_RDWR** Open for reading and writing.
- O_APPEND** If set, the file pointer is set to the end of the file prior to each write.
- O_CREAT** If the file exists, this flag has no effect. Otherwise, the user ID of the file is set to the effective user ID of the process; the group ID of the file is set to the effective group ID of the process or to the group ID of the file's parent directory; and the low-order 12 bits of the file mode are set to the value of *mode*, modified as follows (see `creat(S)`):
 - All bits set in the file mode creation mask of the process are cleared (see `umask(S)`).
 - The "save text image after execution bit" of the mode is cleared (see `chmod(S)`).
- O_EXCL** Causes `open` to fail if the file exists. This can only be used if **O_CREAT** is also set.
- O_NOCTTY** If this flag is set and *path* identifies a terminal, then that terminal is prevented from becoming the controlling terminal for the process

- O_SYNC** When opening a regular file, this flag affects subsequent writes. If set, each **write(S)** waits for both the file data and file status to be physically updated.
- O_TRUNC** If the file exists, is a regular file, and is opened for writing, its length is truncated to 0. The mode, owner and group are left unchanged. It has no effect on FIFO special files or directories.
- O_NONBLOCK**
This flag may affect subsequent reads and writes (see **read(S)** and **write(S)**).

When opening a FIFO with **O_RDONLY** or **O_WRONLY** set:

If **O_NONBLOCK** is set:

An **open** for reading only returns without delay. An **open** for writing only returns an error if no process currently has the file open for reading.

If **O_NONBLOCK** is not set:

An **open** for reading only will block until a process opens the file for writing. An **open** for writing only blocks until a process opens the file for reading.

When opening a file associated with a communication line:

If **O_NONBLOCK** is set:

The **open** returns without waiting for carrier.

If **O_NONBLOCK** is not set:

The **open** blocks until carrier is present.

When opening a STREAMS file, *oflag* may be constructed from **O_NONBLOCK** or-ed with either **O_RDONLY**, **O_WRONLY** or **O_RDWR**. Other flag values are not applicable to STREAMS devices and have no effect on them. The value of **O_NONBLOCK** affects the operation of STREAMS drivers and certain system calls (see **read(S)**, **getmsg(S)**, **putmsg(S)**, and **write(S)**). For drivers, the implementation of **O_NONBLOCK** is device-specific. Each STREAMS device driver may treat this option differently.

Certain flag values can be set following **open** as described in **fcntl(S)**.

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is set to remain open across **exec** system calls (see **fcntl(S)**).

The named file is opened unless one or more of the following is true:

[EACCES]	• A component of the path prefix denies search permission. • <i>oflag</i> permission is denied for the named file. • The file does not exist and write permission is not set for the parent directory of the file to be created. • O_TRUNC is specified and write permission is not set for the file.
[EAGAIN]	The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file (see <code>chmod(S)</code>) and O_TRUNC is included in <i>oflag</i> .
[EEXIST]	O_CREAT and O_EXCL are set, and the named file exists.
[EFAULT]	<i>path</i> points outside the allocated address space of the process.
[EINTR]	A signal was caught during the open system call.
[EINVAL]	The value of <i>oflag</i> is not valid.
[EIO]	A hangup or error occurred during a STREAMS open.
[EISDIR]	The named file is a directory and <i>oflag</i> includes O_WRONLY or O_RDWR
[EMFILE]	OPEN_MAX descriptors are open in the current process.
[EMULTIHOP]	Components of <i>path</i> require hopping to multiple remote machines.
[ENAMETOOLONG]	The length of <i>path</i> exceeds PATH_MAX or the length of a pathname component exceeds NAME_MAX while _POSIX_NO_TRUNC is in effect.
[ENFILE]	The system file table is full, SYS_OPEN files are open in the system.
[ENOENT]	• O_CREAT is not set and the named file does not exist. • O_CREAT is set and either the <i>path</i> prefix does not exist or the <i>path</i> argument is a null string.
[ENOLINK]	<i>path</i> points to a remote machine, and the link to that machine is no longer active.
[ENOMEM]	The system is unable to allocate a send descriptor.
[ENOSPC]	O_CREAT and O_EXCL are set, and the file system is out of inodes.
[ENOSR]	Unable to allocate a <i>stream</i> .
[ENOTDIR]	A component of the path prefix is not a directory.

[ENXIO]	• The named file is a character special or block special file, and the device associated with this special file does not exist. • O_NONBLOCK is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading. • A STREAMS module or driver open routine failed.
[EROFS]	The named file resides on a read-only file system and <i>oflag</i> includes either O_WRONLY , O_RDWR , O_CREAT (if the file does not exist) or O_TRUNC .
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed and <i>oflag</i> includes either O_WRONLY or O_RDWR .

Limitation

A maximum of three XENIX or UNIX partitions may be opened on a hard disk. Even if any of these are subsequently closed, no more may be opened until reboot.

Diagnostics

Upon successful completion, the file descriptor is returned. Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

See also

chmod(S), close(S), creat(S), dup(S), fcntl(S), getmsg(S), lseek(S), putmsg(S), read(S), umask(S), write(S)

Standards conformance

open is conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

opensem

opens a semaphore

Syntax

```
cc ... -lx

int opensem (sem_name)
char *sem_name;
```

Description

opensem opens a semaphore named by *sem_name* and returns the unique semaphore identification number *sem_num* used by **waitsem** and **sigsem**. **creatsem** should always be called to initialize the semaphore before the first attempt to open it.

System compatibility

opensem can only be used to open semaphores created under UNIX version 3.0, not for UNIX System V semaphores.

Diagnostics

opensem returns a value of -1 if an error occurs. If the semaphore named does not exist, **errno** is set to **ENOENT**. If the file specified is not a semaphore file (i.e., a file previously created by a process using a call to **creatsem**), **errno** is set to **ENOTNAM**. If the semaphore has become invalid due to inappropriate use, **errno** is set to **ENAVAIL**.

Notes

This feature is a XENIX specific enhancement which may not be present in all UNIX implementations.

Warning

It is not advisable to open the same semaphore more than once. Although it is possible to do this, it may result in a serious deadlock.

opensem(S)

See also

creatsem(S), sigsem(S), waitsem(S)

Standards conformance

opensem is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

paccess

used in conjunction with ptrace for tracing a child process

Syntax

```
cc ... -lc

#include <sys/paccess.h>

int paccess (pid, cmd, offset, count, ptr)
int pid, cmd, offset, count;
char *ptr;
```

Description

paccess(S) provides an extended interface for accessing the address space, register save areas and local descriptor table of a child process that is being traced using **ptrace(CP)**. **paccess** also provides information about the u-area layout so that applications can determine this at runtime. This allows the developer to avoid hard coding kernel dependencies into the application.

The primary use of **paccess** is in the implementation of debuggers such as **adb(CP)** and **sdb(CP)**.

Parameters

Except for **P_RUOFFS**, all **paccess** commands transfer information between the calling process and a child process designated by *pid*. The parent and child processes must coordinate tracing using the facilities described by **ptrace**. Each command selects a particular class of data such as data address space or floating point register save area, and is used by the parent process to read or write the child process's context.

offset is a byte granularity logical offset from the base of the particular data area (determined by **paccess** relative to *cmd*).

count is the size in bytes of the requested transfer. *count* may be sized down by **paccess** if *offset* plus *count* would exceed the size of the particular data area. *count* may not exceed **MAXPCDATA**.

ptr designates a buffer in the caller's address space.

Note

paccess will transfer a maximum of *count* bytes between a buffer designated by *ptr* and a point *offset* bytes into a region of the child process's context designated by *cmd*. **P_RUOFFS** is an exception as indicated below.

Commands

The first four commands are used to read or write data in the data, text or stack regions of a process:

P_RDUSER	Reads a child process's D address space. <i>offset</i> is a 32 byte virtual address in the child's address space.
P_RIUSER	Reads a child process's I address space. <i>offset</i> is a 32 byte virtual address in the child's address space.
P_WDUSER	Writes to a child process's D address space. <i>offset</i> is a 32 byte virtual address in the child's address space.
P_WIUSER	Writes to a child process's I address space. <i>offset</i> is a 32 byte virtual address in the child's address space.
P_RUREGS	Reads the child process's register save area in the uarea. <i>offset</i> is a logical offset from the base of the register save area in the uarea. This may be used to read a particular register or a subset of the entire save area. Note that the ordering of registers in the save area is highly machine dependent. The offsets of the registers are defined in <code><sys/reg.h></code> .
P_WUREGS	Writes to the child process's register save area in the uarea. <i>offset</i> is a logical offset from the base of the register save area in the uarea. This may be used to read a particular register or a subset of the entire save area. Note that the ordering of registers in the save area is highly machine dependent. The offsets of the registers are defined in <code><sys/reg.h></code> .

The kernel stack pointer [KESP] will remain unchanged after any instance of P_WUREGS. Certain flags in the flags register [EFL] will remain unchanged after any instance of P_WUREGS. The flags are:

- PS_T
- PS_IE
- PS_IOPL
- PS_NT
- PS_RF
- PS_VM

See the file `<sys/tss.h>` for a definition of these flags.

P_RUFREGS	Reads the child process floating point register save area in the uarea. <i>offset</i> is a logical offset from the base of the register save area in the uarea. This may be used to read a particular register or a subset of the entire save area.

P_WUFREGS Writes to the child process floating point register save area in the uarea. *offset* is a logical offset from the base of the register save area in the uarea. This may be used to read a particular register or a subset of the entire save area.

For the above two commands paccess selects the appropriate save area relative to whether the process is using an Intel 87 series or Weitek co-processor chip and whether or not the hardware is present or is being emulated in software. Note that data is not currently transformed in any way and is highly chip and/or emulator dependent.

The ordering of registers in the save area is highly machine dependent. The offsets of the registers are defined in *<sys/reg.h>*.

P_RULDT Read the child processor's local descriptor table. This command has no write capability. *offset* is a logical byte offset from the base of the local descriptor table. paccess may have to be invoked several times to read the entire local descriptor table.

The following commands provide read and write access to the 80386 debug register save area.

P_RUDREGS Reads the 386 debug register save area. *offset* is a logical offset from the base of the register save area in the uarea. This command may be used to read a particular register or a subset of the entire save area.

P_WUDREGS Writes to the 386 debug register save area. *offset* is a logical offset from the base of the register save area in the uarea. This command may be used to read a particular register or a subset of the entire save area.

The 80386 debug register set may be used to implement text and data breakpoints. The layout and format of the debug register save area is highly chip dependent. The offsets of the registers are defined in *<sys/reg.h>*.

Certain fields, such as global bits, in the status register remains unaffected by any instance of P_WUDREGS. See */usr/include/sys/debugreg.h* for the definition of any flags.

The file */usr/include/sys/paccess.h* includes a structure template for reading and writing the debug register save area:

```
struct debugregs
```

This may be used in conjunction with */usr/include/sys/debugreg.h* for convenient handling of bitwise operations.

P_RUOFFS This command is used to obtain a list of kernel dependent uarea offsets typically used by debuggers such as *adb* and *sdb*. This allows an application developer to avoid using hard coded values and thereby gain greater independence from specific kernel versions.

pid and *offset* are ignored by this command.

The command returns the first *count* bytes of the *offsets* structure defined in *paccess.h*. That structure is defined as follows:

```
typedef long uoff;

/* 3.2 uarea offsets */

struct uoffsets {
    uoff u_info;           /* version */
    uoff u_uaddr;          /* kernel virtual address of uarea */
    uoff u_ar0;            /* user register save area pointer */
    uoff u_fps;            /* floating point save area */
    uoff u_fpemul;         /* separate emulator save area */
    uoff u_fpvalid;        /* if floating point save is valid */
    uoff u_weitek;         /* per proc weitek flag */
    uoff u_weitek_reg;     /* weitek save area */
    uoff u_debugreg;       /* debug register save area */
    uoff u_ldt;            /* offset of ldt */
    uoff u_ldtlimit;       /* size of ldt */
    uoff u_tss;            /* 3.2 adb */
    uoff u_sztss;          /* 3.2 adb */
    uoff u_sigreturn;      /* user signal return */
    uoff u_signal;
    uoff u_sdata;          /* u_exdata.ux_datorg */
    uoff u_dsize;          /* data size */
    uoff u_ssize;          /* stack size */
    uoff u_tsize;          /* text size */
    uoff u_sub;            /* stack upper bound */
    uoff uvstack;          /* virtual address of stack top */
    uoff uvtext;           /* virtual address of text */
};
```

Errors

paccess fails if one or more of the following is true:

[EFAULT]	<i>ptr</i> points outside the allocated address space.
[EINVAL]	1. <i>cmd</i> is invalid. 2. <i>offset</i> is less than 0 or beyond the size of the relevant structure.
[EIO]	An error was encountered when attempting to access data in the child's address space.
[ESRCH]	<i>pid</i> identifies a child that does not exist or has not executed a ptrace with request 0.

Return value

Upon successful completion, paccess returns the number of bytes successfully transferred to or from the child process. Otherwise, a value of -1 is returned and errno is set to indicate the error.

Files

/usr/include/sys/paccess.h

See also

access(S), ptrace(S)

Standards conformance

paccess is an extension of AT&T System V provided by the Santa Cruz Operation.

panel

PANEL library routines

Syntax

```
cc ... -lpanel -lcurses

#include <curses.h>
#include <panel.h>

int bottom_panel(panel)
PANEL *panel;

int del_panel(panel)
PANEL *panel;

int hide_panel(panel)
PANEL *panel;

int move_panel(panel, starty, startx)
PANEL *panel;
int starty, startx;

PANEL *new_panel(win)
WINDOW *win;

PANEL *panel_above(panel)
PANEL *panel;

PANEL *panel_below(panel)
PANEL *panel;

int panel_hidden(panel)
PANEL *panel;

WINDOW *panel_window(panel)
PANEL *panel;

char *panel_userptr(panel)
PANEL *panel;

int replace_panel(panel, window)
PANEL *panel;
WINDOW *window;
```

```

int set_panel_userptr(panel, ptr)
PANEL *panel;
char *ptr;

int show_panel(panel)
PANEL *panel;

int top_panel(panel)
PANEL *panel;

void update_panels()

```

Description

Panels are rectangles of text with depth. They enable your windows to overlap without having hidden portions of underlying windows be mistakenly visible. *stdscr* lies beneath all panels. The set of currently visible panels is the *deck* of panels.

A window is associated with every panel. The panel routines enable you to create panels, fetch their associated windows, shuffle panels in the deck, and manipulate panels in other ways.

Panel routines run on any terminal supported by the *curses(S)*, library. Once you compile your program including the PANEL header file *panel.h*, link your program with the panel and curses library routines.

Functions

bottom_panel (panel) puts *panel* at the bottom of all panels. It leaves the size and contents of its associated window, and its relations to other panels, wholly intact.

del_panel (panel) deletes the panel, but not its associated window.

hide_panel (panel) removes the panel from the panel deck and thus hides it from view. The panel's internal data structure, however, is retained.

move_panel (panel, starty, startx) moves the given *panel* window so that its upper-left corner is at *starty*, *startx*. Be sure to use this function, not *mvwin*, to move a panel window.

new_panel (win) returns a pointer to a new panel associated with *win*. The new panel is placed on top of the panel deck.

panel_above (panel) returns a pointer to the panel just above *panel*. If the panel argument is null, that is, (*panel* *) 0, it returns a pointer to the bottom panel in the deck.

panel_below (*panel*) returns a pointer to the panel just below *panel*. If the *panel* argument is null, it returns a pointer to the top panel in the deck.

panel_hidden (*panel*) returns a boolean value indicating whether or not the given *panel* has been removed from the panel deck.

panel_userptr (*panel*) returns the user pointer for a given *panel*.

panel_window (*panel*) returns a pointer to the window of *panel*.

replace_panel (*panel*, *window*) replaces the current window of *panel* with *window*.

set_panel_userptr (*panel*, *ptr*) sets the panel's user pointer.

show_panel (*panel*) makes a hidden panel visible by placing it on top of the panels in the panel deck.

top_panel (*panel*) puts the given visible *panel* on top of all panels in the deck.

update_panels () refreshes the virtual screen to reflect the relations between the panels in the deck, but does not call **doupdate** to refresh the physical screen.

Diagnostics

Each panel routine that returns a pointer to an object returns null if an error occurs. Each panel routine that returns an int value returns OK if it executes successfully and ERR if not.

See also

curses(S), field(S), fieldtype(S), form(S), item(S), menu(S), tam(S)

Standards conformance

panel is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

passlen

determine minimum password length of an account

Syntax

```
cc ... -lprot -lm

#include <sys/types.h>
#include <sys/security.h>
#include <sys/audit.h>
#include <prot.h>

int passlen (life_dur, login_delay, alphabet_size)
time_t life_dur;
time_t login_delay;
int alphabet_size;
```

Description

The **passlen** routine returns the minimum password length according to the algorithm in the *DoD Password Management Guideline (Green Book)* given the password lifetime duration for an account in seconds, the argument *life_dur*; the login delay between login attempts in seconds, the argument *login_delay*; and the *alphabet_size* of distinct characters. Two of the parameters used by the *Guideline* are fixed, namely the size of the alphabet at 26 characters (the argument *alphabet_size* should always reflect this) and the probability of guessing a password at 1 chance in a million.

The formula for *min_pass_len* is:

$$\text{ceil} \left[\frac{\ln \frac{\text{login\_delay} \times \text{life\_dur}}{\text{prob\_guess}}}{\ln \text{alphabet\_size}} \right]$$

Return value

The **passlen** routine returns the result of the calculation rounded up to the nearest integer so as to err on the side of greater security. **passlen** is always successful and therefore no errors are defined.

Note

If the lifetime duration, the login delay time, or the guess parameter changes in the Protected Password database for this account (or for the system default if those values are used in an account), it is a good idea to invalidate the password in case the new parameters require a longer minimum password length.

See also

exp(S), floor(S), passwd(C), randomword(S)
DoD Password Management Guideline (Green Book),
CSC-STD-002-85, 12 April 1985.

Standards conformance

The **passlen** routine is an extension of AT&T System V provided by the Santa Cruz Operation.

pathconf

get configurable pathname variables

Syntax

```
cc ... -lc

#include <unistd.h>

long pathconf (path, name)
char *path;
int name;

long fpathconf (fildes, name)
int fildes, name;
```

Description

pathconf - determines current value of the pathname of a file or directory

fpathconf - determines current value of an open file descriptor

The **pathconf** and **fpathconf** functions provide a method for an application to determine the current value of a configurable limit or option (variable) associated with a file or directory.

For **pathconf**, the *path* argument points to the pathname of a file or directory. For **fpathconf**, the *fildes* the argument is an open file descriptor.

The *name* argument represents the variable to be queried relative to the file or directory. The implementation supports all of the variables listed in the following table and may support others. The variables in the following table come from *<limits.h>* or *<unistd.h>* and the symbolic constants, defined in *<unistd.h>*, that are the corresponding values used for *name*.

Variable	name Value	Notes
{LINK_MAX}	{_PC_LINK_MAX}	1
{MAX_CANON}	{_PC_MAX_CANON}	2
{MAX_INPUT}	{_PC_MAX_INPUT}	2
{NAME_MAX}	{_PC_NAME_MAX}	3,4
{PATH_MAX}	{_PC_PATH_MAX}	4,5
{PIPE_BUF}	{_PC_PIPE_BUF}	6
{_POSIX_CHOWN_RESTRICTED}	{_PC_CHOWN_RESTRICTED}	7
{_POSIX_NO_TRUNC}	{_PC_NO_TRUNC}	3,4
{_POSIX_VDISABLE}	{_PC_VDISABLE}	2

Note that:

1. If *path* or *fildes* refers to a directory, the value returned applies to the directory itself.
2. The behavior is undefined if *path* or *fildes* does not refer to a terminal file.
3. If *path* or *fildes* refers to a directory, the value returned applies to the filenames within the directory.
4. The behavior is undefined if *path* or *fildes* does not refer to a directory.
5. If *path* or *fildes* refers to a directory, the value returned is the maximum length of a relative pathname when the specified directory is the working directory.
6. If *path* refers to FIFO, or *fildes* refers to a pipe or FIFO, the value returned applies to the referenced object itself. If *path* or *fildes* refers to a directory, the value returned applies to any FIFOs that exist or can be created within the directory. If *path* or *fildes* refer to any other type of file, the behavior is undefined.
7. If *path* or *fildes* refer to a directory, the value returned applies to any files defined in this standard, other than directories, that exist or can be created within this directory.

Return value

If *name* is an invalid value, the *pathconf* and *fpathconf* functions return -1.

If the variable corresponding to *name* has no limit for the path or file descriptor, the *pathconf* and *fpathconf* functions return a -1 without changing *errno*.

If the implementation needs to use *path* to determine the value of *name* and the implementation does not support the association of *name* with the file specified by *path*, or if the process did not have the appropriate privileges to query the file specified by *path*, or *path* does not exist, the *pathconf* function returns -1.

If the implementation needs to use *fildes* to determine the value of *name* and the implementation does not support the association of *name* with the file specified by *fildes*, or if *fildes* is an invalid descriptor, the *fpathcon* function returns -1.

Otherwise, the *pathconf* and *fpathconf* functions return the current variable value for the file or directory without changing *errno*. The value returned is not more restrictive than the corresponding value described to the application when it was compiled with the implementation's *<limits.h>* or *<unistd.h>*.

Diagnostics

If any of the following conditions occur, the **pathconf** and **fpathcon** functions return -1 and set **errno** to the corresponding value:

[EINVAL] The value of *name* is invalid.

For each of the following conditions, if the condition is detected, the **pathconf** function returns a -1 and set **errno** to the following value:

[EACCES] Search permission is denied for a component of the path prefix.

[EINVAL] The implementation does not support an association of the variable name with the specified file.

[ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX} while {POSIX_NO_TRUNC} is in effect.

[ENOENT] The named file does not exist or the *path* argument points to an empty string.

[ENOTDIR] A component of the path prefix is not a directory.

For each of the following conditions, if the condition is detected, the **fpathconf** function returns a -1 and set **errno** to the corresponding value:

[EBADF] The *fildevs* argument is not a valid file descriptor.

[EINVAL] The implementation does not support an association of the variable name with the specified file.

Standards conformance

pathconf is conformant with:

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

pause

suspend process until signal

Syntax

```
cc ... -lc  
  
int pause( )
```

Description

The **pause** system call suspends the calling process until it receives a signal. The signal must be one that is not currently set to be ignored by the calling process.

If the signal causes termination of the calling process, **pause** does not return.

If the signal is caught by the calling process, and control is returned from the signal-catching function (see **signal(S)**), the calling process resumes execution from the point of suspension; with a return value of -1 from **pause** and **errno** set to **EINTR**.

See also

alarm(S), **kill(S)**, **signal(S)**, **sigpause(S)**, **wait(S)**

Standards conformance

pause is conformant with:
AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

perror

system error messages

Syntax

```
cc ... -lc

void perror (s)
const char *s;

extern int errno;

extern char *sys_errlist[];

extern int sys_nerr;
```

Description

The **perror** function produces a message on the standard error output, describing the last error encountered during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, then the message and a new-line. (However, if *s*="", the colon is not printed.) To be of most use, the argument string should include the name of the program that incurred the error. The error number is taken from the external variable **errno**, which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the array of message strings **sys_errlist** is provided; **errno** can be used as an index into this table to get the message string without the new-line. **sys_nerr** is the number of messages in the table; it should be checked because new error codes may be added to the system before they are added to the table.

See also

Intro(S)

Standards conformance

errno is conformant with:

AT&T SVID Issue 2;
 X/Open Portability Guide, Issue 3, 1989;
 Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);
 IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
 and NIST FIPS 151-1.

perror is conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
ANSI X3.159-1989 Programming Language — C;
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

sys_errlist and **sys_nerr** are conformant with:

AT&T SVID Issue 2; however, **sys_errlist** and **sys_nerr** have been withdrawn from XPG3.

phs

Note the MMDF transmission phase (phs_)

Syntax

```
cc ... -lmmdf
```

```
#include <util.h>
#include <mmdf.h>
#include <ch.h>
#include <phs.h>
```

```
time_t
phs_get (thechan, phase)
Chan *thechan;
int phase;
```

```
phs_msg (thechan, naddrs, len)
Chan *thechan;
int naddrs;
long len;
```

```
phs_note (thechan, phase)
Chan *thechan;
int phase;
```

```
phases:
    PHS_CNGOT      /* Got connection to site */
    PHS_CNEND      /* End connection to site */
    PHS_CNSTRT     /* Start connecting to site */
    PHS_REEND      /* End reading (sending) mail from site */
    PHS_RESTRT     /* Start reading (sending) mail from site */
    PHS_WREND      /* End writing (picking up) mail to site */
    PHS_WRSTRT     /* Start writing (picking up) mail to site */
```

Description

phs_get - gets time-stamp of MMDF specified channel and phase

phs_msg - records the transmission of one MMDF message

phs_note - records the indicated MMDF event

These routines are used to record the progress made during a connection between MMDF and another mail system. Sessions are divided into the previously listed phases. External time-stamps are maintained for each channel and for each phase. The time-stamps are primarily used by the **checkque(ADM)** program, but notations also are made in the channel log (see **logs(F)**). The time-stamps are maintained by setting the modified time of files

in channel subdirectories of the "phsdfdir" directory specified in the MMDf configuration. One file is kept per time-stamp (see **FILES** in the next section).

The **phs_note** routine records the indicated event. The **phs_msg** routine is a special case; it records the transmission of one message, noting the number of addresses for it and the number of characters. This is recorded only in the channel log and is not given any external time-stamps. The **phs_get** routine returns the time-stamp of the specified channel and phase. If a time-stamp does not exist for the specified channel and phase, **phs_get** returns zero.

Files

<phsdfdir>/<channel>/cend	PSH_CNEND time-stamp file
<phsdfdir>/<channel>/cgot	PSH_CNGOT time-stamp file
<phsdfdir>/<channel>/cstrt	PSH_CNSTRT time-stamp file
<phsdfdir>/<channel>/rend	PSH_REEND time-stamp file
<phsdfdir>/<channel>/rmsg	PSH_REMSG time-stamp file
<phsdfdir>/<channel>/rstrt	PSH_RESTRT time-stamp file
<phsdfdir>/<channel>/wend	PSH_WREND time-stamp file
<phsdfdir>/<channel>/wmsg	PSH_WRMSG time-stamp file
<phsdfdir>/<channel>/wstrt	PSH_WRSTRT time-stamp file

See also

llog(S), ml_send(S), mmdf(S), tai(S)

Standards conformance

The **phs_get**, **phs_msg**, and **phs_note** routines are not part of any currently supported standard; they were developed at the University of Delaware and are used by permission.

pipe

create an interprocess channel

Syntax

```
cc ... -lc

int pipe (fildes)
int fildes[2];
```

Description

The **pipe** system call creates an I/O mechanism called a pipe and returns two file descriptors, *fildes*[0] and *fildes*[1]. *fildes*[0] is opened for reading and *fildes*[1] is opened for writing.

Up to 5120 bytes of data are buffered by the pipe before the writing process is blocked. A read-only file descriptor *fildes*[0] accesses the data written to *fildes*[1] on a first-in-first-out (FIFO) basis.

The **pipe** system call fails if:

[EMFILE]	The current process has reached the limit of open files.
[ENFILE]	The system file table is full.
[ENODEV]	There is no pipe filesystem.

Diagnostics

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

See also

read(S), sh(C), write(S), fcntl(S), sysi86(S), pipe(ADM)

Standards conformance

pipe is conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

plock

lock process, text, or data in memory

Syntax

```
cc ... -lc

#include <sys/lock.h>

int plock (op)
int op;
```

Description

The **plock** system call allows the calling process to lock its text segment (text lock), its data segment (data lock), or both its text and data segments (process lock) into memory. Locked segments are immune to all routine swapping. **plock** also allows these segments to be unlocked. The effective user ID of the calling process must be super user to use this call. *op* specifies the following:

PROCLOCK- lock text and data segments into memory (process lock)

TXTLOCK- lock text segment into memory (text lock)

DATLOCK- lock data segment into memory (data lock)

UNLOCK- remove locks

The **plock** system call fails and does not perform the requested operation if one or more of the following is true:

[EAGAIN] Not enough memory.

- [EINVAL]**
1. *op* is equal to **PROCLOCK** and a process lock, a text lock, or a data lock already exists on the calling process.
 2. *op* is equal to **TXTLOCK** and a text lock or a process lock already exists on the calling process.
 3. *op* is equal to **DATLOCK** and a data lock or a process lock already exists on the calling process.
 4. *op* is equal to **UNLOCK** and no type of lock exists on the calling process.

[EPERM] The effective user ID of the calling process is not super user.

Diagnostics

Upon successful completion, a value of 0 is returned to the calling process. Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

See also

exec(S), exit(S), fork(S)

Standards conformance

plock is conformant with:

AT&T SVID Issue 2, but has been withdrawn from XPG3.

(S)

plot

graphics interface subroutines

Syntax

```
cc ... -lplot

int arc(x, y, x0, y0, x1, y1)
int x, y, x0, y0, x1, y1;

int box(x0, y0, x1, y1)
int x0, y0, x1, y1;

int circle(x, y, r)
int x, y, r;

int closepl()

int cont(x, y)
int x, y;

int erase()

int label(s)
char *s;

int line(x1, y1, x2, y2)
int x1, y1, x2, y2;

int linemod(s)
char *s;

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

int openpl()

int point(x, y)
int x, y;

int space(x0, y0, x1, y1)
int x0, y0, x1, y1;
```

Description

These subroutines generate graphic output in a relatively device-independent manner. **space** must be used before any of these functions to declare the amount of space necessary (see *plot(FP)*). **openpl** must be used before any of the others to open the device for writing. **closepl** flushes the output.

box draws a box using the four coordinates specified as arguments.

circle draws a circle of radius r with center at the point (x, y) .

arc draws an arc of a circle with center at the point (x, y) between the points $(x0, y0)$ and $(x1, y1)$.

String arguments to **label** and **linemod** are terminated by nulls and do not contain new-lines.

See **plot(FP)** for a description of the effect of the remaining functions.

The library files listed below provide several flavors of these routines.

Return value

Even though all routines have integer return values, no meaningful value is returned.

Files

<code>/usr/lib/libplot.a</code>	produces output for tplot(ADM) filters
<code>/usr/lib/lib300.a</code>	for DASI 300
<code>/usr/lib/lib300s.a</code>	for DASI 300s
<code>/usr/lib/lib450.a</code>	for DASI 450
<code>/usr/lib/lib4014.a</code>	for TEKTRONIX 4014

Warnings

To compile a program containing these functions in *file.c*, it is necessary to use:

```
cc file.c -lplot
```

In order to execute it, it is necessary to use:

```
a.out | tplot
```

Since the **plot** routines use the *stdio.h* header file, programs that do not use standard I/O are increased in size more than might be expected.

See also

graph(ADM), **plot(FP)**, **stat(FP)**, **tplot(ADM)**

Standards conformance

The **plot** package is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

poll

STREAMS input/output multiplexing

Syntax

```
cc ... -lc

#include <stropts.h>
#include <poll.h>

int poll(fds, nfd, timeout)
struct pollfd fds[];
unsigned long nfd;
int timeout;
```

Description

The **poll** system call provides users with a mechanism for multiplexing input/output over a set of file descriptors that reference open STREAMS or any driver that supports the **select(S)** system call. The **poll** system call identifies those STREAMS on which a user can send or receive messages, or on which certain events have occurred. A user can receive messages using **read(S)** or **getmsg(S)** and can send messages using **write(S)** and **putmsg(S)**. Certain **ioctl(S)** calls, such as **I_RECVFD** and **I_SENDFD**, can also be used to receive and send messages.

fds specifies the file descriptors to be examined and the events of interest for each file descriptor. It is a pointer to an array with one element for each open file descriptor of interest. The array's elements are **pollfd** structures which contain the following members:

```
int fd;           /* file descriptor */
short events;     /* requested events */
short revents;    /* returned events */
```

where **fd** specifies an open file descriptor and **events** and **revents** are bit-masks constructed by or-ing any combination of the following event flags:

- POLLIN** A non-priority or file descriptor passing message (see **I_RECVFD**) is present on the stream head read queue. This flag is set even if the message is of zero length. In *revents*, this flag is mutually exclusive with **POLLPRI**.
- POLLPRI** A priority message is present on the stream head read queue. This flag is set even if the message is of zero length. In *revents*, this flag is mutually exclusive with **POLLIN**.
- POLLOUT** The first downstream write queue in the stream is not full. Priority control messages can be sent (see **putmsg**) at any time.

- POLLERR** An error message has arrived at the stream head. This flag is only valid in the *revents* bitmask; it is not used in the *events* field.
- POLLHUP** A hangup has occurred on the stream. This event and **POLLNVAL** are mutually exclusive; a stream can never be writable if a hangup has occurred. However, this event and **POLLIN** or **POLLPRI** are not mutually exclusive. This flag is only valid in the *revents* bitmask; it is not used in the *events* field.
- POLLNVAL** The specified *fd* value does not belong to an open stream. This flag is only valid in the *revents* field; it is not used in the *events* field.

For each element of the array pointed to by *fds*, **poll** examines the given file descriptor for the event(s) specified in *events*. The number of file descriptors to be examined is specified by *nfds*. If *nfds* exceeds **NOFILES**, the system limit of open files (see **ulimit(S)**), **poll** fails.

If the value *fd* is less than zero, *events* is ignored and *revents* is set to 0 in that entry on return from **poll**.

The results of the **poll** query are stored in the *revents* field in the **pollfd** structure. Bits are set in the *revents* bitmask to indicate which of the requested events are true. If none are true, none of the specified bits is set in *revents* when the **poll** call returns. The event flags **POLLHUP**, **POLLERR**, and **POLLNVAL** are always set in *revents* if the conditions they indicate are true; this occurs even though these flags were not present in *events*.

If none of the defined events have occurred on any selected file descriptor, **poll** waits at least *timeout* msec for an event to occur on any of the selected file descriptors. On a computer where millisecond timing accuracy is not available, *timeout* is rounded up to the nearest legal value available on that system. If the value *timeout* is 0, **poll** returns immediately. If the value of *timeout* is -1, **poll** blocks until a requested event occurs or until the call is interrupted. The **poll** system call is not affected by the **O_NONBLOCK** flag.

The **poll** system call fails if one or more of the following is true:

- [EAGAIN]** Allocation of internal data structures failed but request should be attempted again.
- [EFAULT]** Some argument points outside the allocated address space.
- [EINTR]** A signal was caught during the **poll** system call.
- [EINVAL]** The argument *nfds* is less than zero, or *nfds* is greater than **NOFILES**.

Diagnostics

Upon successful completion, a non-negative value is returned. A positive value indicates the total number of file descriptors that has been selected (that is, file descriptors for which the *revents* field is non-zero). A value of 0 indicates that the call timed out and no file descriptors have been selected. Upon failure, a value of -1 is returned, and *errno* is set to indicate the error.

Note

The *poll* system call also works on devices that support the *select* system call, such as the event driver and the console driver.

See also

getmsg(S), *Intro(S)*, *putmsg(S)*, *read(S)*, *write(S)*, *select(S)*,

Standards conformance

poll is conformant with:
AT&T SVID Issue 2.

popen, pclose

initiate a pipe to or from a process

Syntax

```
cc ... -lc

#include <stdio.h>

FILE *popen (command, type)
const char *command;
char *type;

int pclose (stream)
FILE *stream;
```

Description

popen - creates pipe between calling program and command

pclose - returns exit status of command

The **popen** function creates a pipe between the calling program and the command to be executed. The arguments to **popen** are pointers to null-terminated strings. *command* consists of a shell command line. *type* is an I/O mode, either *r* for reading or *w* for writing. The value returned is a stream pointer such that one can write to the standard input of the command, if the I/O mode is *w*, by writing to the file stream; and one can read from the standard output of the command, if the I/O mode is *r*, by reading from the file stream.

A stream opened by **popen** should be closed by **pclose**, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type *r* command may be used as an input filter and a type *w* as an output filter.

Example

A typical call may be:

```
char *cmd = 'ls *.c';
FILE *ptr;
if ((ptr = popen(cmd, "r")) != NULL)
    while (fgets(buf, n, ptr) != NULL)
        (void) printf("%s ", buf);
```


This prints in **stdout** (see **stdio(S)**) all the file names in the current directory that have a ".c" suffix.

Diagnostics

The **popen** function returns a NULL pointer if files or processes cannot be created.

The **pclose** function returns -1 if stream is not associated with a command opened with the **popen** routine.

Warnings

If the original and those processes opened with **popen** concurrently read or write a common file, neither should use buffered I/O. Problems with an output filter may be forestalled by careful buffer flushing, for example, with **fflush(S)**.

On UNIX systems, **pclose** uses **waitpid(S)** to wait for the child process executing the command to terminate. On XENIX systems, **pclose** uses **wait(S)**. If the function evaluated on receipt of a **SIGCLD** signal is set to **SIG_IGN**, then the behavior of **pclose** depends on which type of system the process is running on. On UNIX systems, **pclose** will return a status of -1, and set **errno** to **ECHILD**. On XENIX systems, **pclose** will block the process until all child processes terminate, and then return a status of -1, and set **errno** to **ECHILD**. However, a deadlock occurs because the child process that executed the command will not terminate until its parent receives its **SIGCLD** signal.

See also

fclose(S), **fflush(S)**, **fopen(S)**, **pipe(S)**, **wait(S)**, **stdio(S)**, **system(S)**

Standards conformance

pclose and **popen** are conformant with:

AT&T SVID Issue 2;
and X/Open Portability Guide, Issue 3, 1989.

printf, fprintf, sprintf

print formatted output

Syntax

```
cc ... -lc

#include <stdio.h>

int printf(format, arg . . .
const char *format;

int fprintf(stream, format, arg . . .
FILE *stream;
const char *format;

int sprintf(s, format [, arg ] . . .
const char *s;
char *format;
```

Description

The `printf` function places output on the standard output stream `stdout`. `fprintf` places output on the named output *stream*. `sprintf` places "output," followed by the null character (`\0`), in consecutive bytes starting at **s*; it is the user's responsibility to ensure that enough storage is available. Each function returns the number of characters transmitted (not including the `\0` in the case of `sprintf`), or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its *args* under control of the *format*. The *format* is a character string that contains three types of objects: plain characters, which are simply copied to the output stream; escape sequences that represent non-graphic characters; and conversion specifications, each of which results in fetching of zero or more *args*. The results are undefined if there are insufficient *args* for the format. If the format is exhausted while *args* remain, the excess *args* are simply ignored.

Each conversion specification is introduced by the character `"%"`. After the `"%"`, the following appear in sequence:

- Zero or more *flags*, which modify the meaning of the conversion specification.
- An optional, decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it is padded on the left (or right, if the left-adjustment flag `'-'`, described below, has been given) to the field width. The padding is with blanks unless the field width digit string starts with a zero, in which case the padding is with zeros.

- A precision that gives the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x`, or `X` conversions, the number of digits to appear after the decimal point for the `e`, `E`, and `f` conversions, the maximum number of significant digits for the `g` and `G` conversion, or the maximum number of characters to be printed from a string in `s` conversion. The precision takes the form of a period (".") followed by a decimal digit string; a null digit string is treated as zero. Padding specified by the precision overrides the padding specified by the field width.
- An optional `l` (ell) specifying that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion character applies to a long integer `arg`. An `l` before any other conversion character is ignored.
- A character that indicates the type of conversion to be applied.

A field width or precision or both may be indicated by an asterisk (" * ") instead of a digit string. In this case, an integer *arg* supplies the field width or precision. The *arg* that is actually converted is not fetched until the conversion letter is seen, so the *args* specifying field width or precision must appear *before* the *arg* (if any) to be converted. A negative field width argument is taken as a '-' flag followed by a positive field width. If the precision argument is negative, it is changed to zero.

The flag characters and their meanings are:

- The result of the conversion is left-justified within the field.
- + The result of a signed conversion always begins with a sign (+ or -).
- (Space) If the first character of a signed conversion is not a sign, a blank is prefixed to the result. This implies that if the blank and + flags both appear, the blank flag is ignored.
- # This flag specifies that the value is to be converted to an "alternate form." For `c`, `d`, `i`, `s`, and `u` conversions, the flag has no effect. For `o` conversion, it increases the precision to force the first digit of the result to be a zero. For `x` or `X` conversion, a non-zero result has `0x` or `0X` prefixed to it. For `e`, `E`, `f`, `g`, and `G` conversions, the result always contains a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For `g` and `G` conversions, trailing zeroes are *not* removed from the result (which they normally are).

The conversion characters and their meanings are:

- d, i, o, u, x, X** The integer *arg* is converted to signed decimal (d or i) , unsigned octal, (o) , decimal (u) , or hexadecimal notation (x or X), respectively; the letters abcdef are used for x conversion and the letters ABCDEF for X conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeroes. The default precision is 1. The result of converting a zero value with a precision of zero is a null string.
- f** The float or double *arg* is converted to decimal notation in the style "[*-*]ddd.ddd," where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, six digits are output; if the precision is explicitly 0, no decimal point appears.
- e, E** The float or double *arg* is converted in the style "[*-*]d.ddd~~e~~dd," where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, six digits are produced; if the precision is zero, no decimal point appears. The E format code produces a number with E instead of e introducing the exponent. The exponent always contains at least two digits.
- g, G** The float or double *arg* is printed in style f or e (or in style E in the case of a G format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style e is used only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.
- c** The character *arg* is printed.
- s** The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (\0) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed. A NULL value for *arg* yields undefined results.
- %** Print a %; no argument is converted.

In printing floating point types (float and double), if the exponent is 0x7FF and the mantissa is not equal to zero, then the output is

`[ - ]NaN0xdddddddd`

where *0xdddddddd* is the hexadecimal representation of the leftmost 32 bits of the mantissa. If the mantissa is zero, the output is

`[±]inf`

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by `printf` and `fprintf` are printed as if `putc(S)` had been called.

Examples

To print a date and time in the form "Sunday, July 3, 10:02," where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %i, %d:%.2d", weekday, month, day, hour, min);
```

To print π to 5 decimal places:

```
printf("pi = %.5f", 4 * atan(1.0));
```

See also

`ecvt(S)`, `putc(S)`, `scanf(S)`, `stdio(S)`

Standards conformance

`fprintf`, `printf` and `sprintf` are conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
ANSI X3.159-1989 Programming Language — C;
Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

proct1

controls active processes or process groups

Syntax

```
cc ... -lx

#include <sys/proct1.h>

int proct1(pid, command, arg)
int pid, command;
char *arg;
```

Description

proct1 performs a variety of functions on active processes or process groups. It has the same form as the **ioctl(S)** system call, except that a process ID (*pid*) is substituted for a file descriptor as the first parameter.

command is an integer mnemonic, specifying the action to be taken, and *arg* is a pointer to a data structure which defines the parameters associated with the *command* if necessary.

If *pid* is greater than zero (0), the *command* affects the process whose process ID is equal to *pid*. *pid* may be 1.

If *pid* is zero, the command is sent to all processes, except processes 0 and 1 whose process group ID is equal to the process group ID of the sender.

If *pid* is -1 and the effective user ID of the sender is not the super user, the command is sent to all processes, except processes 0 and 1 whose real user ID is equal to the effective user ID of the sender.

If *pid* is -1 and the effective user ID of the sender is super user, the command is sent to all processes except processes 0 and 1.

If *pid* is negative but not -1, a signal is sent to all processes whose process group ID is equal to the absolute value of *pid*.

proct1 fails if one or more of the following are true:

[EFAULT]	<i>arg</i> is not a valid address.
[EINVAL]	<i>command</i> or <i>arg</i> is not valid.
[ENOMEM]	The program has requested more memory than is available.

- [EPERM] The user ID of the sending process is not super user, and its real or effective user ID does not match the real or effective user ID of the receiving process.
- [ESRCH] No process can be found to match the specified *pid*.

Memory restrictions

exec(S) may fail when the required physical memory is larger than the available swap space. This restriction may be lifted using one of the following *proct1* commands:

PRHUGEX

Allows programs to be executed by this process even if they exceed the available swap space. Such programs must still fit in the available physical memory and the caller's effective user ID must be super user. Such HUGE processes are locked in memory to prevent them from being swapped. Processes that are marked HUGE with this system call but are not greater than the size of the swapper behave normally but can expand into a HUGE, locked process.

PRNORMEX

Makes a process unable to *exec(S)* HUGE programs. This call may be executed by any user. If an attempt is made to classify a process as normal using the PRNORMEX call when the process is already too big to swap, the *proct1* call fails, returning EINVAL.

For example, you can use the following code to allow a process to be executed even if it exceeds the available memory swapping space:

```
if (argc < 2) {
    fputs ("usage: runbig command arg ...\n", stderr);
    exit(2);
}
argv[argc] = 0;
if (proct1(getpid(), PRHUGEX, (char *) 0) < 0) {
    perror ("runbig");
    exit(1);
}
```

Return value

If an error has occurred, a value of -1 is returned and **errno** is set to indicate the error.

See also

exec(S), ioctl(S), kill(S)

Standards conformance

proctl is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

(S)

profil

execution time profile

Syntax

```
cc ... -lc

void profil (buff, bufsiz, offset, scale)
void (* offset)();
char *buff;
unsigned bufsiz, scale;
```

Description

buff points to an area of core whose length (in bytes) is given by *bufsiz*. After this call, the user's program counter (pc) is examined each clock tick. Then the value of *offset* is subtracted from it, and the remainder multiplied by *scale*. If the resulting number corresponds to an entry inside *buff*, that entry is incremented. An entry is defined as a series of bytes with length *sizeof(short)*.

The scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: 0177777 (octal) gives a 1-1 mapping of pc's to entries in *buff*; 077777 (octal) maps each pair of instruction entries together. 02(octal) maps all instructions onto the beginning of *buff* (producing a non-interrupting core clock).

Profiling is turned off by giving a *scale* of 0 or 1. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is turned off when an exec is executed, but remains on in child and parent both after a fork. Profiling is turned off if an update in *buff* would cause a memory fault.

See also

prof(CP), monitor(S), times(S)

Standards conformance

profil is conformant with:
AT&T SVID Issue 2, but has been withdrawn from XPG3.

ptrace

process trace

Syntax

```
cc ...-lc
```

```
int ptrace (request, pid, addr, data);
int request, pid, data;
```

Description

The **ptrace** system call provides a means by which a parent process may control the execution of a child process. Its primary use is for the implementation of breakpoint debugging (see **sdb**(CP)). The child process behaves normally until it encounters a signal (see **signal**(S) for the list), at which time it enters a stopped state and its parent is notified via **wait**(S). When the child is in the stopped state, its parent can examine and modify its "core image" using **ptrace**. Also, the parent can cause the child either to terminate or continue, with the possibility of ignoring the signal that caused it to stop. The data type of the argument *addr* depends upon the particular request given to **ptrace**.

The *request* argument determines the precise action to be taken by **ptrace** and is one of the following:

- 0 This request must be issued by the child process if it is to be traced by its parent. It turns on the child's trace flag that stipulates that the child should be left in a stopped state upon receipt of a signal rather than the state specified by *func* (see **signal**(S)). The *pid*, *addr*, and *data* arguments are ignored, and a return value is not defined for this request. Peculiar results ensue if the parent does not expect to trace the child.

The remainder of the requests can only be used by the parent process. For each, *pid* is the process ID of the child. The child must be in a stopped state before these requests are made.

- 1, 2 With these requests, the word at location *addr* in the address space of the child is returned to the parent process. If I and D space are separated, request 1 returns a word from I space, and request 2 returns a word from D space. If I and D space are not separated, either request 1 or request 2 may be used with equal results. The *data* argument is ignored.

- 3 With this request, the word at location *addr* in the child's **USER** area in the system's address space (see *<sys/user.h>*) is returned to the parent process. The *data* argument is ignored. This request fails if *addr* is outside the **USER** area, in which case a value of -1 is returned to the parent process and the parent's **errno** is set to **EIO**.
- 4, 5 With these requests, the value given by the *data* argument is written into the address space of the child at location *addr*. If I and D space are separated, request 4 writes a word into I space, and request 5 writes a word into D space. If I and D space are not separated, either request 4 or request 5 may be used with equal results. Upon successful completion, the value written into the address space of the child is returned to the parent. These two requests fail if *addr* is a location in a pure procedure space and another process is executing in that space. Upon failure a value of -1 is returned to the parent process and the parent's **errno** is set to **EIO**.
- 6 With this request, a few entries in the child's **USER** area can be written. *Data* gives the value that is to be written and *addr* is the location of the entry. The few entries that can be written are all registers.

On the 80386, the **ptrace** system call can be used to modify the debug registers.

The 80386 debug registers are used to specify an address to monitor in a user process. Any access to this location by the user process delivers a **SIGTRAP** (see **signal(S)**) to the user process and possibly restart the parent process.

The 80386 debug registers can be accessed by using the 3 or 6 options of the **ptrace** system call to read or write a traced-process's u-area. The file *<sys/debugreg.h>* should be included in the parent process that wants to control the debug registers. This header file defines bit masks that describe the debug-registers in the `u_debugreg[]` array in the u-area.

The debug registers numbered `u.u_debugreg [DR_FIRSTADDR] (%dr0)` to `u.u_debugreg[DR_LASTADDR] (%dr3)` contain process addresses which are monitored according to the instructions provided in `u.u_debugreg[DR_CONTROL] (%dr7)`. Only the **DR_LOCAL_ENABLE_MASK** and the various read/write and length bits in `u.u_debugreg[DR_CONTROL]` can be set. Setting **DR_LOCAL_SLOWDOWN** to slow down processing is also highly recommended. The setting of all other bits is undefined and should be set to zero to ensure compatibility with future Intel processors.

In the process being debugged, these registers are automatically loaded before entering user-mode (privilege level 3) and cleared before entering the system for any reason. In System V Release 3.2, if the location specified by a debug-register is accessed during a system call, core-dump, or interrupt service, no trap ensues.

- 7 This request causes the child to resume execution. If the **data** argument is 0, all pending signals including the one that caused the child to stop are canceled before it resumes execution. If the **data** argument is a valid signal number, the child resumes execution as if it had incurred that signal, and any other pending signals are canceled. The **addr** argument must be equal to 1 for this request. Upon successful completion, the value of **data** is returned to the parent. This request fails if **data** is not 0 or a valid signal number, in which case a value of -1 is returned to the parent process and the parent's **errno** is set to **EIO**.
- 8 This request causes the child to terminate with the same consequences as **exit(S)**.
- 9 This request sets the trace bit in the Processor Status Word of the child and then executes the same steps as listed above for request 7. The trace bit causes an interrupt upon completion of one machine instruction. This effectively allows single stepping of the child.
- 10 Attach to the process **PID**, and begin tracing it. Process **PID** must have the same uid as the current process, and it must not have executed a **suid** or **sgid** program, nor may it be executing a file for which the current process' UID did not have read permission at the time of the **exec**. Process **PID** need not be a child of the current process. Root may attach to any process.
- 11 Detach the process being traced. Process **PID** is no longer traced, and continues executing. The same steps as listed above for request 7 are performed.
- 12 The **STRCFRK** flag for the process **PID** is set. This causes any child processes resulting from a **fork(S)** by the process **PID** to be controlled by the current process. The child processes stops before executing the return from the **fork** call. On a multiprocessor it is possible for another process to successfully **ATTACH** before the child is set up to be traced by the parent's controller. In this case, the other process remains the child's controller, and the **STRCFRK** bit is ignored.
- 13 The process with process ID equal to **DATA** is given control of the process **PID**. The process with a process ID equal to **DATA** must have the same UID as the current process. The current process no longer controls the process **PID**. The process with process ID equal to **DATA** is sent a **SIGCLD** signal.

To forestall possible fraud, **ptrace** inhibits the set-user-ID facility on subsequent **exec(S)** calls. If a traced process calls **exec**, it stops before executing the first instruction of the new image showing signal **SIGTRAP**.

General errors

The **ptrace** system call generally fails if the child process is running under **i286emul(CP)** or one or more of the following is true:

[EIO]	request is an illegal number.
[EINVAL]	For request 10, the process is already being traced. For request 0, the parent of the current process has a process ID of 1, or the process is already traced, or, on a multiprocessor, the parent process is exiting.
[EPERM]	For request 10, the process has a different UID, the process's GID is not in the group list of the current process, the process is executing a setuid/setgid program, or the process is executing a file for which it does not have read permission. For request 13, the process data has a different UID than the current process.
[ESRCH]	pid identifies a process that does not exist or is not controlled by the current process, or is in the process of terminating (on a multiprocessor).

See also

exec(S), **sdb(CP)**, **signal(S)**, **wait(S)**

Standards conformance

ptrace is conformant with:

AT&T SVID Issue 2, but has been withdrawn from XPG3.

putc, putchar, fputc, putw

put character or word on a stream

Syntax

```
cc ...-lc

#include <stdio.h>

int putc (c, stream)
int c;
FILE *stream;

int putchar (c)
int c;

int fputc (c, stream)
int c;
FILE *stream;

int putw (w, stream)
int w;
FILE *stream;
```

Description

putc - writes character to output stream
putchar - writes character to output stream
fputc - behaves as putc but is a function
putw - writes integer to output stream

The **putc** macro writes the character *c* onto the output stream (at the position where the file pointer, if defined, is pointing). **putchar(c)** is defined as **putc(c, stdout)**. **putc** and **putchar** are macros.

fputc behaves like **putc**, but is a function rather than a macro. **fputc** runs more slowly than **putc**, but it takes less space per invocation and its name can be passed as an argument to a function.

putw writes the word (that is, integer) *w* to the output stream (at the position at which the file pointer, if defined, is pointing). The size of a word is the size of an integer and varies from machine to machine. **putw** neither assumes nor causes special alignment in the file.

Diagnostics

On success, these functions (with the exception of **putw**) each return the value they have written. (**putw** returns **ferror(stream)**.) On failure, they return the constant EOF. This occurs if the file stream is not open for writing or if the output file cannot grow. Because EOF is a valid integer, **ferror(S)** should be used to detect **putw** errors.

Notes

Because it is implemented as a macro, **putc** evaluates a stream argument more than once. In particular, **putc(c,*f++)** doesn't work sensibly. **fputc** should be used instead.

Because of possible differences in word length and byte ordering, files written using **putw** are machine-dependent, and may not be read using **getw** on a different processor.

See also

fclose(S), ferror(S), fopen(S), fread(S), printf(S), puts(S), setbuf(S), stdio(S)

Standards conformance

fputc, putc and **putchar** are conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
ANSI X3.159-1989 Programming Language — C;
Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1); and NIST FIPS 151-1.

putw is conformant with:

AT&T SVID Issue 2;
X/Open Portability Guide, Issue 3, 1989;
Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

putenv

change or add value to environment

Syntax

```
cc ... -lc

int putenv (string)
char *string;
```

Description

string points to a string of the form *name = value*. The **putenv** function makes the value of the environment variable *name* equal to *value* by altering an existing variable or creating a new one. In either case, the string pointed to by *string* becomes part of the environment, so altering the string changes the environment. The space used by *string* is no longer used once a new string defining *name* is passed to **putenv**.

Diagnostics

The **putenv** function returns non-zero if it was unable to obtain enough space via **malloc** for an expanded environment, otherwise zero.

Warnings

The **putenv** function manipulates the environment pointed to by *environ*, and can be used in conjunction with **getenv**. However, *envp* (the third argument to **main**) is not changed. After **putenv** is called, environmental variables are not in alphabetical order.

A potential error is to call **putenv** with an automatic variable as the argument, then exit the calling function while *string* is still part of the environment.

This routine uses **malloc(S)** to enlarge the environment.

See also

environ(M), **exec(S)**, **getenv(S)**, **malloc(S)**

Standards conformance

putenv is conformant with:

AT&T SVID Issue 2;
and X/Open Portability Guide, Issue 3, 1989.

putmsg

send a message on a stream

Syntax

```
cc ... -lc

#include <stropts.h>

int putmsg (fd, ctlptr, dataptr, flags)
int fd;
struct strbuf *ctlptr;
struct strbuf *dataptr;
int flags;
```

Description

The **putmsg** system call creates a message from user-specified buffer(s) and sends the message to a STREAMS file. The message may contain either a data part, a control part or both. The data and control parts to be sent are distinguished by placement in separate buffers, as described below. The semantics of each part is defined by the STREAMS module that receives the message.

fd specifies a file descriptor referencing an open stream. *ctlptr* and *dataptr* each point to a **strbuf** structure which contains the following members:

```
int maxlen;    /* not used */
int len;       /* length of data */
char *buf;     /* ptr to buffer */
```

ctlptr points to the structure describing the control part, if any, to be included in the message. The *buf* field in the **strbuf** structure points to the buffer where the control information resides, and the *len* field indicates the number of bytes to be sent. The *maxlen* field is not used in **putmsg** (see **getmsg(S)**). In a similar manner, *dataptr* specifies the data, if any, to be included in the message. *flags* may be set to the values 0 or **RS_HIPRI** and is used as described below.

To send the data part of a message, *dataptr* must be non-NULL and the *len* field of *dataptr* must have a value of 0 or greater. To send the control part of a message, the corresponding values must be set for *ctlptr*. No data (control) part is sent if either *dataptr* (*ctlptr*) is NULL or the *len* field of *dataptr* (*ctlptr*) is set to -1.

If a control part is specified, and *flags* is set to **RS_HIPRI**, a *priority* message is sent. If *flags* is set to 0, a non-priority message is sent. If no control part is specified, and *flags* is set to **RS_HIPRI**, **putmsg** fails and sets **errno** to **EINVAL**. If no control part and no data part are specified, and *flags* is set to 0, no message is sent, and 0 is returned.

For non-priority messages, **putmsg** blocks if the stream write queue is full due to internal flow control conditions. For priority messages, **putmsg** does not block on this condition. For non-priority messages, **putmsg** does not block when the write queue is full and **O_NONBLOCK** is set. Instead, it fails and sets **errno** to **EAGAIN**.

The **putmsg** system call also blocks, unless prevented by lack of internal resources, waiting for the availability of message blocks in the stream, regardless of priority or whether **O_NONBLOCK** has been specified. No partial message is sent.

The **putmsg** system call fails if one or more of the following is true:

[EAGAIN]	1. A non-priority message was specified, the O_NONBLOCK flag is set, and the stream write queue is full due to internal flow control conditions. 2. Buffers could not be allocated for the message that was to be created.
[EBADF]	<i>fd</i> is not a valid file descriptor open for writing.
[EFAULT]	<i>ctlptr</i> or <i>dataptr</i> points outside the allocated address space.
[EINTR]	A signal was caught during the putmsg system call.
[EINVAL]	1. An undefined value was specified in <i>flags</i>, or <i>flags</i> is set to RS_HIPRI and no control part was supplied. 2. The stream referenced by <i>fd</i> is linked below a multiplexer.
[ENOSTR]	A stream is not associated with <i>fd</i> .
[ENXIO]	A hangup condition was generated downstream for the specified stream.
[ERANGE]	The size of the data part of the message does not fall within the range specified by the maximum and minimum packet sizes of the topmost stream module. This value is also returned if the control part of the message is larger than the maximum configured size of the control part of a message, or if the data part of a message is larger than the maximum configured size of the data part of a message.

A **putmsg** also fails if a STREAMS error message had been processed by the stream head before the call to **putmsg**. The error returned is the value contained in the STREAMS error message.

Diagnostics

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

putmsg(S)

See also

getmsg(S), poll(S), read(S), write(S)

Standards conformance

putmsg is conformant with:
AT&T SVID Issue 2.

putpwent

write password file entry

Syntax

```
cc ... -lc

#include <pwd.h>

int putpwent (p, f)
struct passwd *p;
FILE *f;
```

Description

The **putpwent** routine is the inverse of the **getpwent(S)** routine. Given a pointer to a **passwd** structure created by **getpwent** (or two other routines that appear on the same man page - **getpwuid(S)** or **getpwnam(S)**), **putpwent** writes a line which matches the format of */etc/passwd* onto the stream *f*.

Return value

Upon successful completion, the **putpwent** routine returns a value of zero. If an error was detected during its operation, it will return a non-zero value.

See also

getpwent(S), **getpwuid(S)**, **getpwnam(S)**

Standards conformance

putpwent conforms with:

AT&T SVID Issue 2, but has been withdrawn from XPG3.

puts, fputs

put a string on a stream

Syntax

```
cc ... -lc

#include <stdio.h>

int puts (s)
char *s;

int fputs (s, stream)
const char *s;
FILE *stream;
```

Description

puts - writes null-terminated string to standard output stream

fputs - writes nul-terminated string to named output stream

The **puts** function writes the null-terminated string pointed to by *s*, followed by a new-line character, to the standard output stream **stdout**.

fputs writes the null-terminated string pointed to by *s* to the named output stream.

Neither function writes the terminating null character.

Diagnostics

Both routines return EOF on error, which occurs if the routines try to write to a file that has not been opened for writing.

Notes

The **puts** function appends a new-line character while **fputs** does not.

See also

ferror(S), fopen(S), fread(S), printf(S), putc(S), stdio(S)

Standards conformance

fputs and **puts** are conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

ANSI X3.159-1989 Programming Language — C;

Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

putspent

write shadow password file entry

Syntax

```
#include <shadow.h>

int putspent (p, fp)
struct spwd *p;
FILE *fp;
```

Description

The **putspent** routine is the inverse of **getspent(S)**. Given a pointer to a **spwd** structure created by the **getspent** routine (or the **getspnam** routine), the **putspent** routine writes a line on the stream **fp**, which matches the format of */etc/shadow*.

If the “**sp_min**”, “**sp_max**” or “**sp_lstchg**” field of the **spwd** structure is -1, the corresponding */etc/shadow* field is cleared.

Diagnostics

The **putspent** routine returns non-zero if an error was detected during its operation, otherwise zero.

Notes

If a program not otherwise using standard I/O uses this routine, the size of the program increases more than might be expected.

This routine is provided for compatibility purposes only, and may not function correctly on all systems. We recommend that the **putprpwnam(S)** routine be used as an alternative.

See also

getspent(S), **getpwent(S)**, **putpwent(S)**, **getprpwent(S)**

Standards conformance

The **putspent** routine is conformant with AT&T SVID Issue 3.

pw_nametoid, pw_idtoname, gr_nametoid, gr_idtoname

map between user and group names and IDs

Syntax

```
cc ... -lprot

int pw_nametoid(name)
char *name;

char *pw_idtoname(id)
int id;

int gr_nametoid(name)
char *name;

char *gr_idtoname(id)
int id;
```

Description.

pw_nametoid - map between user names and IDs

pw_idtoname - map between user IDs and names

gr_nametoid - map between group names and IDs

gr_idtoname - map between group IDs and names

These routines provide an efficient mapping between user and group names and identifiers (IDs). They maintain a separate binary database, which is automatically updated each time the routines encounter a changed */etc/passwd* or */etc/group* file. Also, these routines do not interfere with the behavior of the **getpwent(S)** and **getgrent(S)** routines, and are thus used by the Protected Database routines, which must frequently convert between names and identifiers but may not disrupt an application's use of password mapping routines.

The **pw_nametoid** expects the character argument *name*, a user name, and returns its associated user ID. The **pw_idtoname** expects the argument *id*, a user ID number, and returns its associated user name.

The **gr_nametoid** expects the character argument *name*, a group name, and returns its associated group ID. The **gr_idtoname** expects the argument *id*, a group ID number, and returns its associated group name.

These routines are much more efficient than the `getpwent(S)` and `getgrent(S)` routines, in that they do not parse the underlying files and keep the database in memory after the first use. Thus, savings are much greater for the second and subsequent lookups. The binary databases are stored in the */etc/auth/system* directory, which is maintained by the Auth protected subsystem.

Return value

Upon successful completion, the `pw_idtoname` and the `gr_idtoname` routines return a pointer to a string in an internal (to the routines) memory area containing the user or the group names. These routines return a NULL on failure.

Upon successful completion, the `pw_nametoid` and the `gr_nametoid` routines return a non-negative user or group ID. On failure, these routines return a value of -1.

Files

/etc/auth/system/pw_id_map
/etc/auth/system/gr_id_map
/etc/passwd
/etc/group

See also

`getpwent(S)`, `getgrent(S)`, `getprpwent(S)`

Standards conformance

All of these routines are an extension of AT&T System V provided by the Santa Cruz Operation.

qsort

quicker sort

Syntax

```
cc ... -lc

#include <stdlib.h>

void qsort (base, nel, width, compar)
void *base;
size_t nel, width;
int (*compar)();
```

Description

The **qsort** function is an implementation of the quicker-sort algorithm. It sorts a table of data in place.

The *base* argument points to the element at the base of the table. The *nel* argument is the number of elements in the table. The *width* argument is the size in bytes of each element. The *compar* argument is the name of the comparison function, which is called with two arguments that point to the elements being compared. The comparison function must return an integer less than, equal to, or greater than zero, according to whether the first argument is to be considered as less than, equal to, or greater than the second argument.

Notes

The pointer to the base of the table should be of type pointer-to-element, and cast to type pointer-to-void.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

The order in the output of two items which compare as equal is unpredictable.

See also

bsearch(S), **lsearch(S)**, **sort(C)**, **string(S)**

Standards conformance

qsort is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

ANSI X3.159-1989 Programming Language — C;

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

raise

send signal to the execution program

Syntax

```
cc ... -lc
#include <signal.h>

int raise (sig);
int sig;
```

Description

The raise function sends the signal *sig* to the execution program.

Return value

The raise function returns a zero if successful, nonzero if unsuccessful.

See also

sigaction(S), signal(S), sigset(S)

Standards conformance

raise is conformant with:
ANSI X3.159-1989 Programming Language — C.

(S)

rand, srand

simple random-number generator

Syntax

```
cc ... -lc

#include <stdlib.h>

int rand ()

void srand (seed)
unsigned seed;
```

Description

rand - simple random-number generator

srand - reset random-number generator

The **rand** function uses a multiplicative congruential random-number generator with period 2^{32} that returns successive pseudo-random numbers in the range from 0 to $2^{15}-1$.

The **srand** function can be called at any time to reset the random-number generator to a random starting point. The generator is initially seeded with a value of 1.

Notes

The spectral properties of **rand** are limited. The **drand48(S)** function provides a much better, though more elaborate, random-number generator.

The following functions define the semantics of the functions **rand** and **srand**.

```
static unsigned long int next = 1;
int rand()
{
    next = next * 1103515245 + 12345;
    return ((unsigned int)(next/65536) % 32768);
}
void srand(seed)
unsigned int seed;
{
    next = seed;
}
```

Specifying the semantics makes it possible to reproduce the behavior of programs that use pseudo-random sequences. This facilitates the testing of portable applications in different implementations.

See also

drand48(S)

Standards conformance

rand and **srand** are conformant with:

AT&T SVID Issue 2;
 X/Open Portability Guide, Issue 3, 1989;
 ANSI X3.159-1989 Programming Language — C;
 IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
 and NIST FIPS 151-1.

randomword

generate a pronounceable password

Syntax

```
cc ... -lprot

#include <sys/types.h>
#include <sys/security.h>
#include <sys/audit.h>
#include <prot.h>

int randomword (word, hyphenated_word, min, max, restrict, seed)
char *word;
char *hyphenated_word;
unsigned short int min;
unsigned short int max;
int restrict;
long seed;
```

Description

randomword places a null-terminated random pronounceable password into the user-supplied space *word* and returns the length of the generated word. The hyphenated version of the word is placed in *hyphenated_word*.

The *min* and *max* arguments denote the minimum and maximum lengths allowed for *word*. *min* may equal *max*, but may not be greater than *max*, and neither may be negative. The user space pre-allocated is at least *max* for *word* and $2*max-1$ for *hyphenated_word*. Of course, the smaller *min* and *max* are, the less the selection of random words. For a program like `login(M)`, suggested values are 6 for *min* and 8 for *max*.

The argument *restrict* is 0 when no restrictions are placed on the generated word. It is non-zero when the words generated pass the tests of the routine *acceptable_password* described by `accept_pw(S)`.

The *seed* argument specifies an initial seed for the random number generator used by **randomword**. It is used by the routine only on the first time it is called; the argument is ignored on subsequent calls.

Notes

The password generator relies on a random number generator that produces uniformly distributed integers. Because the password generator invokes the random number generator many times even for one word, the random number generator has to be very good. The period (distinct numbers produced given a particular seed) and number space (range of possible numbers) must both be large.

Files

/etc/passwd
/etc/group

See also

accept_pw(S), drand48(S), login(M), passwd(C), seed(S)

Standards conformance

randomword is an extension of AT&T System V provided by the Santa Cruz Operation.

rdchk

checks to see if there is data to be read

Syntax

```
cc ... -lx

int rdchk(fdes)
int fdes;
```

Description

rdchk checks to see if a process will block if it attempts to read the file designated by *fdes*. **rdchk** returns 1 if the process will not block when a read is attempted or if it is the end of the file (EOF). In this context, the proper sequence of calls using **rdchk** is:

```
if(rdchk(fildes) > 0)

    read(fildes, buffer, nbytes);
```

Diagnostics

rdchk returns -1 if an error occurs (for example, **EBADF**), 0 when the process will block if it issues a **read** and 1 if it is okay to read. **EBADF** is returned if a **rdchk** is used on a semaphore file or if the file specified does not exist.

See also

read(S)

Standards conformance

rdchk is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

read

read from file

Syntax

```
cc ... -lc

int read (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

Description

fildes is a file descriptor obtained from a `creat(S)`, `open(S)`, `dup(S)`, `Lfcntl(S)`, or `pipe(S)` system call.

The `read` system call attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*.

On devices capable of seeking, the `read` starts at a position in the file given by the file pointer associated with *fildes*. Upon return from `read`, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a file is undefined.

Upon successful completion, `read` returns the number of bytes actually read and placed in the buffer; this number may be less than *nbyte* if the file is associated with a communication line (see `ioctl(S)` and `termio(M)`), or if the number of bytes left in the file is less than *nbyte* bytes. A value of 0 is returned when an end-of-file has been reached.

A `read` from a STREAMS file can operate in three different modes: "byte-stream" mode, "message-nondiscard" mode, and "message-discard" mode. The default is byte-stream mode. This can be changed using the `I_SRDOPT` `ioctl` request, (see `streamio(STR)`) and can be tested with the `I_GRDOPT` `ioctl`. In byte-stream mode, `read` retrieves data from the *stream* until it has retrieved *nbyte* bytes, or until no more data can be retrieved. Byte-stream mode ignores message boundaries.

In STREAMS message-nondiscard mode, *read* retrieves data until it has read *nbyte* bytes, or until it reaches a message boundary. If the read does not retrieve all the data in a message, the remaining data are replaced on the *stream*, and can be retrieved by the next *read* or *getmsg(S)* call. Message-discard mode also retrieves data until it has retrieved *nbyte* bytes, or it reaches a message boundary. However, unread data remaining in a message after the read returns are discarded and are not available for a subsequent read or *getmsg*.

When attempting to read from a regular file with mandatory file/record locking set (see *chmod(S)*), and there is a blocking (that is, owned by another process) write lock on the segment of the file to be read:

- If *O_NONBLOCK* is set, the read returns a -1 and set *errno* to *EAGAIN*.
- If *O_NONBLOCK* is clear, the read sleeps until the blocking record lock is removed.

When attempting to read from an empty pipe (or FIFO):

- If *O_NONBLOCK* is set, the read returns a 0.
- If *O_NONBLOCK* is clear, the read blocks until data is written to the file or the file is no longer open for writing.

When attempting to read a file associated with a tty that has no data currently available:

- If *O_NONBLOCK* is set, the read returns a 0.
- If *O_NONBLOCK* is clear, the read blocks until data becomes available.

When attempting to read a file associated with a stream that has no data currently available:

- If *O_NONBLOCK* is set, the read returns a -1 and set *errno* to *EAGAIN*.
- If *O_NONBLOCK* is clear, the read blocks until data becomes available.

When reading from a STREAMS file, handling of zero-byte messages is determined by the current read mode setting. In byte-stream mode, *read* accepts data until it has read *nbyte* bytes, or until there is no more data to read, or until a zero-byte message block is encountered. The *read* system call then returns the number of bytes read, and places the zero-byte message back on the *stream* to be retrieved by the next *read* or *getmsg*. In the two other modes, a zero-byte message returns a value of 0 and the message is removed from the *stream*. When a zero-byte message is read as the first message on a *stream*, a value of 0 is returned regardless of the read mode.

A read from a STREAMS file can only process data messages. It cannot process any type of protocol message and fails if a protocol message is encountered at the *stream head*.

The `read` system call fails if one or more of the following are true:

[EAGAIN]	Mandatory file/record locking was set, <code>O_NONBLOCK</code> was set, and there was a blocking record lock.
[EAGAIN]	Total amount of system memory available when reading via raw IO is temporarily insufficient.
[EAGAIN]	No message waiting to be read on a <i>stream</i> and <code>O_NONBLOCK</code> flag set.
[EBADF]	<i>fdes</i> is not a valid file descriptor open for reading.
[EBADMSG]	Message waiting to be read on a <i>stream</i> is not a data message.
[EDEADLK]	The read was going to go to sleep and cause a deadlock situation to occur.
[EFAULT]	<i>buf</i> points outside the allocated address space.
[EINTR]	A signal was caught during the <code>read</code> system call.
[EINVAL]	Attempted to read from a <i>stream</i> linked to a multiplexer.
[EIO]	A physical I/O error has occurred.
[ENOLCK]	The system record lock table was full, so the read could not go to sleep until the blocking record lock was removed.
[ENOLINK]	<i>fdes</i> is on a remote machine and the link to that machine is no longer active.
[ENXIO]	Mandatory file/record locking was set, <code>O_NONBLOCK</code> was set, and there was a blocking record lock.

Diagnostics

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. Otherwise, a -1 is returned, and `errno` is set to indicate the error.

See also

`creat(S)`, `dup(S)`, `fcntl(S)`, `getmsg(S)`, `ioctl(S)`, `open(S)`, `pipe(S)`

Standards conformance

`read` is conformant with:

AT&T SVID Issue 2;
 X/Open Portability Guide, Issue 3, 1989;
 Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);
 IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
 and NIST FIPS 151-1.

readlink

reads a symbolic link

Syntax

```
cc ... -lc

int readlink (path, buf, bufsiz)
char *path, *buf;
int bufsiz;
```

Description

The **readlink** function reads a symbolic link previously created by a call to **symlink(S)**.

readlink places the contents of the symbolic link referred to by *path* in the *buf* buffer, which has the size *bufsiz*. The contents of the link are not null terminated when returned.

This routine fails and the buffer is unchanged if one or more of the following is true:

[EACCES]	Search permission is denied for a component of the <i>oath</i> prefix.
[EFAULT]	<i>path</i> or <i>buf</i> extends outside the process's allocated address space.
[EINVAL]	The named file is not a symbolic link.
[EIO]	An I/O error occurred while reading from or writing to the file-system.
[ELOOP]	Too many symbolic links were encountered in translating <i>path</i> .
[ENAMETOOLONG]	The length of a component of <i>path</i> exceeds 255 characters or the length of <i>path</i> exceeds 1023 characters.
[ENOENT]	The named file does not exist.

Return value

The count of characters placed in the buffer is returned if the call succeeds, or -1 is returned if an error occurs, and the error value is placed in **errno**.

See also

stat(S), symlink(S)

Standards conformance

readlink is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

(S)

regcmp, regex

compiles and executes regular expressions

Syntax

```
cc ... -lintl

char *regcmp (string1 [, string2, . . . (char *)0)
char *string1, *string2, . . .

char *regex (re, subject, [ ret0, . . .
char *re, *subject, *ret0, . . .

extern char *__loc1;
```

Description

The **regcmp** routine compiles a regular expression (consisting of the concatenated arguments) and returns a pointer to the compiled form. The **malloc(S)** routine creates space for the compiled form. It is the user's responsibility to free unneeded space so allocated. A **NULL** return from **regcmp** indicates an incorrect argument. **regcmp(CP)** has been written to generally preclude the need for this routine at execution time.

The **regex** routine executes a compiled pattern against the subject string. Additional arguments are passed to receive values back. **regex** returns **NULL** on failure or a pointer to the next unmatched character on success. A global character pointer **__loc1** points to where the match began.

The **regex** and **regcmp** routines were borrowed from the editor, **ed(C)**; however, the syntax and semantics have been changed slightly.

The following are the symbols understood by **regex** and **regcmp**, and their meanings.

- [] * . ^ These symbols retain their meaning in **ed(C)**.
- \$ Matches the end of the string; **\n** matches a new-line.
- Within brackets the minus means through. For example, [a-z] is equivalent to [abcd...xyz]. The "-" can appear as itself only if used as the first or last character. For example, the character class expression []-] matches the characters] and -.
- + A regular expression followed by "+" means **one or more times**. For example, [0-9]⁺ is equivalent to [0-9] [0-9]*.

{m} {m,} {m,u}

Integer values enclosed in "{}" indicate the number of times the preceding regular expression is to be applied. The value *m* is the minimum number and *u* is a number, less than 256, which is the maximum. If only *m* is present (for example, {m}), it indicates the exact number of times the regular expression is to be applied. The value {m,} is analogous to {m,infinity}. The plus ("+") and star ("*") operations are equivalent to {1,} and {0,} respectively.

(...)\$*n* The value of the enclosed regular expression is to be returned. The value is stored in the (*n*+1)th argument following the subject argument. At most ten enclosed regular expressions are allowed. **regex** makes its assignments unconditionally.

(...) Parentheses are used for grouping. An operator, for example, "*", "+", "{}", can work on a single character or a regular expression enclosed in parentheses. For example, (a*(cb+)*)\$0.

By necessity, all the above defined symbols are special. They must, therefore, be escaped with a \ (backslash) to be used as themselves.

Notes

The user program may run out of memory if **regcmp** is called iteratively without freeing the vectors that are no longer required.

The **regcmp** and **regex** routines are in the international library **libintl**, and thus will not be accessible if the cc(CP) flag **-nointl** is used when compiling.

See also

ed(C), **free(S)**, **malloc(S)**

Standards conformance

regcmp, **regex** and **__loc1** are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

Examples

Example 1:

```
char *cursor, *newcursor, *ptr;
...
newcursor = regex((ptr = regcmp("\n", (char *)0)), cursor);
free(ptr);
```

This example matches a leading new-line in the subject string pointed at by **cursor**.

Example 2:

```
char ret0[9];
char *newcursor, *name;
...
name = regcmp("([A-Za-z][A-Za-z0-9]{0,7})$0", (char *)0);
newcursor = regex(name, "012Testing345", ret0);
```

This example matches through the string "Testing3" and returns the address of the character after the last matched character (the "4"). The string "Testing3" is copied to the character array *ret0*.

Example 3:

```
#include "file.i"
char *string, *newcursor;
...
newcursor = regex(name, string);
```

This example applies a precompiled regular expression in *file.i* (see *regcmp(CP)*) against *string*.

Example 4:

```
char *ptr, *newcursor;

ptr = regcmp("[a-[-i=][:digit:]]*", (char*)0);
newcursor = regex(ptr, "123CHIC0321");
```

It is assumed in this example that the current locale's collation rules specify the following sequence:

A, a, B, b, C, c, CH, Ch, ch, D, d, E, e, F, f, G, g, H, h, I, i, ...

The characters *I* and *i* are also both in the same "primary" collation group.

The following characters are all members of the *digit ctype* class:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

This example matches through the string "123CHIC" and returns the address of the character "O" in the string.

regex

regular expression compile and match routines

Syntax

```
#define INIT <declarations>
#define GETC ( ) <getc code>
#define PEEKC ( ) <peekc code>
#define UNGETC (c) <ungetc code>
#define RETURN (pointer) <return code>
#define ERROR (val) <error code>

#include <regex.h>

int advance (string, expbuf)
char *string, *expbuf;

char *compile (instring, expbuf, endbuf, eof)
char *instring, *expbuf, *endbuf;
int eof;

int step (string, expbuf)
char *string, *expbuf;

extern char *loc1, *loc2, *locs;

extern int circf, sed, nbra;
```

Description

This page describes general-purpose, regular expression matching routines in the form of `ed(C)`, defined in `<regex.h>`. Programs such as `ed(C)`, `sed(C)`, `grep(C)`, `bs(C)`, `expr(C)`, and so on, that perform regular expression matching use this source file. In this way, only this file need be changed to maintain regular expression compatibility.

The **advance** and **step** functions do pattern matching give an character string and a compiled regular expression as input.

The interface to this file is complex. Programs that include this file must have the following five macros declared before the `#include <regex.h>` statement. These macros are used by the **compile** routine.

GETC() Return the value of the next character in the regular expression pattern. Successive calls to **GETC()** should return successive characters of the regular expression.

- PEEKC()** Return the next character in the regular expression. Successive calls to **PEEKC()** should return the same character [which should also be the next character returned by **GETC()**].
- UNGETC(c)** Cause the argument *c* to be returned by the next call to **GETC()** [and **PEEKC()**]. No more than one character of push-back is ever needed, and this character is guaranteed to be the last character read by **GETC()**. The value of the macro **UNGETC(c)** is always ignored.
- RETURN(pointer)** This macro is used on normal exit of the **compile** routine. The value of the argument *pointer* is a pointer to the character after the last character of the compiled regular expression. This is useful to programs which have memory allocation to manage.
- ERROR(val)** This is the abnormal return from the **compile** routine. The argument *val* is an error number (see table below for meanings). This call should never return.

Error	Meaning
11	Range endpoint too large.
16	Bad number.
25	"\" \"digit\" out of range.
36	Illegal or missing delimiter.
41	No remembered search string.
42	"\" \" \" imbalance.
43	Too many "\" \".
44	More than 2 numbers given in "\" \" \" \".
45	"\" \" expected after "\" \".
46	First number exceeds second in "\" \" \" \".
49	"\" \" imbalance.
50	Regular expression overflow.

The syntax of the **compile** routine is as follows:

```
compile(instring, expbuf, endbuf, eof)
```

The first parameter *instring* is never used explicitly by the **compile** routine but is useful for programs that pass down different pointers to input characters. It is sometimes used in the **INIT** declaration (see below). Programs that call functions to input characters or have characters in an external array can pass down a value of `((char *) 0)` for this parameter.

The next parameter *expbuf* is a character pointer. It points to the place where the compiled regular expression is placed.

The parameter *endbuf* is one more than the highest address where the compiled regular expression may be placed. If the compiled expression cannot fit in *(endbuf-expbuf)* bytes, a call to **ERROR(50)** is made.

The parameter *eof* is the character which marks the end of the regular expression. For example, in `ed(C)`, this character is usually a `" / "`.

Each program that includes this file must have a `#define` statement for `INIT`. This definition is placed right after the declaration for the function `compile` and the opening curly brace (`" { "`). It is used for dependent declarations and initializations. Most often it is used to set a register variable to point to the beginning of the regular expression so that this register variable can be used in the declarations for `GETC()`, `PEEKC()` and `UNGETC()`. Otherwise it can be used to declare external variables that might be used by `GETC()`, `PEEKC()` and `UNGETC()`. See the example below of the declarations taken from `grep(C)`.

There are other functions in this file which perform actual regular expression matching, one of which is the function `step`. The call to `step` is as follows:

```
step(string, expbuf)
```

The first parameter to `step` is a pointer to a string of characters to be checked for a match. The second parameter, *expbuf*, is the compiled regular expression which was obtained by a call of the function `compile`.

The function `step` returns non-zero if the given string matches the regular expression, and zero if the expressions do not match. If there is a match, two external character pointers are set as a side effect to the call to `step`. The variable set in `step` is `loc1`. This is a pointer to the first character that matched the regular expression. The variable `loc2`, which is set by the function `advance`, points to the character after the last character that matches the regular expression. Thus if the regular expression matches the entire line, `loc1` points to the first character of *string* and `loc2` points to the null at the end of *string*.

`step` uses the external variable `circf` which is set by `compile` if the regular expression begins with `"^"`. If this is set, then `step` tries to match the regular expression to the beginning of the string only. If more than one regular expression is to be compiled before the first is executed, the value of `circf` should be saved for each compiled expression, and `circf` should be set to that saved value before each call to `step`.

The function `advance` is called from `step` with the same arguments as `step`. The purpose of `step` is to step through the *string* argument and call `advance` until `advance` returns non-zero indicating a match or until the end of *string* is reached. If one wants to constrain *string* to the beginning of the line in all cases, `step` need not be called; simply call `advance`.

When **advance** encounters a "*" or "\{~\}" sequence in the regular expression, it advances its pointer to the string to be matched as far as possible and recursively calls itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, **advance** backs up along the string until it finds a match or reaches the point in the string that initially matched the "*" or "\{~\}". It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external character pointer *locs* is equal to the point in the string at sometime during the backing up process, **advance** breaks out of the loop that backs up and returns zero. This is used by **ed(C)** and **sed(C)** for substitutions done globally (not just the first occurrence, but the whole line) so, for example, expressions like *s/y*/lg* do not loop forever.

The additional external variables **sed** and **nbra** are used for special purposes.

Examples

The following is an example of how the regular expression macros and calls look from **grep(C)**:

```
#define INIT      register char *sp = instring;
#define GETC()    (*sp++)
#define PEEKC()   (*sp)
#define UNGETC(c) (--sp)
#define RETURN(c) return;
#define ERROR(c)  regerr()

#include <regex.h>
...
    (void) compile(*argv, expbuf, &expbuf[ESIZE], '\0');
...
    if (step(linebuf, expbuf)
        succeed());
```

See also

ed(C), **expr(C)**, **grep(C)**, **regcmp(S)**, **sed(C)**

Standards conformance

advance, **compile**, **step**, **loc1**, **loc2** and **locs** are conformant with:

AT&T SVID Issue 2 ;
and X/Open Portability Guide, Issue 3, 1989

circf, **sed** and **nbra** are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

remove

removes filename

Syntax

```
cc ... -lc

#include <stdio.h>

int remove (filename)
const char *filename;
```

Description

The **remove** function causes the file whose name is the string pointed to by *filename* to be no longer accessible by that name. A subsequent attempt to open that file using that name fails, unless it is created anew. If the file is open, the behavior of the **remove** function is implementation-defined.

Return value

The **remove** function returns zero if the operation succeeds, nonzero if it fails.

Standards conformance

remove is conformant with:

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

rename

changes filename

Syntax

```
cc ... -lc

#include <stdio.h>

int rename (old, new)
const char *old;
const char *new;
```

Description

The **rename** function causes the file or directory whose pathname is the string pointed to by *old* to be henceforth known by the pathname given by the string pointed to by *new*. The file or directory named *old* is effectively removed. If a file or directory named by the string pointed to by *new* exists prior to the call to the **rename** function, *new* is removed.

If the *old* argument and the *new* argument both refer to links to the same existing file, the **rename** function returns successfully and performs no other action.

The *old* and the *new* arguments must both point to the pathname of a file or they must both point to the pathname of a directory. If the link named by the *new* argument exists, it is removed and *old* renamed to *new*. In this case, a link named *new* exists throughout the the renaming operation and refers either to the file referred to by *new* or *old* before the operation began. Write access permission is required for both the directory containing *old* and the directory containing *new*.

If the *old* argument points to the pathname of a directory, the *new* argument cannot point to the pathname of a file. If the directory named by the *new* argument exists, it is removed and *old* renamed to *new*. In this case, a link named *new* exists throughout the renaming operation and refers either to the file referred to by *new* or *old* before the operation began. Thus, if *new* names an existing directory, it is required to be an empty directory.

The *new* pathname cannot contain a path prefix that names *old*. Write access permission is required for the directory containing *old* and the directory containing *new*. If the *old* argument points to the pathname of a directory, write access permission may be required for the directory named by *old*, and, if it exists, the directory named by *new*.

If the link named by the *new* argument exists and the file's link count becomes zero when it is removed and no process has the file open, the space occupied by the file is freed and the file is no longer accessible. If one of more processes have the file open when the last link is removed, the link is removed before **rename** returns, but the removal of the file contents is postponed until all references to the file have been closed.

Upon successful completion, the **rename** function marks for update the `st_ctime` and `st_mtime` fields of the parent directory of each file.

Return value

The **rename** function returns zero if the operation succeeds. A return value of -1 indicates that an error has occurred and an error code has been stored in `errno`.

Diagnostics

If any of the following conditions occur, the **rename** function returns -1 and sets `errno` to the corresponding value:

- [EACCES]** A component of either path prefix denies search permission; or one of the directories containing *old* or *new* denies write permissions; or, write permission is required and is denied for a directory pointed to the the *old* or *new* arguments.
- [EBUSY]** The directory named by *old* or *new* cannot be renamed because it is being used by the system or another process and the implementation considers this to be an error.
- [EEXIST]** The link named by *new* is a directory containing entries other than dot and dot-dot.
- [ENOTEMPTY]** Same as **[EEXIST]**.
- [EINVAL]** The *new* directory pathname contains a path prefix that names the *old* directory.
- [EISDIR]** The *new* argument points to a directory and the *old* argument points to a file that is not a directory.
- [ENAMETOOLONG]** The length of the *old* or *new* argument exceeds `{PATH_MAX}` or a pathname component is longer than `{NAME_MAX}` while `{_POSIX_NO_TRUNC}` is in effect.
- [ENOENT]** The link named by the *old* argument does not exist or either *old* or *new* points to an empty string.
- [ENOSPC]** The directory that would contain *new* cannot be extended.
- [ENOTDIR]** A component of either path prefix is not a directory; or the *old* argument names a directory and the *new* argument names a nondirectory file.

rename(S)

[EROFS]	The requested operation requires writing in a directory on a read-only file system.
[EXDEV]	The links named by <i>new</i> and <i>old</i> are on different file systems and the implementation does not support links between file systems.

See also

link(S), rmdir(S), unlink(S)

Standards conformance

rename is conformant with:

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
and NIST FIPS 151-1.

rmdir

remove a directory

Syntax

```
cc ... -lc

int rmdir (path)
char *path;
```

Description

rmdir removes the directory named by the pathname pointed to by *path*. The directory must not have any entries other than "." and "..".

A directory is not removed when all of the following conditions are true:

- the parent directory has the sticky bit set
- the parent directory is not owned by the user
- the directory is not owned by the user
- the directory is not writable by the user
- the user is not super user

Return value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

Diagnostics

The **rmdir** function will fail if:

[EACCES]	• Search permission is denied for a component of the path prefix. • Write permission is denied on the directory containing the directory to be removed.
[EBUSY]	The directory to be removed is the mount point for a mounted filesystem.
[EEXIST]	The directory contains entries other than those for "." and "..".
[EFAULT]	<i>path</i> points outside the process's allocated address space.

[EINVAL]	• The current directory may not be removed. • The "." entry of a directory may not be removed.
[EIO]	An I/O error occurred while accessing the filesystem.
[EMULTIHOP]	Components of <i>path</i> require hopping to multiple remote machines.
[ENOENT]	The named directory does not exist.
[ENOLINK]	<i>path</i> points to a remote machine, and the link to that machine is no longer active.
[ENOTDIR]	A component of the path prefix is not a directory.
[EROFS]	The directory entry to be removed is part of a read-only filesystem.

See also

mkdir(C), mkdir(S), rmdir(C), rm(C)

Standards conformance

rmdir is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1); and NIST FIPS 151-1.

sc_init

scanCode Application Programming Interface (API) initialization and internal administration functions

Syntax

```
cc ... -lsc_s

#include <scanCode.h>

int sc_init (filedes)
int filedes;

int sc_exit ()

int sc_mapinit (filedes)
int filedes;

ulong *sc_getkbmap ()

void sc_receive_kb (scanCode)
scanCode_t scanCode;

keymap_t *sc_getkeymap ()

void sc_setkeymap (keymap_ptr)
keymap_t *keymap_ptr;

char *sc_getfkeystr (keyno)
int keyno;

int sc_setfkeystr (keyno, string)
int keyno;
char *string;

char sc_getled ()

int sc_setled (value)
char value;
```

Description

sc_init takes a file descriptor parameter and initializes the tty software to accept scanCode input. It performs the following steps:

1. Calls **sc_raw(S)** to get the current scanCode modes and put the tty into raw scanCode mode, saving the original modes to be restored later. If **KB_ISSCANCODE** is not set, the routine returns failure status.
2. Calls **sc_mapinit(S)** to read the *keymap* table, the *mapchan* table and the **sc_bitmask** array into local data structures for later use by the mapping functions.

3. Gets the current tty modes and XON/XOFF characters with an `XCGETA ioctl` call and saves them to be restored later.
4. Sets the correct tty modes (raw, echo off) with an `XCSETA ioctl` call.
5. Saves the state of the keyboard LEDs (if running on the console) to be restored later.

`sc_exit` restores the terminal mode and tty settings that `sc_init` saves. Do not call `sc_exit` if you did not use `sc_init` to initialize scancode mode. `sc_exit` performs the following steps:

1. Calls `sc_setinfo(S)` to restore the tty scancode modes.
2. Restores the tty modes and XON/XOFF characters with an `XCSETA ioctl` call.
3. Restores the state of the keyboard LEDs (if running on the console).

`sc_mapinit` takes a file descriptor as a parameter and initializes the maps used by the input/output mapping functions. It reads the *keymap* and *mapchan* tables into local data structures, constructs the mapcode-to-scancode table used by `sc_str2kb(S)`, and initializes the function key table. `sc_mapinit` provides a lower level initialization interface than `sc_init`. If the application itself sets tty modes, it should call `sc_mapinit` instead of `sc_init`. The application can only use one of these two functions. It does not need to call either function to simply read scancodes or mapcodes or to translate scancodes to mapcodes. However, if the desired end result is ASCII data, or if the application wants the API to do screen switching, the application must call one of these routines.

`sc_getkbmap` returns a pointer to *sc_bitmap*, which contains the make/break state of all the keys on the keyboard. The `sc_kb2mapcode(S)` and `sc_receive_kb(S)` functions maintain this map and they are called by `sc_readkb(S)`, `sc_readmapcode(S)` and `sc_readstr(S)`. If the "read" functions have not read all the characters in the input buffer, *sc_bitmap* will not accurately reflect the current state of the keyboard.

`sc_receive_kb` parses an input scancode to maintain the internal state of the (Shift), (Ctrl), (Alt), and (Lock) keys. It also performs screen switching (on the console only). `sc_receive_kb` returns a scancode that is usually the same as its input parameter. The application needs to call `sc_receive_kb` only if it reads scancodes on its own and does not want any other translation performed but does want screen switching to occur. If the application uses any other input or translation routines (such as `sc_readkb(S)` or `c_kb2mapcode(S)`) it should not call `sc_receive_kb`.

`sc_getkeymap` returns the current active keymap table in the API. `sc_setkeymap` takes a pointer to a *keymap* structure as a parameter and sets the current active keymap table in the API.

`sc_getfkeystr` returns a pointer to the character string associated with the function key specified by the input parameter. A parameter of 1 returns (F1), and so on. `sc_setfkeystr` sets the character string of the function key

associated with the input parameter. A parameter of 1 causes <F1> to be set, and so on.

sc_getled gets the current keyboard LED state. **sc_setled** sets the keyboard LED state. These functions only work if the application is running on the console. The LED state is defined as follows:

```
#define LED_CLK 0x01    /* caps lock on */
#define LED_NLK 0x02    /* num lock on */
#define LED_SLK 0x04    /* scroll lock on */
```

Diagnostics

sc_init, **sc_exit**, **sc_mapinit**, and **sc_setfkeystr** return 0 on success or -1 on failure. **sc_getkbmap** returns a NULL pointer if **sc_bitmap** has not been initialized. **sc_receive_kb** returns 0 if its input scancode causes a screen switch. **sc_getkeymap** returns a NULL pointer if the **keymap** table has not been initialized. **sc_getfkeystr** returns a NULL pointer if the function key table has not been initialized. **sc_setled** returns -1 if it cannot set the keyboard LED state.

If an API function returns failure status, the global variable **sc_error** might contain a number corresponding to an error condition listed in **scancode.h**.

Notes

The API does not set the XON/XOFF characters for scancode mode. The application must set these characters.

sc_init(S) and **sc_mapinit(S)** affect only one terminal at a time, and the application using the API affects only the terminal specified for **sc_init(S)** or **sc_mapinit(S)**.

sc_getkeymap does not return the **keymap** table stored in the kernel.

See also

sc_kb2mapcode(S),	sc_mapinit(S),	sc_raw(S),	sc_readkb(S),
sc_readmapcode(S),	sc_readstr(S),	sc_receive_kb(S),	sc_setinfo(S),
sc_str2kb(S)			

Standards conformance

sc_exit, **sc_mapinit**, **sc_getkbmap**, **sc_receive_kb**, **sc_getkeymap**, **sc_setkeymap**, **sc_getfkeystr**, **sc_setfkeystr**, **sc_getled**, and **sc_setled** are extensions of AT&T System V provided by the Santa Cruz Operation.

sc_raw

scancode Application Programming Interface (API) line-discipline and screen-switching functions

Syntax

```
cc ... -lsc_s

#include <scancode.h>

int sc_raw (filedes)
int filedes;

int sc_unraw (filedes)
int filedes;

int sc_getinfo (filedes)
int filedes;

int sc_setinfo (filedes, value)
int filedes;
int value;

int sc_copyscstate (srcfiledes, destfiledes)
int srcfiledes;
int destfiledes;

char sc_getscreenswitch ()

int sc_setscreenswitch (mode)
char mode;
```

Description

sc_raw - turns off scancode translation and returns the original mode
sc_unraw - turns on scancode translation and returns the original mode
sc_getinfo - returns scancode information
sc_setinfo - sets scancode information from input parameter
sc_copyscstate - copies the scancode state from one tty to another
sc_getscreenswitch - gets scancode screen switch keys
sc_setscreenswitch - sets scancode screen switch keys

sc_raw, **sc_unraw**, **sc_getinfo**, **sc_setinfo**, and **sc_copyscstate** perform line-discipline-level ioctl calls to get or set keyboard parameters (such as RAW/XLATE mode).

sc_raw turns off scancode translation with a **TCSETSC ioctl** call and returns the original mode.

sc_unraw turns on scancode translation with a **TCSETSC ioctl** call and returns the original mode.

sc_getinfo returns the scancode capability information it gets from the **TCGETSC ioctl** call. **sc_setinfo** sets the scancode capability information from its input parameter, using the **TCSETSC ioctl** call.

sc_copyscstate copies the scancode state from one tty to another. The information it copies includes the keymap and the function key strings, which are stored in the kernel. If the application modifies the API's version of the keymap through a call to **sc_setscreenswitch**, **sc_copyscstate** does not copy the updated keymap information. **sc_copyscstate** takes source and destination file descriptors as parameters.

sc_getscreenswitch and **sc_setscreenswitch** get and set the combination(s) of function keys and mode keys (<Shift>, <Ctrl>, and <Alt>) that produce a screen switch. (The API provides screen-switching capability on the console only.)

sc_getscreenswitch returns the current mode key information.

sc_setscreenswitch determines which mode keys are required (in combination with a function key) to switch screens. The input parameter, of type *char*, has the following bits defined:

```
#define MODE_OFF    0x00 /* disable screen switching */
#define MODE_SHIFT  0x01 /* use shift to switch screens */
#define MODE_CTRL   0x02 /* use ctrl to switch screens */
#define MODE_ALT    0x04 /* use alt to switch screens */
#define MODE_SC     0x08 /* use shift, ctrl to switch screens */
#define MODE_SA     0x10 /* use shift, alt to switch screens */
#define MODE_CA     0x20 /* use ctrl, alt to switch screens */
#define MODE_SCA    0x40 /* use shift, ctrl, alt to switch screens */
```

By setting more than one bit, you can define more than one combination of mode keys to enable screen switching. Calling **sc_setscreenswitch** with a value of zero disables screen switching.

If the application does not set screen switch mode with a call to **sc_setscreenswitch**, screen switching is determined by the application's original keyboard mapping.

Diagnostics

sc_raw and **sc_unraw** return the old scancode mode on success or -1 on failure. **sc_getinfo** returns -1 if it cannot obtain the scancode capability information from the **TCGETSC ioctl** call. **sc_setinfo** and **sc_copyscstate** return 0 on success or -1 on failure. A return value of 0x80 for **sc_getscreenswitch** indicates one of two things: either the application is not running on the console so screen switching is unavailable, or the screen switch mode has not

been set through the API, and the original screen switch entries in the keymap will be used. `sc_setscreenswitch` returns -1 if the application is not running on the system console.

If an API function returns failure status, the global variable `sc_error` might contain a number corresponding to an error condition listed in `scancode.h`.

Notes

`sc_init(S)` and `sc_mapinit(S)` affect only one terminal at a time, and the application using the API affects only the terminal specified for `sc_init(S)` or `sc_mapinit(S)`.

See also

`mapkey(M)`

Standards conformance

`sc_raw`, `sc_unraw`, `sc_getinfo`, `sc_setinfo`, `sc_setscreenswitch`, and `sc_getscreenswitch` are extensions of AT&T System V provided by the Santa Cruz Operation.

sc_readkb

scancode translation functions

Syntax

```
cc ... -lsc_s

#include <scancode.h>

scancode_t sc_readkb (filedes)
int filedes;

mapcode_t sc_readmapcode (filedes)
int filedes;

char *sc_readstr (filedes)
int filedes;

mapcode_t sc_kb2mapcode (scancode)
scancode_t scancode;

char *sc_mapcode2str (mapcode)
mapcode_t mapcode;

uchar sc_mapin (c)
uchar c;

uchar *sc_mapout (c)
uchar c;

scancode_t *sc_str2kb (c)
char c;

scancode_t *sc_mapcode2kb (mapcode)
mapcode_t mapcode;
```

Description

sc_readkb returns the next input scancode. The application does not need to call **sc_init(S)** or **sc_mapinit(S)** to use this routine.

sc_readmapcode returns the next input mapcode. The application does not need to call **sc_init(S)** or **sc_mapinit(S)** to use this routine.

sc_readstr returns a pointer to an ASCII string that contains either one character or, in the case of a function key input, a string of characters. The application must call either **sc_init** or **sc_mapinit** before **sc_readstr**.

sc_readkb, **sc_readmapcode**, and **sc_readstr** all maintain the API's internal state variables and do screen switching if the application has set a screen switch mode with **sc_setscreenswitch(S)**. (The API provides screen-switching capability on the console only.) Therefore, the application should not call **sc_receive_kb(S)**, **sc_kb2mapcode**, or **sc_mapcode2str** in combination with these functions. For example, if further processing of a scancode is required, the application should read the scancode itself and not call **sc_readkb**.

Not every key generates a unique scancode. For codeset 1, each non-unique key generates a multi-byte scancode sequence, an 0xE0 byte followed by a non-unique scancode. When translating scancodes, we turn these sequences into SCO-invented codes called mapcodes. These mapcodes have values greater than the normal range of scancodes (see the values above 127 in */usr/lib/keyboard/keys*). This allows translation functions to look up key functionality on a single linear table.

sc_kb2mapcode and **sc_mapcode2str** provide a lower level translation interface than **sc_readkb**, **sc_readmapcode**, and **sc_readstr**. Do not use the two sets of routines in combination with each other. An application should use **sc_kb2mapcode** and **sc_mapcode2str** if it is not using the API to read scancodes from the terminal.

sc_kb2mapcode takes a scancode as input and returns a mapcode. It parses multi-byte scancode sequences that map to a single mapcode, returning zero for scancodes that do not directly produce mapped output. **sc_kb2mapcode** maintains the state of the keyboard and performs a screen switch if the application has set a screen switch mode with **sc_setscreenswitch(S)**. (The API provides screen-switching capability on the console only.) On the first input following a screen switch (that is, when you switch *to* the application), **sc_kb2mapcode** updates the internal representation of the keyboard and the LED state. The application does not need to call **sc_init** or **sc_mapinit** to use **sc_kb2mapcode** unless the application is using the API for screen switching.

sc_mapcode2str takes a mapcode as its argument and returns the ASCII string to which that mapcode maps. The input can be expanded to a character string on output if the mapcode maps to a function key. The application must call **sc_init** or **sc_mapinit** to use **sc_mapcode2str**.

sc_mapin is an international input mapping (mapchan-style) function. It takes a character as input and returns a mapped character.

sc_mapout is an international output mapping function. It takes a character as input and returns a pointer to a string of characters that constitute the mapped output.

`sc_str2kb` takes a keytop value as input and returns a pointer to a string of scancodes that produce that keytop value. This mapping is non-deterministic; for example, there is no way to tell whether LeftShift or RightShift should be used to produce a capital letter. In cases where an application expects to use international output mapping, it must call `sc_mapout` before calling `sc_str2kb`. `sc_str2kb` translates only (Space), (Newline), (Tab), and printable ASCII characters.

`sc_mapcode2kb` takes a mapcode as input and returns a pointer to a string that contains a sequence of scancodes. This sequence, in turn, maps to the input mapcode.

Before an application calls `sc_str2kb` or `sc_mapcode2kb`, it must call `sc_init` or `sc_mapinit` to initialize the translation tables that `sc_str2kb` and `sc_mapcode2kb` use.

Diagnostics

`sc_readkb` and `sc_readmapcode` return zero on a read error or if there are no characters to read. `sc_readstr` returns a NULL pointer if neither `sc_init` nor `sc_mapinit` is called before `sc_readstr`. `sc_kb2mapcode` returns zero for scancodes that do not directly produce mapped output. `sc_mapcode2str` returns NULL if the input mapcode represents a modifier key that does not directly produce mapped output. `sc_mapin` returns a NULL character if the input is a dead key or part of a compose sequence. `sc_mapout` returns a NULL pointer if the input character does not result in any mapped output.

If an API function returns failure status, the global variable `sc_error` might contain a number corresponding to an error condition listed in *scancode.h*.

Notes

`sc_init` and `sc_mapinit` affect only one terminal at a time, and an application using the API affects only the terminal specified for `sc_init` or `sc_mapinit`.

See also

`mapchan(M)`, `sc_init(S)`, `sc_mapinit(S)`, `sc_receive_kb(S)`,
`sc_setscreenswitch(S)`

Standards conformance

`sc_readkb`, `sc_readmapcode`, `sc_readstr`, `sc_kb2mapcode`, `sc_mapcode2str`, `sc_mapin`, `sc_mapout`, `sc_str2kb`, and `sc_mapcode2kb` are extensions of AT&T System V provided by the Santa Cruz Operation.

Examples

Here are some examples of various ways of using the API routines:

Scenario 1: The API switches the tty modes and does screen switching; the application wants to receive scancodes.

```

sc_init(fd);
sc_setscreenswitch(mode);
while (condition) {
    scancode = sc_readkb(fd);
    .
    .
}
sc_exit();

```

Scenario 2: The application switches tty modes and reads scancodes itself, but wants the API to perform screen switches.

```

sc_mapinit(fd);
sc_setscreenswitch(mode);
while (condition) {
    /* Next line is pseudo-code */
    (obtain scancode)
    sc_receive_kb(scancode);
    .
    .
}

```

Scenario 3: The API switches tty modes and performs screen switches; the application obtains scancodes itself and wants them translated to ASCII.

```

sc_init(fd);
sc_setscreenswitch(mode);
while (condition) {
    /* Next line is pseudo-code */
    (obtain scancode)
    mapcode = sc_kb2mapcode(scancode);
    str_pointer = sc_mapcode2str(mapcode);
    .
    .
}
sc_exit();

```

Scenario 4: The application just wants scancodes translated to ASCII.

```

while (condition) {
    /* Next line is pseudo-code */
    (obtain scancode)
    mapcode = sc_kb2mapcode(scancode);
    str_pointer = sc_mapcode2str(mapcode);
    .
    .
}

```

Scenario 5: The application sets tty modes and wants to read ASCII with no screen switches.

```
sc_mapinit(fd);
sc_setscreenswitch(MODE_OFF);
while (condition) {
    str_pointer = sc_readstr(fd);
    .
    .
}
```

(S)

scanf

 convert formatted input

Syntax

```
cc ... -lc

#include <stdio.h>

int scanf (format [, pointer ] . . .
const char *format;

int fscanf (stream, format [, pointer ] . . .
FILE *stream;
const char *format;

int sscanf (s, format [, pointer ] . . .
char *s;
const char *format;
```

Description

scanf - convert formatted input

fscanf - convert formatted input

sscanf - convert formatted input

The **scanf** function reads from the standard input stream *stdin*. **fscanf** reads from the named input *stream*. **sscanf** reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string *format* described below, and a set of *pointer* arguments indicating where the converted input should be stored. The results are undefined in that there are insufficient *args* for the format. If the format is exhausted while *args* remain, the excess *args* are simply ignored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. White-space characters (blanks, tabs, new-lines, or form-feeds) which, except in two cases described below, cause input to be read up to the next non-white-space character.
2. An ordinary character (not %), which must match the next character of the input stream.
3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width, an optional *l* (ell) or *h* indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. The suppression of assignment provides a way of describing an input field which is to be skipped. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted. For all descriptors except "[and "c", white space leading an input field is ignored.

The conversion code indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. For a suppressed field, no pointer argument is given. The following conversion codes are legal:

- %** a single % is expected in the input at this point; no assignment is done.
- d** a decimal integer is expected; the corresponding argument should be an integer pointer.
- u** an unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.
- o** an octal integer is expected; the corresponding argument should be an integer pointer.
- x** a hexadecimal integer is expected; the corresponding argument should be an integer pointer.
- i** an integer is expected; the corresponding argument should be an integer pointer. It stores the value of the next input item interpreted according to C conventions: a leading "0" implies octal; a leading "0x" implies hexadecimal; otherwise, decimal.
- n** stores in an integer argument the total number of characters (including white space) that have been scanned so far since the function call. No input is consumed.
- e,f,g** a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a *float*. The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an E or an e, followed by an optional + or - , followed by an integer.
- s** a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating \0, which is added automatically. The input field is terminated by a white-space character.
- c** a character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-space character, use %1s. If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read.

- [indicates string data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, which is called the *scanset*, and a right bracket; the input field is the maximal sequence of input characters consisting entirely of characters in the scanset. The caret (^), when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters **not** contained in the remainder of the scanset string. There are some conventions used in the construction of the scanset. A range of characters may be represented by the construct *first-last*, thus [0123456789] may be expressed [0-9]. Using this convention, *first* must be lexically less than or equal to *last*, or else the dash stands for itself. The dash also stands for itself whenever it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a caret) of the scanset, and in this case it is not syntactically interpreted as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating \0, which is added automatically. At least one character must match for this conversion to be considered successful.

The conversion characters *d*, *u*, *o*, *x* and *i* may be preceded by *l* or *h* to indicate that a pointer to *long* or to *short* rather than to *int* is in the argument list. Similarly, the conversion characters *e*, *f*, and *g* may be preceded by *l* to indicate that a pointer to *double* rather than to *float* is in the argument list. The *l* or *h* modifier is ignored for other conversion characters.

The *scanf* function conversion terminates at EOF, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

The *scanf* function returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, EOF is returned.

Diagnostics

These functions return EOF on end of input and an int count for missing or illegal data items.

Notes

Trailing white space (including a new-line) is left unread unless matched in the control string.

scanf has a file limit of FOPEN_MAX; for more information see *stdio(S)*.

See also

getc(S), printf(S), stdio(S), strtod(S), strtol(S)

Standards conformance

fscanf, scanf and sscanf are conformant with:

AT&T SVID Issue 2;
 X/Open Portability Guide, Issue 3, 1989;
 ANSI X3.159-1989 Programming Language — C;
 IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);
 and NIST FIPS 151-1.

Examples

The call:

```
int n ; float x; char name[50];
n = scanf("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

assigns to **n** the value 3, to **i** the value 25, to **x** the value 5.432, and **name** contains *thompson*\0 . Or:

```
int i, j; float x; char name[50];
(void) scanf("%i%2d%f%d %0-9] ", &j, &i, &x, name);
```

with input:

```
011 56789 0123 56a72
```

assigns 9 to **j**, 56 to **i**, 789.0 to **x**, skip 0123, and place the string 56\0 in **name**. The next call to **getchar** (see **getc(S)**) returns a. Or:

```
int i, j, s, e; char name[50];
(void) scanf("%i %i %n%s%n", &i, &j, &s, name, &e);
```

with input:

```
0x11 0xy johnson
```

assigns 17 to **i**, 0 to **j**, 6 to **s**, places the string *xy*\0 in **name**, and assigns 8 to **e**. Thus, the length of **name** is **e** - **s** = 2 . The next call to **getchar** (see **getc(S)**) returns a blank.

`__scoinfo`

get kernel and system information

Syntax

```
#include <sys/utsname.h>

int __scoinfo (buffer, bufsize)
struct scoutsname *buffer;
int bufsize;
```

Description

The `__scoinfo` system call fills the `scoutsname` structure pointed to by *buffer* with information on kernel and system parameters. The following are the contents of this structure:

```
struct scoutsname {
    char        sysname[9];
    char        nodename[9];
    char        release[16];
    char        kernelid[20];
    char        machine[9];
    char        bustype[9];
    char        sysserial[10];
    unsigned short sysorigin;
    unsigned short sysoem;
    char        numuser[9];
    unsigned short numcpu;
};
```

The information returned by `__scoinfo` is displayed by the command `uname -X`.

Diagnostics

`__scoinfo` fails if `errno` contains either of the following errors:

<code>[EINVAL]</code>	<i>bufsize</i> is larger than the <code>scoutsname</code> structure
<code>[EFAULT]</code>	<i>buffer</i> points to an illegal address

Return value

Upon error, -1 is returned and `errno` is set to indicate the error.

See also

`uname(C)`

Standards conformance

`__scoinfo` is an extension of AT&T System V provided by The Santa Cruz Operation, Inc.

(S)

sdenter, sdleave

synchronizes access to a shared data segment

Syntax

```
cc ... -lx

#include <sys/sd.h>

int sdenter(addr, flags)
char *addr;
int flags;

int sdleave(addr)
char *addr;
```

Description

sdenter - access shared data segment

sdleave - modify shared data segment

sdenter is used to indicate that the current process is about to access the contents of a shared data segment. *addr* is the valid return code from a previous **sdget** call. The actions performed depend on the value of *flags*. *flags* values are formed by OR-ing together entries from the following list:

SD_NOWAIT	If another process has called sdenter but not sdleave for the indicated segment, and the segment was not created with the SD_UNLOCK flag set, return an ENAVAIL error instead of waiting for the segment to become free.
SD_WRITE	Indicates that the process wants to write data to the shared data segment. A process that has attached to a shared data segment with the SD_RDONLY flag set is not allowed to enter with the SD_WRITE flag set.

sdleave is used to indicate that the current process is done modifying the contents of a shared data segment.

Only changes made between invocations of **sdenter** and **sdleave** are guaranteed to be reflected in other processes. **sdenter** and **sdleave** are very fast; consequently, it is recommended that they be called frequently rather than leave **sdenter** in effect for any period of time. In particular, system calls should be avoided between **sdenter** and **sdleave** calls.

The **fork** system call is forbidden between calls to **sdenter** and **sdleave** if the segment was created without the **SD_UNLOCK** flag.

Return value

Successful calls return 0. Unsuccessful calls return -1, and **errno** is set to indicate the error. **errno** is set to **EINVAL** if a process does an **scenter** with the **SD_WRITE** flag set and the segment is already attached with the **SD_RDONLY** flag set. **errno** is set to **ENAVAIL** if the **SD_NOWAIT** flag is set for **scenter** call and the shared data segment is not free.

Notes

This feature is a XENIX specific enhancement and may not be present on all UNIX implementations.

See also

sdget(S), sdgetv(S)

Standards conformance

scenter and **sdleave** are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

sdget, sdfree

attaches and detaches a shared data segment.

Syntax

```

cc ... -lx

#include <sys/sd.h>

char *sdget(path, flags, [size, mode])
char *path;
int flags, mode;
long size;

int sdfree(addr);
char *addr;

```

Description

sdget attaches a shared data segment to the data space of the current process. The actions performed are controlled by the value of *flags*. *flags* values are constructed by OR-ing flags from the following list:

- SD_RDONLY** Attach the segment for reading only.
- SD_WRITE** Attach the segment for both reading and writing.
- SD_CREAT** If the segment named by *path* exists and is not in use (active), this flag has the same effect as creating a segment from scratch. Otherwise, the segment is created according to the values of *size* and *mode*. Read and write access to the segment is granted to other processes based on the permissions passed in *mode*, and functions the same as those for regular files. Execute permission is meaningless. The segment is initialized to contain all zeroes.
- SD_UNLOCK** If the segment is created because of this call, the segment is made so that more than one process can be between **sdenter** and **sdleave** calls.

sdfree detaches the current process from the shared data segment that is attached at the specified address. If the current process has done **sdenter** but not an **sdleave** for the specified segment, **sdleave** is done before detaching.

When no process remains attached to the segment, the contents of that segment disappear, and no process can attach to the segment without creating it by using the **SD_CREAT** flag in **sdget**. **errno** is set to **EEXIST** if a process tries to create a shared data segment that exists and is in use. **errno** is set to **ENOTNAM** if a process attempts an **sdget** on a file that exists but is not a shared data type.

Notes

Use of the SD_UNLOCK flag on systems without hardware support for shared data may cause severe performance degradation.

For 286 programs, it is strongly recommended that **sdget** and other shared data functions be reserved for large model programs only. Small or middle model programs that attempt to use shared data may run out of available memory. Also, due to the 286 hardware, it is not possible to enforce the read-only aspect of small model shared data. However, read-only segments are honored in large model programs.

The 386 provides a 32 bit address space, even in small model. As a result, shared data may be conveniently used without regard to the restrictions that apply to 286 programs.

sdget automatically increments the process's original break value to the memory location immediately after the shared data segment. This affects subsequent **sbrk** or **brk** calls which attempt to restore the original break value. In particular, attempts to restore the break value to its value before the **sdget** call causes an error.

This feature is a XENIX specific enhancement and may not be present in all UNIX implementations. This routine must be linked using the linker option **-lx**.

The *size* variable in **sdget** has changed from unsigned to long between UNIX Version 3.0 and UNIX System V. Although this requires that source code be modified to use a long *size* parameter when compiling with the libraries, an unsigned *size* parameter is still correctly interpreted by the kernel when passed by binaries compiled with the Version 3.0 libraries.

Return value

On successful completion, the address at which the segment was attached is returned. Otherwise, -1 is returned, and **errno** is set to indicate the error. **errno** is set to **EINVAL** if a process does an **sdget** on a shared data segment to which it is already attached. **errno** is set to **EEXIST** if a process tries to create a shared data segment that exists and is in use. **errno** is set to **ENOTNAM** if a process attempts an **sdget** on a file that exists but is not a shared data type.

The mode parameter must be included on the first call of the **sdget** function.

See also

sbrk(S), **sdenter(S)**, **sdgetv(S)**

Standards conformance

sdget and **sdfree** are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

sdgetv, sdwaitv

synchronizes shared data access

Syntax

```
cc ... -lx

#include <sys/sd.h>

int sdgetv(addr)
char *addr;

int sdwaitv(addr, vnum)
char *addr;
int vnum;
```

Description

sdgetv - returns version number of shared data segment

sdwaitv - synchronizes shared data access

sdgetv and **sdwaitv** may be used to synchronize cooperating processes that are using shared data segments. The return value of both routines is the version number of the shared data segment attached to the process at address *addr*. The version number of a segment changes whenever some process does an **sdleave** for that segment.

sdgetv simply returns the version number of the indicated segment.

sdwaitv forces the current process to sleep until the version number for the indicated segment is no longer equal to *vnum*.

Return value

Upon successful completion, both **sdgetv** and **sdwaitv** return a positive integer that is the current version number for the indicated shared data segment. Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

See also

sdenter(S), sdget(S)

Standards conformance

sdgetv and **sdwaitv** are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

seed: get_seed, set_seed

obtain or set seed for random number generator

Syntax

```
cc ... -lprot

#include <sys/types.h>
#include <sys/macro.h>
#include <sys/security.h>
#include <sys/audit.h>
#include <sys/param.h>
#include <sys/filsys.h>
#include <prot.h>

long get_seed (prpwd)
struct pr_passwd *prpwd;

void set_seed (seed)
long seed;
```

Description

get_seed - obtain seed for random number generator

set_seed - set seed for random number generator

These routines handle seed management for users of the random number generator used in the Authentication subsystem. They exist to centralize control so that multiple seed algorithms can be handled together and so that different routines do not all set an initial seed (the randomness desired is promoted here).

The **get_seed** routine uses an automatic seed generation algorithm to choose a value to return. Currently, it does not use the argument *prpwd*, but to be compatible with more secure versions of the system, it should pass in the pointer to the user of interest's protected password structure. If no user is relevant to setting the seed, *prpwd* should be set to NULL.

The **set_seed** routine primes the program's random number generator (using **drand48(S)**) to the *seed* passed in. The routine ignores all requests except the first one, so that the generator is not reset after its initial invocation.

Return value

Upon successful completion, the **get_seed** routine returns an integer value calculated by an automatic seed generator. If an error occurs during seed generation, **get_seed** will return a value of -1. No return value or errors are defined for the **set_seed** routine.

Notes

The **drand48(S)** routines used on an implementation should be tested to produce good uniformity, a long period, and a large name space since not all ports test for this. Poor performance of these routines translates into predictable results in such vital operations as password generation.

See also

drand48(S)

Standards conformance

The **get_seed** and **set_seed** routines are an extension of AT&T System V provided by the Santa Cruz Operation.

select

synchronous I/O multiplexing

Syntax

```
cc ... -lc

#include <sys/types.h>
#include <sys/times.h>
#include <sys/select.h>

int select(nfds, readfds, writefds, exceptfds, timeout)
int nfds;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;

FD_CLR(fd, &fdset)
int fd;
fd_set fdset;

FD_ISSET(fd, &fdset)
int fd;
fd_set fdset;

FD_SET(fd, &fdset)
int fd;
fd_set fdset;

FD_ZERO(&fdset)
fd_set fdset;
```

Description

select examines the I/O descriptor sets whose addresses are passed in *readfds*, *writefds*, and *exceptfds* to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first *nfds* descriptors are checked in each set; that is, the descriptors from 0 through *nfds*-1 in the descriptor sets are examined. On return, **select** replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets is returned in *nfound*.

The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such descriptor sets: **FD_ZERO(&fdset)** initializes a descriptor set *fdset* to the null set. **FD_SET(fd, &fdset)** includes a particular descriptor *fd* in *fdset*. **FD_CLR(fd, &fdset)** removes *fd* from *fdset*. **FD_ISSET(fd, &fdset)** is nonzero if *fd* is a member of *fdset*, zero otherwise. The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to **FD_SETSIZE**, which is normally at least equal to the maximum number of descriptors supported by the system.

If *timeout* is a non-NULL pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a NULL pointer, the select blocks indefinitely. To effect a poll, the *timeout* argument should be non-NULL, pointing to a zero-valued timeval structure.

Any of *readfds*, *writefds*, and *exceptfds* may be given as NULL pointers if no descriptors are of interest.

Return value

select returns the number of ready descriptors that are contained in the descriptor sets, or -1 if an error occurred. If the time limit expires then **select** returns 0. If **select** returns with an error, including one due to an interrupted call, the descriptor sets are unmodified.

Diagnostics

An error return from **select** indicates:

[EBADF]	One of the descriptor sets specified an invalid descriptor.
[EINTR]	A signal was delivered before the time limit expired and before any of the selected events occurred.
[EINVAL]	The specified time limit is invalid. One of its components is negative or too large. Or, the device driver being polled has not implemented select support.

Notes

select should probably return the time remaining from the original *timeout*, if any, by modifying the time value in place. This may be implemented in future versions of the system. Thus, it is unwise to assume that the *timeout* value is unmodified by the **select** call.

See also

read(S), **write(S)**

Standards conformance

select is not part of any currently supported standard; it was developed at the University of California at Berkeley and is used by permission.

semctl

semaphore control operations

Syntax

```
cc ... -lc

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (semid, semnum, cmd, arg)
int semid, cmd;
int semnum;
union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
} arg;
```

Description

The **semctl** system call provides a variety of semaphore control operations as specified by *cmd*.

The following *cmds* are executed with respect to the semaphore specified by *semid* and *semnum*:

- GETVAL** Return the value of *semval*. (READ)
- SETVAL** Set the value of *semval* to *arg.val*. (ALTER) When this *cmd* is successfully executed, the *semadj* value corresponding to the specified semaphore in all processes is cleared.
- GETPID** Return the value of *sempid*. (READ)
- GETNCNT** Return the value of *semncnt*. (READ)
- GETZCNT** Return the value of *semzcnt*. (READ)

The following *cmds* return and set, respectively, every *semval* in the set of semaphores.

- GETALL** Place *semvals* into array pointed to by *arg.array*. (READ)
- SETALL** Set *semvals* according to the array pointed to by *arg.array*. (ALTER)
When this *cmd* is successfully executed the *semadj* values corresponding to each specified semaphore in all processes are cleared.

The following *cmds* are also available:

- IPC_STAT** Place the current value of each member of the data structure associated with *semid* into the structure pointed to by *arg.buf*. (READ)
- IPC_SET** Set the value of the following members of the data structure associated with *semid* to the corresponding value found in the structure pointed to by *arg.buf*:
- ```
sem_perm.uid
sem_perm.gid
sem_perm.mode /* only low 9 bits */
```
- This *cmd* can only be executed by a process that has an effective user ID equal to either that of super-user, or to the value of *sem\_perm.cuid* or *sem\_perm.uid* in the data structure associated with *semid*.
- IPC\_RMID** Remove the semaphore identifier specified by *semid* from the system and destroy the set of semaphores and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of super-user, or to the value of *sem\_perm.cuid* or *sem\_perm.uid* in the data structure associated with *semid*.

The *semctl* system call fails if one or more of the following is true:

- [EACCES]** Operation permission is denied to the calling process.
- [EINVAL]** *semid* is not a valid semaphore identifier.
- [EINVAL]** *semnum* is less than zero or greater than *sem\_nsems*.
- [EINVAL]** *cmd* is not a valid command.
- [EPERM]** *cmd* is equal to *IPC\_RMID* or *IPC\_SET* and the effective user ID of the calling process is not equal to that of super-user or to the value of *sem\_perm.cuid* or *sem\_perm.uid* in the data structure associated with *semid*.
- [ERANGE]** *cmd* is *SETVAL* or *SETALL* and the value to which *semval* is to be set is greater than the system imposed maximum.

## Diagnostics

---

Upon successful completion, the value returned depends on *cmd* as follows:

|            |                       |
|------------|-----------------------|
| GETVAL     | The value of semval.  |
| GETPID     | The value of sempid.  |
| GETNCNT    | The value of semncnt. |
| GETZCNT    | The value of semzcnt. |
| All others | A value of 0.         |

Otherwise, a value of -1 is returned, and *errno* is set to indicate the error.

## See also

---

**semget(S), semop(S)**

## Standards conformance

---

**semctl** is conformant with:

AT&T SVID Issue 2;  
and X/Open Portability Guide, Issue 3, 1989.



# semget

---

get set of semaphores

## Syntax

---

```
cc ... -lc

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key, nsems, semflg)
key_t key;
int nsems, semflg;
```

## Description

---

The **semget** system call returns the semaphore identifier associated with *key*.

A semaphore identifier and associated data structure and set containing *nsems* semaphores are created for *key* if one of the following is true:

- *key* is equal to **IPC\_PRIVATE**.
- *key* does not already have a semaphore identifier associated with it, and (*semflg* & **IPC\_CREAT**) is “true”.

Upon creation, the data structure associated with the new semaphore identifier is initialized as follows:

- **sem\_perm.cuid**, **sem\_perm.uid**, **sem\_perm.cgid**, and **sem\_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order 9 bits of **sem\_perm.mode** are set equal to the low-order 9 bits of *semflg*.
- **sem\_nsems** is set equal to the value of *nsems*.
- **sem\_otime** is set equal to 0 and **sem\_ctime** is set equal to the current time.

The data structure associated with each semaphore in the set is not initialized. The function **semctl** with the command **setval** or **setall** can be used to initialize each semaphore.

The `semget` system call fails if one or more of the following is true:

- [EACCES] A semaphore identifier exists for *key*, but operation permission as specified by the low-order 9 bits of *semflg* would not be granted.
- [EEXIST] A semaphore identifier exists for *key*, but (*semflg* & `IPC_CREAT`) and (*semflg* & `IPC_EXCL`) are "true".
- [EINVAL] *nsems* is either less than or equal to zero or greater than the system-imposed limit.
- [EINVAL] A semaphore identifier exists for *key*, but the number of semaphores in the set associated with it is less than *nsems*, and *nsems* is not equal to zero.
- [ENOENT] A semaphore identifier does not exist for *key*, and (*semflg* & `IPC_CREAT`) is "false".
- [ENOSPC] A semaphore identifier is to be created, but the system-imposed limit on the maximum number of allowed semaphore identifiers system wide would be exceeded.
- [ENOSPC] A semaphore identifier is to be created, but the system-imposed limit on the maximum number of allowed semaphores system wide would be exceeded.

## Diagnostics

---

Upon successful completion, a non-negative integer, namely a semaphore identifier, is returned. Otherwise, a value of -1 is returned, and `errno` is set to indicate the error.

## See also

---

`semctl(S)`, `semop(S)`

## Standards conformance

---

`semget` is conformant with:

AT&T SVID Issue 2;  
and X/Open Portability Guide, Issue 3, 1989.

# semop

semaphore operations

---

## Syntax

---

```
cc ... -lc

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop (semid, sops, nsops)
int semid;
struct sembuf *sops;
unsigned nsops;
```

## Description

---

The **semop** system call is used to automatically perform an array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by *semid*. *sops* is a pointer to the array of semaphore-operation structures. *nsops* is the number of such structures in the array. The contents of each structure includes the following members:

```
short sem_num; /* semaphore number */
short sem_op; /* semaphore operation */
short sem_flg; /* operation flags */
```

Each semaphore operation specified by **semop** is performed on the corresponding semaphore specified by *semid* and *sem\_num*.

**semop** specifies one of three semaphore operations as follows:

If **semop** is a negative integer, one of the following occurs: {ALTER}

- If *semval* is greater than or equal to the absolute value of **semop**, the absolute value of **semop** is subtracted from *semval*. Also, if (*sem\_flg* & SEM\_UNDO) is "true", the absolute value of **semop** is added to the calling process's *semadj* value (see *exit(S)*) for the specified semaphore.
- If *semval* is less than the absolute value of **semop** and (*sem\_flg* & IPC\_NOWAIT) is "true", **semop** returns immediately.

If **semval** is less than the absolute value of **semop** and (**sem\_flg** & **IPC\_NOWAIT**) is "false", **semop** increments the **semncnt** associated with the specified semaphore and suspend execution of the calling process until one of the following conditions occurs:

- **semval** becomes greater than or equal to the absolute value of **semop**. When this occurs, the value of **semncnt** associated with the specified semaphore is decremented, the absolute value of **semop** is subtracted from **semval** and, if (**sem\_flg** & **SEM\_UNDO**) is "true", the absolute value of **semop** is added to the calling process's **semadj** value for the specified semaphore.
- The **semid** for which the calling process is awaiting action is removed from the system (see LR **semctl S**). When this occurs, **errno** is set equal to **EIDRM**, and a value of -1 is returned.
- The calling process receives a signal that is to be caught. When this occurs, the value of **semncnt** associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in LR signal **S**.

If **semop** is a positive integer, the value of **semop** is added to **semval** and, if (**sem\_flg** & **SEM\_UNDO**) is "true", the value of **semop** is subtracted from the calling process's **semadj** value for the specified semaphore. {ALTER}

If **semop** is zero, one of the following occurs: {READ}

If **semval** is zero, **semop** returns immediately.

If **semval** is not equal to zero and (**sem\_flg** & **IPC\_NOWAIT**) is "true", **semop** returns immediately.

If **semval** is not equal to zero and (**sem\_flg** & **IPC\_NOWAIT**) is "false", **semop** increments the **semzcnt** associated with the specified semaphore and suspends execution of the calling process until one of the following occurs:

- **semval** becomes zero, at which time the value of **semzcnt** associated with the specified semaphore is decremented.
- The **semid** for which the calling process is awaiting action is removed from the system. When this occurs, **errno** is set equal to **EIDRM**, and a value of -1 is returned.
- The calling process receives a signal that is to be caught. When this occurs, the value of **semzcnt** associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in signal(S).

The **semop** system call fails if one or more of the following is true for any of the semaphore operations specified by *sops*:

- [E2BIG]      *nsops* is greater than the system-imposed maximum.
- [EACCES]    Operation permission is denied to the calling process.
- [EAGAIN]    The operation would result in suspension of the calling process but (*sem\_flg* & *IPC\_NOWAIT*) is "true".
- [EFAULT]    *sops* points to an illegal address.
- [EFBIG]      *sem\_num* is less than zero or greater than or equal to the number of semaphores in the set associated with *semid*.
- [EINVAL]    *semid* is not a valid semaphore identifier.
- [EINVAL]    The number of individual semaphores for which the calling process requests an *SEM\_UNDO* would exceed the limit.
- [ENOSPC]    The limit on the number of individual processes requesting an *SEM\_UNDO* would be exceeded.
- [ERANGE]    An operation would cause a *semval* to overflow the system-imposed limit.
- [ERANGE]    An operation would cause a *semadj* value to overflow the system-imposed limit.

Upon successful completion, the value of *sempid* for each semaphore specified in the array pointed to by *sops* is set equal to the process ID of the calling process.

## *Diagnostics*

---

If **semop** returns due to the receipt of a signal, a value of -1 is returned to the calling process and *errno* is set to *EINTR*. If it returns due to the removal of a *semid* from the system, a value of -1 is returned and *errno* is set to *EIDRM*.

Upon successful completion, a value of zero is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

## *See also*

---

*exec(S)*, *exit(S)*, *fork(S)*, *semctl(S)*, *semget(S)*

## *Standards conformance*

---

**semop** is conformant with:

AT&T SVID Issue 2;  
and X/Open Portability Guide, Issue 3, 1989.

# setbuf, setvbuf

assign buffering to a stream

## Syntax

```
cc ... -lc

#include <stdio.h>

void setbuf (stream, buf)
FILE *stream;
char *buf;

int setvbuf (stream, buf, type, size)
FILE *stream;
char *buf;
int type;
size_t size;
```

## Description

**setbuf** - assign buffer to a stream

**setvbuf** - assign buffer after a stream has been opened

The **setbuf** function may be used after a stream has been opened but before it is read or written. It causes the array pointed to by *buf* to be used instead of an automatically allocated buffer. If *buf* is the NULL pointer, input/output is completely unbuffered.

A constant **BUFSIZ**, defined in the *<stdio.h>* header file, tells how big an array is needed:

```
char buf[BUFSIZ];
```

**setvbuf** may be used after a stream has been opened but before it is read or written. *type* determines how *stream* is buffered. Legal values for *type* (defined in *<stdio.h>*) are:

- \_IOFBF** causes input/output to be fully buffered.
- \_IOLBF** causes output to be line buffered; the buffer is flushed when a new line is written, the buffer is full, or input is requested.
- \_IONBF** causes input/output to be completely unbuffered.

If *buf* is not the NULL pointer, the array it points to is used for buffering, instead of an automatically allocated buffer. *size* specifies the size of the buffer to be used. The constant **BUFSIZ** in *<stdio.h>* is suggested as a good buffer size. If input/output is unbuffered, *buf* and *size* are ignored.

By default, output to a terminal is line-buffered and all other input/output is fully buffered.

## ***Return value***

---

If an illegal value for *type* or *size* is provided, **setvbuf** returns a non-zero value. Otherwise, the value returned is zero.

## ***Diagnostics***

---

The **setvbuf** may fail if:

[EBADF] The file descriptor underlying *stream* is not valid.

## ***Notes***

---

A common source of error is allocating buffer space as an “automatic” variable in a code block, and then failing to close the stream in the same block.

The parameter order of the **setvbuf** function is altered by the **-compat** option of the **cc(CP)** command.

## ***See also***

---

**fopen(S)**, **getc(S)**, **malloc(S)**, **putc(S)**, **stdio(S)**

## ***Standards conformance***

---

**setbuf** is conformant with:

AT&T SVID Issue 2;  
X/Open Portability Guide, Issue 3, 1989;  
ANSI X3.159-1989 Programming Language — C;  
Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);  
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1); and NIST FIPS 151-1.

**setvbuf** is conformant with:

AT&T SVID Issue 2;  
X/Open Portability Guide, Issue 3, 1989; and  
ANSI X3.159-1989 Programming Language — C.

# setgroups

set supplementary group ID's

## Syntax

```
cc ... -lc

#include <sys/types.h>

int setgroups(gidsetsize, grouplist)
int gidsetsize;
gid_t *grouplist;
```

## Description

The **setgroups** routine sets the supplementary group access list of the calling process according to the array argument *grouplist*. The argument *gidsetsize* indicates the number of entries in the array and must be no larger than the current value of the constant **NGROUPS**. **NGROUPS** is a tunable kernel parameter.

Only the super user may set supplemental group ID's.

## Return value

Upon successful completion, **setgroups** returns a value of 0. Otherwise, **setgroups** returns a value of -1 and **errno** is set to indicate the appropriate error.

## Diagnostics

If one of the following conditions occurs, **setgroups** fails and **errno** is set to the corresponding value:

- [EFAULT] The address specified for *grouplist* is outside the process address space.
- [EINVAL] The *gidsetsize* parameter is less than zero or greater than **NGROUPS**.
- [EINVAL] An element of *grouplist* is out of range.
- [EPERM] The caller is not the super user.
- [EPERM] The login UID has not yet been set for the calling process.



## ***See also***

---

**getgroups(S), setluid(S), sysconf(S)**

## ***Standards conformance***

---

The **setgroups** routine is an extension of AT&T System V provided by the Santa Cruz Operation.

# setjmp, longjmp

non-local goto

## Syntax

```
cc ... -lc

#include <setjmp.h>

int setjmp (env)
jmp_buf env;

void longjmp (env, val)
jmp_buf env;
int val;
```

## Description

**setjmp** - saves stack environment

**longjmp** - restores last saved environment

These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

**setjmp** saves its stack environment in *env* (whose type, *jmp\_buf*, is defined in the *<setjmp.h>* header file) for later use by **longjmp**. It returns the value 0.

**longjmp** restores the environment saved by the last call of **setjmp** with the corresponding *env* argument. After **longjmp** is completed, program execution continues as if the corresponding call of **setjmp** had just returned the value *val*. **longjmp** cannot cause **setjmp** to return the value 0. If **longjmp** is invoked with a second argument of 0, **setjmp** returns 1. At the time of the second return from **setjmp**, all external and static variables have values as of the time **longjmp** is called (see example). The values of register and automatic variables are undefined.

In a future release, C language users will be able to identify syntactically those automatic variables on whose values they need to rely after the second return from **setjmp**.

## Example

---

```
#include <setjmp.h>

jmp_buf env;
int i = 0;
main ()
{
 void exit();

 if(setjmp(env) != 0) {
 (void) printf("value of i on 2nd return from setjmp: %d\n", i);
 exit(0);
 }

 (void) printf("value of i on 1st return from setjmp: %d\n", i);
 i = 1;
 g();
 /*NOTREACHED*/
}

g()
{
 longjmp(env, 1);
 /*NOTREACHED*/
}
```

If the *a.out* resulting from this C language code is run, the output is:

- value of i on 1st return from setjmp: 0
- value of i on 2nd return from setjmp: 1

## Warning

---

Problems may occur if `longjmp` is called before *env* is primed with a call to `setjmp`.

## Note

---

If signals are to be saved as part of the environment the `sigsetjmp` and `siglongjmp` routines should be used instead.

## See also

---

`signal(S)`, `sigsetjmp(S)`

## ***Standards conformance***

---

**longjmp setjmp** conform with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

ANSI X3.159-1989 Programming Language — C;

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

(S)

# setlocale

---

set or read international environment

## Syntax

---

```
cc ... -lc

#include <locale.h>

char *setlocale(category, locale)
int category;
char *locale;
```

## Description

---

The function **setlocale** sets all or part of the current program's locale (definitions of language and conventions). Once invoked, **setlocale()** immediately updates the information used by the routines that support locale-specific capabilities (for example character and string handling routines, collation, and so on).

The **setlocale()** function accepts two arguments. The *category* argument describes the category of international information to set. The following categories (defined in *<locale.h>*) are recognised:

|                    |                                                                                               |
|--------------------|-----------------------------------------------------------------------------------------------|
| <b>LC_ALL</b>      | Set all categories in the international environment.                                          |
| <b>LC_COLLATE</b>  | Set only the category that contains collating information.                                    |
| <b>LC_CTYPE</b>    | Set only the category that contains character classification and case conversion information. |
| <b>LC_NUMERIC</b>  | Set only the category that contains numeric formatting information.                           |
| <b>LC_MESSAGES</b> | Set only the category that contains message language information.                             |
| <b>LC_MONETARY</b> | Set only the category that contains currency formatting information.                          |
| <b>LC_TIME</b>     | Set only the category that contains date and time formatting information.                     |

The *locale* argument should be a character string that corresponds to the name of a particular international environment. There are three special locale strings, as follows:

|            |                                                                                                                                                                                                                                                                   |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| "C"        | Defines the minimal environment needed for the C programming language.                                                                                                                                                                                            |
| ""         | (Null string) Specifies that the locale is set to be the implementation-defined native environment which may be affected by the <i>/etc/defaults/lang</i> file, the LANG environment variable, or any of the LC_ environment variables. (See <i>environ(M)</i> ). |
| (char *) 0 | (Null pointer) Used to query the current locale for a specific category.                                                                                                                                                                                          |

Other strings use the same format as the user environment variables (see *environ(M)*).

If the program was compiled in default or *-svid* mode, then at program startup (before *main*), the equivalent of:

```
setlocale(LC_ALL, "");
```

is executed, so that the initial locale for the program is as defined in the user environment. If the program was compiled in *-ansi*, *-posix*, or *-xpg3* mode, then the equivalent of the following call is executed at startup:

```
setlocale(LC_ALL, "C");
```

In addition to setting the locale, *setlocale()* may be used to query the current locale. This is done by calling *setlocale()* with a null pointer (that is, (char \*)0) as the locale argument. *setlocale()* returns the current locale associated with this category and the program's locale is not changed.

If *setlocale()* is unable to successfully set the locale for any category, the active locale is unchanged; if the unsuccessful call was during program startup, the effect is as if the previous locale was "C". If the failing call was for the category LC\_ALL, all subcategories which could be set are changed; only those which could not be set remain as before.

## Return value

If successful, *setlocale()* returns the locale name string associated with the locale just set. On error, *setlocale()* returns a null pointer.

When a null pointer is given as the locale, *setlocale()* returns the locale name string of the current locale. If the individual categories differ at all, and the specified category is LC\_ALL, then an expanded locale name string is returned containing the information for each category.

The name string returned by *setlocale()* is such that a subsequent call to *setlocale()* with that locale name string and its associated category restores that part of the program's locale.

When a *locale* is partially defined and does not have a corresponding entry in the file */etc/default/lang*, the following error message is displayed:

```
libc: setlocale: xxxx: no LANG= line in /etc/default/lang
```

where *xxxx* is one of the environment variables found in *environ(M)*. For example, *LC\_CTYPE*.

When a *locale* is inaccurately defined, the following error message is displayed:

```
libc: setlocale: xxxx: cannot open locale file
```

where *xxxx* is one of the environment variables found in *environ(M)*. For example, *LC\_CTYPE*.

## ***Notes***

---

The return values from *setlocale()* are pointers to static data, the content of which is overwritten with each call.

## ***See also***

---

*strcoll(S)*, *ctime(S)*, *ctype(S)*, *environ(M)*, *locale(M)*

## ***Standards conformance***

---

*setlocale()* is conformant with:

ANSI X3.159-1989 Programming Language — C;  
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);  
and NIST FIPS 151-1.

# setluid

set login user ID

## Syntax

```
cc ... -lprot

#include <sys/types.h>
#include <sys/security.h>
#include <sys/audit.h>
#include <prot.h>

int setluid (uid)
unsigned short uid;
```

## Description

The **setluid** routine is used to set the login user ID of the calling process. The login user ID, or LUID, should be set at login time. Only the super user can set the LUID. Once set, the LUID cannot be reset, even by the super user.

Until the LUID is set, the **setuid(S)** and **setgid(S)** routines fail. This ensures that the LUID is set before any identity changes in the other (effective and real) user IDs.

The **setluid** routine is invoked by the **login(C)** program just prior to the identity changes caused by **setuid(S)** and **setgid(S)** calls. It is also used by **at(C)** and **crontab(C)** job entries before starting a non-interactive session for a user.

The LUID is an accurate representation of the user who logged into the system and cannot be altered during the session. The LUID is needed because both the effective and real user IDs can be altered by use of **setuid(S)** or the **setuid** bits on an executable file, and consequently, at times during a session, do not accurately reflect the login user.

The LUID is inherited by all children of the process. If the LUID was not set before a **fork(S)**, the child would also contain an unset LUID.

## Return value

Upon successful completion, the **setluid** routine returns a value of 0. Otherwise, a value of -1 is returned and **errno** is set to indicate the appropriate error.



## ***Diagnostics***

---

If one of the following conditions occurs, the **setluid** routine fails and **errno** is set to the corresponding value:

**[EINVAL]** user ID is out of range.

**[EPERM]** The LUID has already been set for this process or some ancestor of this process.

## ***See also***

---

**getluid(S), getuid(S), setuid(S), setgid(S), stat(S)**

## ***Standards conformance***

---

The **setluid** routine is an extension of AT&T System V provided by the Santa Cruz Operation.

# setpgid

set process group ID for job control

## Syntax

```
cc ... -lc

#include <sys/types.h>

int setpgid (pid, pgid)
pid_t pid, pgid;
```

## Description

If the constant `POSIX_JOB_CONTROL` is defined, the `setpgid` routine is used to either join an existing process group or create a new process group within the session of the calling process. The process group ID of a session leader does not change. Upon successful completion, processes with a process ID that match the argument *pid* have their process group IDs set to the argument *pgid*. As a special case, if *pid* is zero, the process ID of the calling process is used. Also, if *pgid* is zero, the process ID of the indicated process is used.

If `POSIX_JOB_CONTROL` is not defined, the `setpgid` routine may not work as described above and it may fail instead.

## Return value

Upon successful completion, the `setpgid` routine returns a value of zero. If the routine fails, a value of -1 is returned and `errno` is set to indicate the appropriate error.

## Diagnostics

If one of the following conditions occurs, the `setpgid` routine fails and `errno` is set to the corresponding value:

- |          |                                                                                                                                                                                                 |
|----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [EACCES] | The value of the <i>pid</i> argument matches the process ID of a child process of the calling process and the child process has successfully executed one of the <code>exec(S)</code> routines. |
| [EINVAL] | The value of the <i>pgid</i> argument is less than zero or is an unsupported value.                                                                                                             |
| [ENOSYS] | The <code>setpgid</code> routine is not supported by this implementation.                                                                                                                       |

- [EPERM]      The process specified by the *pid* argument is a session leader.  
The value of the *pid* argument is valid but matches the process ID of a child process of the calling process but the child process is not in the same session as the calling process.  
The value of the *pgid* argument does not match the process ID of the process indicated by the *pid* argument and there is no process with a process group ID that matches the value of the *pgid* argument in the same session as the calling process.
- [ESRCH]      The value of the *pid* argument does not match the process ID of the calling process nor of a child process of the calling process.

## *See also*

---

*exec(S)*, *getpid(S)*, *setsid(S)*

## *Standards conformance*

---

**setpgid** conforms with:

X/Open Portability Guide, Issue 3, 1989;  
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);  
and NIST FIPS 151-1.

# setpgrp

---

set process group ID

## Syntax

---

```
cc ... -lc
int setpgrp ()
```

## Description

---

The **setpgrp** routine sets the process group ID of the calling process to the process ID of the calling process and returns the new process group ID.

## Return value

---

The **setpgrp** system call returns the value of the new process group ID. No errors are defined.

## See also

---

**exec(S), fork(S), getpid(S), kill(S), signal(S)**

## Standards conformance

---

**setpgrp** conforms with:  
AT&T SVID Issue 2, but has been withdrawn from XPG3.

# setpriv

set system privileges for this process

---

## Syntax

---

```
cc ... -lprot

#include <sys/types.h>
#include <sys/security.h>
#include <sys/audit.h>

int setpriv (privtype, privs)
int privtype;
priv_t *privs;
```

## Description

---

The **setpriv** routine sets the system privilege vector for the current process to that in the user-supplied *privs* vector. This vector should have at least **SEC\_PRIVVEC\_SIZE** (a system constant) entries. The *privtype* argument may only contain the privilege type **SEC\_EFFECTIVE\_PRIV** (another system constant).

At system initialization, all privileges are included. System privileges are inherited by all children of any process and must call the **setpriv** routines themselves to further restrict system privileges.

The system privilege vector contains per-process privileges used by the TCB. The following system privileges are defined:

### [LABEL\_TERMINAL]

With this privilege, the process can output the string to set or change the terminal label, or otherwise modify the field where the terminal label resides. Without this privilege, the sequence to set the terminal label is intercepted by the system and altered to a harmless (to the label field) sequence.

### [PROMAIN]

Allow a SUID program to access any pathname, subject to the normal discretionary access checking. Without this privilege, a SUID program, after invoking **setuid(S)** to change identity from the program owner to the real user, may only access a pathname (restricted to the real user) in or under the current directory. Path names above the current directory are only accessible if the program owner may access them. Changing the current directory has no effect on this, for the current directory at the time of the SUID program execution (called the *promain root*) is remembered. Previously open files continue to be accessible, no matter how they were opened. Until this privilege was devised, a user had no protection against errant

or malicious SUID programs. The privilege provides a means for the process to restrict the environment used by the SUID program, so that the program owner cannot usurp files owned by the real UID. With this privilege off, the user may run a SUID program with the current directory the root of a subtree that contains no important data therein. Any attempt to access a pathname above the current directory returns an error of [ENOENT]. This mechanism prevents many kinds of Trojan horses from SUID programs, where the SUID program uses the `setuid(S)` call to assign the effective UID to the real UID so that files inaccessible to the prior effective UID become accessible, all done without the knowledge or consent of the session user.

- [SELFAUDIT] The process does its own auditing. The system does not produce audit records for this process.
- [SETID] Allow a program to set the SUID or SGID bits on a file. Turning this privilege off prevents a new user from accidentally propagating his identity. Turning this privilege off and running an untrusted program prevents that program from secretly creating a file owned by you (like a copy of `/bin/sh`) and setting the SUID bit so that it can run as you unrestricted. There are other similar uses.
- [SETOWNER] Allow a program to give a file away (either the user or group). This privilege is needed for a user to execute the System V `chown(S)` call. Without this privilege, a user operates with the `chown` semantics of BSD, where a normal user cannot give a file away.
- [SUID] The process may execute SUID programs. Without this privilege, the process cannot execute any SUID program not set to the same process owner.

## *Return value*

---

Upon successful completion, the `setpriv` routine returns a value of zero. If the routine fails, a value of -1 is returned and `errno` is set to indicate the appropriate error.

## ***Diagnostics***

---

If one of the following conditions occurs, the **setpriv** routine fails and **errno** is set to the corresponding value:

- |                 |                                                                           |
|-----------------|---------------------------------------------------------------------------|
| <b>[EFAULT]</b> | <i>privs</i> points to an invalid address.                                |
| <b>[EPERM]</b>  | <i>privs</i> has more privileges set than what the process has currently. |
| <b>[EINVAL]</b> | <i>privtype</i> is not set to <b>SEC_EFFECTIVE_PRIV</b> .                 |

## ***See also***

---

.() **chdir(S)**, **chroot(S)**, **getpriv(S)**, **audit(S)**

## ***Standards conformance***

---

The **setpriv** routine is an extension of AT&T System V provided by the Santa Cruz Operation.

# setsid

---

create session and set process ID

## Syntax

---

```
cc ... -lc

#include <sys/types.h>
pid_t setsid ()
```

## Description

---

If the calling process is not already a process group leader, the **setsid** routine creates a new session. After the routine is called, the calling process is the session leader of this new session, the process group leader of a new process group, and has no controlling terminal. The process group ID of the calling process is set to the process ID of the calling process. The calling process is the only process in the new process group and in the new session.

## Return value

---

Upon successful completion, the **setsid** routine returns the value of the process group ID of the calling process. If the routine fails, **setsid** returns a value of -1 and **errno** is set to indicate the appropriate error.

## Diagnostics

---

If the following condition occurs, the **setsid** routine fails and **errno** is set to the corresponding value:

[EPERM] The calling process is already a process group leader, or the process group ID of a process other than the calling process matches the process ID of the calling process.

## See also

---

**exec(S)**, **exit(S)**, **fork(S)**, **getpid(S)**, **kill(S)**, **setpgid(S)**, **sigaction(S)**

## Standards conformance

---

**setsid** conforms with:

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);  
and NIST FIPS 151-1.



## setuid, setgid

---

set user and groupIDs

### Syntax

---

```
cc ... -lc

int setuid (uid)
uid_t uid;

int setgid (gid)
gid_t gid;
```

### Description

---

**setuid** - set user IDs

**setgid** - set group IDs

The **setuid** routine is used to set the real user ID and effective user ID of the calling process. If the effective user ID of the calling process is root, the real user ID and effective user ID are set to *uid*. If the effective user ID of the calling process is not root, but its real user ID is equal to *uid*, the effective user ID is set to *uid*. If the effective user ID of the calling process is not root, but the saved set-user ID from **exec(S)** is equal to *uid*, the effective user ID is set to *uid*, except when the saved set-user ID from **exec()** is root.

The **setgid** routine precisely parallels the **setuid** routine except that it sets group IDs rather than user IDs.

The **setgid** routine is used to set the real group ID and effective group ID of the calling process. If the effective user ID of the calling process is root, the real group ID and effective group ID are set to *gid*. If the effective user ID of the calling process is not root, but its real group ID is equal to *gid*, the effective group ID is set to *gid*. If the effective user ID of the calling process is not root, but the saved set-group ID from **exec(S)** is equal to *gid*, the effective group ID is set to *gid*.

### Return value

---

Upon successful completion, **setuid** and **setgid** return a value of 0. If these routines fail, they return a value of -1 and **errno** is set to indicate the appropriate error.

## Diagnostics

---

If one of the following conditions occurs, the **setuid** (**setgid**) routine fails and **errno** is set to the corresponding value:

[EINVAL] The *uid* (*gid*) is out of range.

[EPERM] The real user (group) ID of the calling process is not equal to *uid* (*gid*) and its effective user ID is not root.

## See also

---

getuid(S)

## Standards conformance

---

**setgid** and **setuid** are conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

# shmctl

shared memory control operations

## Syntax

```
cc ... -lc

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl (shmids, cmd, buf)
int shmids, cmd;
struct shmids *buf;
```

## Description

The **shmctl** system call provides a variety of shared memory control operations as specified by *cmd*. The following *cmds* are available:

**IPC\_RMID** Remove the shared memory identifier specified by *shmids* from the system and destroy the shared memory segment and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to that of super user, or to the value of `shm_perm.cuid` or `shm_perm.uid` in the data structure associated with *shmids*.

**IPC\_SET** Set the value of the following members of the data structure associated with *shmids* to the corresponding value found in the structure pointed to by *buf*:

```
shm_perm.uid
shm_perm.gid
shm_perm.mode /* only low 9 bits */
```

This *cmd* can only be executed by a process that has an effective user ID equal to that of super user, or to the value of `shm_perm.cuid` or `shm_perm.uid` in the data structure associated with *shmids*.

**IPC\_STAT** Place the current value of each member of the data structure associated with *shmids* into the structure pointed to by *buf*.

**SHM\_LOCK** Lock the shared memory segment specified by *shmids* in memory. This *cmd* can only be executed by a process that has an effective user ID equal to super user.

**SHM\_UNLOCK** Unlock the shared memory segment specified by *shmids*. This *cmd* can only be executed by a process that has an effective user ID equal to super user.

## Return value

---

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned, and `errno` is set to indicate the error.

## Diagnostics

---

The `shmctl` system call fails if one or more of the following is true:

- [EACCES] `cmd` is equal to `IPC_STAT` and `(READ)` operation permission is denied to the calling process.
- [EFAULT] `buf` points to an illegal address.
- [EINVAL] `shmid` is not a valid shared memory identifier.
- [EINVAL] `cmd` is not a valid command.
- [ENOMEM] `cmd` is equal to `SHM_LOCK`, and there is not enough memory.
- [ENOSYS] The functionality is not supported on this implementation.
- [EPERM] `cmd` is equal to `IPC_RMID` or `IPC_SET`, and the effective user ID of the calling process is not equal to that of super user or to the value of `shm_perm.cuid` or `shm_perm.uid` in the data structure associated with `shmid`.
- [EPERM] `cmd` is equal to `SHM_LOCK` or `SHM_UNLOCK`, and the effective user ID of the calling process is not equal to that of super user.

## Notes

---

The user must explicitly remove shared memory segments after the last reference to them has been removed.

## See also

---

`shmget(S)`, `shmop(S)`

## Standards conformance

---

`shmctl` is conformant with:

AT&T SVID Issue 2;  
and X/Open Portability Guide, Issue 3, 1989.

# shmget

---

get shared memory segment identifier

## Syntax

---

```
cc ... -lc

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget (key, size, shmflg)
key_t key;
int size, shmflg;
```

## Description

---

The **shmget** system call returns the shared memory identifier associated with *key*.

A shared memory identifier and associated data structure and shared memory segment of at least *size* bytes are created for *key* if one of the following is true:

- *key* is equal to **IPC\_PRIVATE**.
- *key* does not already have a shared memory identifier associated with it, and (*shmflg* & **IPC\_CREAT**) is “true”.

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

- **shm\_perm.cuid**, **shm\_perm.uid**, **shm\_perm.cgid**, and **shm\_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order 9 bits of **shm\_perm.mode** are set equal to the low-order 9 bits of *shmflg*. **shm\_segsz** is set equal to the value of *size*.
- **shm\_lpid**, **shm\_nattch**, **shm\_atime**, and **shm\_dtime** are set equal to 0.
- **shm\_ctime** is set equal to the current time.

## Return value

---

Upon successful completion, a non-negative integer, namely a shared memory identifier is returned. Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

## Diagnostics

---

The `shmget` system call fails if one or more of the following is true:

- [EACCES] A shared memory identifier exists for *key*, but operation permission (see `Intro(S)`) as specified by the low-order 9 bits of *shmflg* would not be granted.
- [EEXIST] A shared memory identifier exists for *key* but (*shmflg* & `IPC_CREAT`) and (*shmflg* & `IPC_EXCL`) are "true".
- [EINVAL] *size* is less than the system-imposed minimum or greater than the system-imposed maximum.
- [EINVAL] A shared memory identifier exists for *key*, but the size of the segment associated with it is less than *size*, and *size* is not equal to zero.
- [ENOENT] A shared memory identifier does not exist for *key*, and (*shmflg* & `IPC_CREAT`) is "false".
- [ENOMEM] A shared memory identifier and associated shared memory segment are to be created, but the amount of available memory is not sufficient to fill the request.
- [ENOSPC] A shared memory identifier is to be created, but the system-imposed limit on the maximum number of allowed shared memory identifiers system wide would be exceeded.
- [ENOSYS] The functionality is not supported on this implementation.

## Notes

---

The user must explicitly remove shared memory segments after the last reference to them has been removed.

## See also

---

`shmctl(S)`, `shmop(S)`

## Standards conformance

---

`shmget` is conformant with:

AT&T SVID Issue 2;  
and X/Open Portability Guide, Issue 3, 1989.

# shmop: shmat, shmdt

shared memory operations

---

## Syntax

---

```
cc ... -lc

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

char *shmat (shmid, shmaddr, shmflg)
int shmid;
char *shmaddr;
int shmflg;

int shmdt (shmaddr)
char *shmaddr;
```

## Description

---

**shmat** - attaches shared memory segment

**shmdt** - detaches shared memory segment

The **shmat** system call attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the data segment of the calling process. The segment is attached at the address specified by one of the following criteria:

- If *shmaddr* is equal to zero, the segment is attached at the first available address as selected by the system.
- If *shmaddr* is not equal to zero and (*shmflg* & SHM\_RND) is "true", the segment is attached at the address given by (*shmaddr* - (*shmaddr* modulus SHMLBA)).
- If *shmaddr* is not equal to zero and (*shmflg* & SHM\_RND) is "false", the segment is attached at the address given by *shmaddr*.

**shmdt** detaches from the calling process's data segment the shared memory segment located at the address specified by *shmaddr*.

The segment is attached for reading if (*shmflg* & SHM\_RDONLY) is "true" [READ]; otherwise it is attached for reading and writing [READ/WRITE].

**shmat** fails and does not attach the shared memory segment if one or more of the following is true:

- [EACCES] Operation permission is denied to the calling process (see **Intro(S)**).
- [EINVAL] *shmid* is not a valid shared memory identifier.
- [EINVAL] *shmaddr* is not equal to zero, and the value of (*shmaddr* - (*shmaddr* modulus SHMLBA)) is an illegal address.
- [EINVAL] *shmaddr* is not equal to zero, (*shmflg* & SHM\_RND) is "false", and the value of *shmaddr* is an illegal address.
- [EINVAL] *shmdt* fails and does not detach the shared memory segment if *shmaddr* is not the data segment start address of a shared memory segment.
- [EMFILE] The number of shared memory segments attached to the calling process would exceed the system-imposed limit.
- [ENOMEM] The available data space is not large enough to accommodate the shared memory segment.

## Diagnostics

---

Upon successful completion, the return value is as follows:

- **shmat** returns the data segment start address of the attached shared memory segment.
- **shmdt** returns a value of 0.

Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

## Notes

---

The user must explicitly remove shared memory segments after the last reference to them has been removed.

## See also

---

**exec(S)**, **exit(S)**, **fork(S)**, **shmctl(S)**, **shmget(S)**

## Standards conformance

---

**shmat** and **shmdt** are conformant with:

AT&T SVID Issue 2;  
and X/Open Portability Guide, Issue 3, 1989.



# shutdn

---

flushes block I/O and halts the CPU

## *Syntax*

---

```
cc ... -lx

#include <sys/types.h>
#include <sys/param.h>
#include <sys/filsys.h>

void shutdn (sblk, nsblk, arg);
struct filsys *sblk, *nsblk;
int arg;
```

## *Description*

---

**shutdn** causes all information in memory that should be on disk to be written out. This includes modified super blocks, modified inodes, and delayed block I/O. The super blocks of all writable file systems are flagged 'clean', so that they can be remounted without cleaning when the system is rebooted. **shutdn** then prints Normal System Shutdown on the console and halts the CPU.

The system then stays down or reboots dependent on whether *arg* is 0 or 1.

If *sblk* is greater than 1, it specifies the address of a super block to be written to the root device as the last I/O before the halt, provided that *nsblk* is given as its bit-wise inverse. This facility is provided to allow file system repair programs to supersede the system's copy of the root super block with one of their own.

If *sblk* is 1, the second argument is a command and the third argument is the argument to the command. The **CONFANIC** command, a system configurable system call, is given the argument 0 to stay down, or 1 to reboot. When **shutdn** is called in this way, the purpose is not to bring down the system, but rather, to give instructions to the kernel regarding the way to deal with the next panic.

**shutdown** locks out all other processes while it is doing its work. However, it is recommended that user processes be killed off (see **kill(S)**) before calling **shutdown** as some types of disk activity could cause file systems to not be flagged "clean".

The caller must be the super user.

### *See also*

---

**fsck(ADM)**, **haltsys(ADM)**, **kill(S)**, **mount(S)**, **shutdown(ADM)**

### *Standards conformance*

---

**shutdown** is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

(S)

# sigaction

---

change and/or examine signal action

## Syntax

---

```
cc ... -lc

#include <signal.h>

int sigaction (sig, act, oact)
int sig;
struct sigaction *act, *oact;
```

## Description

---

The routine **sigaction** allows the calling process to examine or specify (or both) the action to be taken upon delivery of the specified signal.

The structure **sigaction**, which describes the action to be taken, is defined in the header file *<signal.h>* to include the following members:

| Member                 | Description                                                                             |
|------------------------|-----------------------------------------------------------------------------------------|
| void (*)( ) sa_handler | SIG_DFL, SIG_IGN, or a pointer to a user-defined, signal-catching function.             |
| sigset_t sa_mask       | Additional set of signals to be blocked during execution of a signal-catching function. |
| int sa_flags           | Special flags to affect the behavior of the signal.                                     |

The *sig* argument can be assigned any one of the following values except SIGKILL and SIGSTOP:

| Signal    | Value             | Description                                 | XPG 3 | POSIX | ANSI |
|-----------|-------------------|---------------------------------------------|-------|-------|------|
| SIGHUP    | 01                | hangup                                      | •     | •     |      |
| SIGINT    | 02                | interrupt                                   | •     | •     | •    |
| SIGQUIT   | 03 <sup>(1)</sup> | quit                                        | •     | •     |      |
| SIGILL    | 04 <sup>(1)</sup> | illegal instruction (not reset when caught) | •     | •     | •    |
| SIGTRAP   | 05 <sup>(1)</sup> | trace trap (not reset when caught)          |       |       |      |
| SIGIOT    | 06 <sup>(1)</sup> | IOT instruction                             |       |       |      |
| SIGABRT   | 06 <sup>(1)</sup> | used by abort, replaces SIGIOT              | •     | •     | •    |
| SIGEMT    | 07 <sup>(1)</sup> | EMT instruction                             |       |       |      |
| SIGFPE    | 08 <sup>(1)</sup> | floating point exception                    | •     | •     | •    |
| SIGKILL   | 09                | kill (cannot be caught or ignored)          | •     | •     |      |
| SIGBUS    | 10 <sup>(1)</sup> | bus error                                   |       |       |      |
| SIGSEGV   | 11 <sup>(1)</sup> | segmentation violation                      | •     | •     | •    |
| SIGSYS    | 12 <sup>(1)</sup> | bad argument to system call                 |       |       |      |
| SIGPIPE   | 13                | write on a pipe with no one to read it      | •     | •     |      |
| SIGALRM   | 14                | alarm clock                                 | •     | •     |      |
| SIGTERM   | 15                | software termination signal                 | •     | •     | •    |
| SIGUSR1   | 16                | user-defined signal 1                       | •     | •     |      |
| SIGUSR2   | 17                | user-defined signal 2                       | •     | •     |      |
| SIGCLD    | 18 <sup>(3)</sup> | death of a child                            |       |       |      |
| SIGCHLD   | 18 <sup>(3)</sup> | synonym for SIGCLD                          |       |       |      |
| SIGPWR    | 19 <sup>(3)</sup> | power fail                                  |       |       |      |
| SIGWINCH  | 20                | window change                               |       |       |      |
| SIGPOLL   | 22 <sup>(4)</sup> | selectable event pending                    |       |       |      |
| SIGSTOP   | 23 <sup>(2)</sup> | sendable stop signal not from tty           | •     | •     |      |
| SIGTSTP   | 24 <sup>(2)</sup> | stop signal from tty                        | •     | •     |      |
| SIGCONT   | 25 <sup>(2)</sup> | continue a stopped process                  | •     | •     |      |
| SIGTTIN   | 26 <sup>(2)</sup> | background tty read attempt                 | •     | •     |      |
| SIGTTOU   | 27 <sup>(2)</sup> | background tty write attempt                | •     | •     |      |
| SIGVTALRM | 28                | virtual timer alarm                         |       |       |      |
| SIGPROF   | 29                | profile alarm                               |       |       |      |

If the argument *act* is not null, it points to a structure specifying the action to be associated with the specified signal. If the argument *oact* is not null, the action previously associated with the signal is stored in the location pointed to by the argument *oact*. If the argument *act* is null, signal handling is unchanged; thus, the call can be used to enquire about the current handling of a given signal.

The field *sa\_handler* of the structure *sigaction* identifies the action to be associated with the specified signal. The *sa\_handler* field is assigned one of the following three values: SIG\_DFL, SIG\_IGN, or an address of a function defined by the user. SIG\_DFL, and SIG\_IGN, are defined in the header file *<signal.h>*. Each is a macro that expands to a constant expression of type pointer to function returning void, and having a unique value that does not match a declarable function.

The actions prescribed by the values of the `sa_handler` are as follows:

**SIG\_DFL** (*execute default signal action*)

Upon receipt of the signal specified by *sig*, the receiving process will take the default action. The default action usually results in the termination of the process with all of the consequences outlined in `exit(S)`. Those signals with a [1] or a [2] are exceptions to this rule. Their default behavior is described in the corresponding Note section.

**SIG\_IGN** (*ignore signal*)

Upon receipt of the signal specified by *sig*, the signal is ignored.

Note: the signals **SIGKILL** and **SIGSTOP** cannot be ignored.

**signal-catching function** (*execute user-defined action*)

Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function pointed to by `sa_handler`. The signal number *sig* is passed as the only argument to the signal-catching function. Additional arguments are passed to the signal-catching function for hardware-generated signals.

Upon return from the signal-catching function, the receiving process resumes execution at the point it was interrupted.

If a signal-catching function is invoked, how it effects the interrupted function is described by each individual function. See the X/Open Portability Guide, Issue 3, 1989 section "Signal Effects on Other Functions", for further information.

The `sigaction` routine does not catch an invalid structure field `sa_handler` and results are undefined when an attempt is made to execute the function at the bad address.

If the `sa_handler` field points to a signal-catching function, the `sa_mask` is used as an additional set of signals that are added to the process's current signal mask (that is, the union of the two signal sets is the result). This new mask is valid before entering and for the duration of the signal-catching function or until a call to either the `sigprocmask` or `sigsuspend` routines. If and when the signal-catching function returns normally, the initial signal mask is restored.

The `sa_flags` field in the structure `sigaction` can be used to alter the behavior of the specified signal.

The **SA\_NOCLDSTOP** flag bit, also defined in the header file `<signal.h>`, can be set in `sa_flags` so that the signal **SIGCHLD** is not generated when children processes stop. If stopped children processes are not of interest, setting this flag bit prevents the invoking of a signal-catching function for these events.

Once an action has been specified for a particular signal, it remains valid until a new course of action has been assigned to the signal with another call to the `sigaction` routine, a call to the `signal` routine, or until one of the `exec` routines is called.

If the previous action for *sig* had been assigned using the **signal** routine, the fields in the structure **sigaction** pointed to by the object *oact* are undefined. However, if a pointer to the same structure or its copy is passed to a subsequent call to **sigaction** using the *act* argument, the signal is handled as if the initial call to **signal** were repeated.

If a call to the **sigaction** routine fails, a new signal handler is not installed.

## Return value

---

Upon successful completion a value of zero is returned. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

## Diagnostics

---

If the following condition occurs, the **sigaction** routine returns -1 and sets **errno** to the corresponding value:

- [EINVAL] The value of the *sig* argument is an invalid or unsupported signal number, or an attempt was made to catch a signal that cannot be caught or to ignore a signal that cannot be ignored.

## Notes

---

- [1] If **SIG\_DFL** is assigned for these signals, in addition to the process being terminated, a "core image" is constructed in the current working directory of the process, if the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named *core* exists and is writable or can be created. If the file must be created, it has the following properties:

- a mode of 0666 modified by the file creation mask (see **umask(S)**).
- a file owner ID that is the same as the effective user ID of the receiving process.
- a file group ID that is the same as the effective group ID of the receiving process.

- [2] For the signals **SIGSTOP**, **SIGTSTP**, **SIGTTIN**, and **SIGTTOU** the default action is to stop the process rather than to terminate the process. The distinction is that a stopped process can be started again with a **SIGCONT** whereas a terminated process cannot be restarted under any circumstance.

For the **SIGCONT** signal, the default action is to continue the process if it is stopped, otherwise it is ignored.

- [3] For the signals **SIGCLD** and **SIGPWR**, the signal-handling function is, again, assigned one of three values: **SIG\_DFL**, **SIG\_IGN**, or a *function address*. The actions prescribed by these values are:

**SIG\_DFL** (*ignore signal*)

The signal is to be ignored.

**SIG\_IGN** (*ignore signal*)

The signal is to be ignored. Also, if *sig* is **SIGCLD**, the calling process's child processes does not create zombie processes when they terminate (see **exit(S)**).

**function address** (*catch signal*)

If the signal is **SIGPWR**, the action to be taken is the same as that described above for *sa\_handler* equal to **function address**. The same is true if the signal is **SIGCLD** with one exception: while the process is executing the signal-catching function, any received **SIGCLD** signals are ignored. (This is the default action.)

In addition, **SIGCLD** affects the **wait**, **waitpid**, and **exit** system calls as follows:

- wait** If the *sa\_handler* for **SIGCLD** is set to **SIG\_IGN** and a **wait** is executed, the **wait** blocks until all of the calling process's child processes terminate; it then returns a value of -1 with **errno** set to **ECHILD**
- waitpid** If the *sa\_handler* for **SIGCLD** is set to **SIG\_IGN** and a **waitpid** is executed, the **waitpid** blocks until all of the calling process's child processes terminate; it then returns a value of -1 with **errno** set to **ECHILD**. However, if the **WNOHANG** option is specified, **waitpid** does not block.
- exit** If in the exiting process's parent process the *sa\_handler* value of **SIGCLD** is set to **SIG\_IGN**, the exiting process does not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the preceding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set **SIGCLD** to be caught.

- [4] **SIGPOLL** is issued when a file descriptor corresponding to a **STREAMS** (see **Intro(S)**) file has a "selectable" event pending. A process must specifically request that this signal be sent using the **I\_SETSIG** **ioctl** call. Otherwise, the process never receives **SIGPOLL**. **SIGPOLL** is defined to have a value of 20 if used in cross-compiling for XENIX.

## ***See also***

---

**ioctl(S), kill(S), signal(S), sigprocmask(S), sigpending(S), sigsuspend(S), sigset(S), sigsetv(S), sigsetjmp(S)**

## ***Standards conformance***

---

**sigaction** is conformant with:

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);  
X/Open Portability Guide, Issue 3, 1989;  
and NIST FIPS 151-1.

(S)



## sigsetjmp, siglongjmp

---

non-local jumps

### Syntax

---

```
#include <setjmp.h>

int sigsetjmp (env, savemask)
sigjmp_buf env;
int savemask;

void siglongjmp (env, val)
sigjmp_buf env;
int val;
```

### Description

---

The **sigsetjmp** macro complies with the definition of the **setjmp** function. The **sigsetjmp** macro deals more proficiently with errors and interrupts encountered in a low-level subroutine of a program.

If the value of the *savemask* argument is not zero, the **sigsetjmp** function also saves the process's current signal mask (see **signal(S)**) as part of the calling environment.

The **siglongjmp** macro complies with the definition of the **longjmp** function. If and only if the *env* argument was initialized by a call to the **sigsetjmp** function with a non-zero *savemask* argument, the **siglongjmp** function restores the saved signal mask.

If the call to **siglongjmp** is in a different function from the corresponding call to **setjmp**, variables with register storage class may have unpredictable values.

### See also

---

**sigaction(S)**, **sigprocmask(S)**, **sigsuspend(S)**, **setjmp(S)**

### Standards conformance

---

**siglongjmp** and **sigsetjmp** are conformant with:

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);  
and NIST FIPS 151-1.

# signal

set a signal action

## Syntax

```
cc ... -lc

#include <signal.h>

void (*signal (sig, func))()

int sig;

void (*func)();
```

## Description

The **signal** routine is not considered to be completely reliable for some applications. Therefore, it is strongly recommended that the **sigaction** routine, which supersedes the **signal** routine, be used in all new applications instead. The **signal** and the **sigaction** routines should never be used in the same application to control the same signal as this results in undefined behavior of the **sigaction** routine. The **signal** routine is still included largely for backward compatibility.

The **signal** routine allows the calling process to choose one of three ways to handle the receipt of a specified signal. The argument **sig** specifies the particular signal and the argument **func** specifies the course of action to be taken.

The **sig** argument can be assigned any one of the following values except **SIGKILL** and **SIGSTOP**:

| Signal    | Value             | Description                                 | XPG 3 | POSIX | ANSI |
|-----------|-------------------|---------------------------------------------|-------|-------|------|
| SIGHUP    | 01                | hangup                                      | •     | •     |      |
| SIGINT    | 02                | interrupt                                   | •     | •     | •    |
| SIGQUIT   | 03 <sup>[1]</sup> | quit                                        | •     | •     |      |
| SIGILL    | 04 <sup>[1]</sup> | illegal instruction (not reset when caught) | •     | •     | •    |
| SIGTRAP   | 05 <sup>[1]</sup> | trace trap (not reset when caught)          |       |       |      |
| SIGIOT    | 06 <sup>[1]</sup> | IOT instruction                             |       |       |      |
| SIGABRT   | 06 <sup>[1]</sup> | used by abort, replaces SIGIOT              | •     | •     | •    |
| SIGEMT    | 07 <sup>[1]</sup> | EMT instruction                             |       |       |      |
| SIGFPE    | 08 <sup>[1]</sup> | floating point exception                    | •     | •     | •    |
| SIGKILL   | 09                | kill (cannot be caught or ignored)          | •     | •     |      |
| SIGBUS    | 10 <sup>[1]</sup> | bus error                                   |       |       |      |
| SIGSEGV   | 11 <sup>[1]</sup> | segmentation violation                      | •     | •     | •    |
| SIGSYS    | 12 <sup>[1]</sup> | bad argument to system call                 |       |       |      |
| SIGPIPE   | 13                | write on a pipe with no one to read it      | •     | •     |      |
| SIGALRM   | 14                | alarm clock                                 | •     | •     |      |
| SIGTERM   | 15                | software termination signal                 | •     | •     | •    |
| SIGUSR1   | 16                | user-defined signal 1                       | •     | •     |      |
| SIGUSR2   | 17                | user-defined signal 2                       | •     | •     |      |
| SIGCLD    | 18 <sup>[3]</sup> | death of a child                            |       |       |      |
| SIGCHLD   | 18 <sup>[3]</sup> | synonym for SIGCLD                          |       |       |      |
| SIGPWR    | 19 <sup>[3]</sup> | power fail                                  |       |       |      |
| SIGWINCH  | 20                | window change                               |       |       |      |
| SIGPOLL   | 22 <sup>[4]</sup> | selectable event pending                    |       |       |      |
| SIGSTOP   | 23 <sup>[2]</sup> | sendable stop signal not from tty           | •     | •     |      |
| SIGTSTP   | 24 <sup>[2]</sup> | stop signal from tty                        | •     | •     |      |
| SIGCONT   | 25 <sup>[2]</sup> | continue a stopped process                  | •     | •     |      |
| SIGTTIN   | 26 <sup>[2]</sup> | background tty read attempt                 | •     | •     |      |
| SIGTTOU   | 27 <sup>[2]</sup> | background tty write attempt                | •     | •     |      |
| SIGVTALRM | 28                | virtual timer alarm                         |       |       |      |
| SIGPROF   | 29                | profile alarm                               |       |       |      |

The **func** argument is assigned one of the following three values: **SIG\_DFL**, **SIG\_IGN**, or an address of a function defined by the user. **SIG\_DFL**, and **SIG\_IGN**, are defined in the header file **<signal.h>**. Each is a macro that expands to a constant expression of type pointer to function returning void, and having a unique value that does not match a declarable function.

The actions prescribed by the values of the **func** argument are as follows:

#### **SIG\_DFL**—execute default signal action

Upon receipt of the signal specified by **sig**, the receiving process will take the default action. The default action usually results in the termination of the process with all of the consequences outlined in **exit(S)**. Those signals with a [1] or a [2] are exceptions to this rule. Their default behavior is described in the corresponding Note section.

**SIG\_IGN** —ignore signal

Upon receipt of the signal specified by **sig**, the signal is ignored.

Note: the **SIGKILL** and **SIGSTOP** signals cannot be ignored.

**signal-catching function** —execute user-defined action

Upon receipt of the signal **sig**, the receiving process executes the signal-catching function pointed to by **func**. The signal number **sig** is passed as the only argument to the signal-catching function. Additional arguments are passed to the signal-catching function for hardware-generated signals. Before entering the signal-catching function, the value of **func** for the caught signal is set to **SIG\_DFL** unless the signal is **SIGILL**, **SIGTRAP**, or **SIGPWR**.

Upon return from the signal-catching function, the receiving process resumes execution at the point it was interrupted.

The signal catching function is not executed during certain slow processes even though a signal has been caught. Slow processes are considered to be a **read(S)**, a **write(S)**, an **open(S)**, or an **ioctl(S)** system call on a slow device (like a terminal; but not a file), a **pause(S)** routine, or during a **wait(S)** routine that does not return immediately due to the existence of a previously stopped or zombie process. Then the interrupted system call may return a -1 to the calling process with **errno** set to **EINTR**.

The **signal** routine does not catch an invalid function argument, **func**, and results are undefined when an attempt is made to execute the function at the bad address.

Note: The **SIGKILL** and **SIGSTOP** signals cannot be caught.

A call to **signal** cancels a pending signal **sig** except for a pending **SIGKILL** or **SIGSTOP** signal.

## Return value

---

Upon successful completion, **signal** returns the previous value of **func** for the specified signal **sig**. Otherwise, a value of **SIG\_ERR** is returned and **errno** is set to indicate the error. **SIG\_ERR** is defined in the include file **<signal.h>**.

## Diagnostics

---

If the following condition occurs, the **signal** routine returns a value of **SIG\_ERR** in **func** and sets **errno** to the corresponding value:

[EINVL] The value of the **sig** argument is an invalid or unsupported signal number, or an attempt was made to catch a signal that cannot be caught or to ignore a signal that cannot be ignored.

## Note

---

- [1] If `SIG_DFL` is assigned for these signals, in addition to the process being terminated, a “core image” is constructed in the current working directory of the process, if the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named *core* exists and is writable or can be created. If the file must be created, it has the following properties:

- a mode of 0666 modified by the file creation mask (see `umask(S)`)
- a file owner ID that is the same as the effective user ID of the receiving process
- a file group ID that is the same as the effective group ID of the receiving process.

- [2] For the signals `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, and `SIGTTOU` the default action is to stop the process rather than to terminate the process. The distinction is that a stopped process can be started again with a `SIGCONT` whereas a terminated process cannot be restarted under any circumstance.

For the `SIGCONT` signal, the default action is to continue the process if it is stopped, otherwise it is ignored.

- [3] For the signals `SIGCLD` and `SIGPWR`, `func` is, again, assigned one of three values: `SIG_DFL`, `SIG_IGN`, or a function address. The actions prescribed by these values are:

`SIG_DFL`—ignore signal

The signal is to be ignored.

`SIG_IGN`—ignore signal

The signal is to be ignored. Also, if `sig` is `SIGCLD`, the calling process’s child processes does not create zombie processes when they terminate (see `exit(S)`).

`function address`—catch signal

If the signal is `SIGPWR`, the action to be taken is the same as that described above for `func` equal to `function address`. The same is true if the signal is `SIGCLD` with one exception: while the process is executing the signal-catching function, any received `SIGCLD` signals are ignored. (This is the default action.)

In addition, SIGCLD affects the **wait**, **waitpid**, and **exit** system calls as follows:

- wait** If the **func** value of SIGCLD is set to SIG\_IGN and a **wait** is executed, the **wait** blocks until all of the calling process's child processes terminate; it then returns a value of -1 with **errno** set to ECHILD.
- waitpid** If the **func** value of SIGCLD is set to SIG\_IGN and a **waitpid** is executed, the **waitpid** blocks until all of the calling process's child processes terminate; it then returns a value of -1 with **errno** set to ECHILD. However, if the WNOHANG option is specified, **waitpid** does not block.
- exit** If in the exiting process's parent process the **func** value of SIGCLD is set to SIG\_IGN, the exiting process does not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the preceding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set SIGCLD to be caught.

- [4] SIGPOLL is issued when a file descriptor corresponding to a STREAMS (see Intro(S)) file has a "selectable" event pending. A process must specifically request that this signal be sent using the L\_SETSIG ioctl call. Otherwise, the process never receives SIGPOLL. SIGPOLL is defined to have a value of 20 if used in cross-compiling for XENIX.

## See also

---

**sigaction(S)**, **sigprocmask(S)**, **sigpending(S)**, **sigsuspend(S)**, **sigset(S)**, **sigsetv(S)**, **sigsetjmp(S)**, **Intro(S)**, **kill(S)**, **pause(S)**, **ptrace(S)**, **wait(S)**, **kill(C)**

## Standards conformance

---

**signal** is conformant with:

AT&T SVID Issue 2;  
X/Open Portability Guide, Issue 3, 1989;  
and ANSI X3.159-1989 Programming Language — C.

# sigpending

---

examine pending signals

## Syntax

---

```
cc ... -lc

#include <signal.h>

int sigpending (set)
sigset_t *set;
```

## Description

---

A call to the **sigpending** routine stores the set of signals blocked from delivery and pending to the calling process in the object to which the argument *set* points.

## Return value

---

Upon successful completion, a value of zero is returned.

**sigpending** fails if the following is true:

**EFAULT**    *set* points outside the allocated address space of the process.  
Where *set* is the name of the documented argument.

## See also

---

**sigaction(S)**, **signal(S)**, **sigprocmask(S)**, **sigsuspend(S)**, **sigset(S)**, **sigsetv(S)**, **sigsetjmp(S)**

## Standards conformance

---

**sigpending** conforms with:

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);  
X/Open Portability Guide, Issue 3, 1989;  
and NIST FIPS 151-1.

# sigprocmask

change and/or examine blocked signals.

## Syntax

```
cc ... -lc

#include <signal.h>

int sigprocmask (how, set, oset)
int how;
sigset_t *set, *oset;
```

## Description

The routine **sigprocmask** changes and/or examines the calling process' signal mask. A process' signal mask lists the signals that are blocked; that is, if a signal in the signal mask is generated, it is not delivered to the process until the signal is unblocked by being removed from the mask.

The signal mask of a process is initialized to that of its parent by **fork(S)**. The mask is changed by the **sigprocmask**, **sigaction(S)**, and **sigsuspend(S)** routines. The contents of a mask are manipulated by the **sigset(S)** routines.

The argument, *how*, dictates how the current signal mask is changed. As defined in the header file, *<signal.h>*, the following options are allowable:

| Name        | Description                                                                                                                                                                               |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SIG_BLOCK   | The resulting set is the union of the current set and the set to which the argument <i>set</i> points (signals in <i>set</i> are added to the existing set).                              |
| SIG_UNBLOCK | The resulting set is the intersection of the current set and the complement of the set to which the argument <i>set</i> points (signals in <i>set</i> are removed from the existing set). |
| SIG_SETMASK | The resulting set is the set to which the argument <i>set</i> points.                                                                                                                     |

If the argument *set* is not NULL, it points to the signal set that will be used to change the current signal mask, depending on the value of the argument *how*. If the argument *set* is NULL, the value of *how* is immaterial and the process's signal mask remains unchanged by the routine. In this latter case, the routine can be used to enquire about currently blocked signals.



If the argument *oset* is not NULL, the previous mask is stored in the object to which the argument *oset* points.

If any pending unblocked signals exist after the call to this routine, at least one of those signals is delivered before the **sigprocmask** routine returns.

Generation of the SIGFPE, SIGILL, or SIGSEGV signals while they are blocked yields an undefined result unless the signal was generated by a call to the **raise(S)** or **kill(S)** routines.

Un-ignorable signals, described in **signal(S)** or **sigaction**, cannot be blocked. The system automatically enforces this restriction without indicating an error.

## ***Return value***

---

Upon successful completion, a value of zero is returned. If **sigprocmask** fails, a value of -1 is returned and **errno** is set to indicate the error. In the latter case, the process's signal mask is not changed.

## ***Diagnostics***

---

If the following condition occurs, the **sigprocmask** function returns -1 and sets **errno** to the corresponding value:

[EINVAL] The value of the *how* argument is not equal to one of the defined values.

## ***See also***

---

**sigaction(S)**, **signal(S)**, **sigpending(S)**, **sigset(S)**, **sigsetjmp(S)** **sigsetv(S)**, **sigsuspend(S)**,

## ***Standards conformance***

---

**sigprocmask** conforms with:

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);  
X/Open Portability Guide, Issue 3, 1989;  
and NIST FIPS 151-1.

# sigsem

signals a process waiting on a semaphore

---

## Syntax

---

```
cc ... -lx

int sigsem(sem_num);
int sem_num;
```

## Description

---

**sigsem** signals a process that is waiting on the semaphore *sem\_num* that it may proceed and use the resource governed by the semaphore. **sigsem** is used in conjunction with **waitsem(S)** to allow synchronization of processes wishing to access a resource. One or more processes may **waitsem** on the given semaphore and is put to sleep until the process which currently has access to the resource issues a **sigsem** call. If there are any waiting processes, **sigsem** causes the process which is next in line on the semaphore's queue to be rescheduled for execution. The semaphore's queue is organized in first in first out (FIFO) order.

## System compatibility

---

**sigsem** can only be used to signal semaphores created under UNIX Version 3.0, not for UNIX System V semaphores.

## Diagnostics

---

**sigsem** returns the value (int) -1 if an error occurs. If *sem\_num* does not refer to a semaphore type file, **errno** is set to ENOTNAM. If *sem\_num* has not been previously opened by **opensem**, **errno** is set to EBADF. If the process issuing a **sigsem** call is not the current "owner" of the semaphore (i.e., if the process has not issued a **waitsem** call before the **sigsem**), **errno** is set to ENAVAIL.

## Notes

---

This feature is a XENIX specific enhancement and may not be present in all UNIX implementations.

*sigsem(S)*

## ***See also***

---

**creatsem(S), opensem(S), waitsem(S)**

## ***Standards conformance***

---

**sigsem** is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

# sigset

manipulate signal sets

## Syntax

```
cc ... -lc

#include <signal.h>

int sigemptyset (set)
sigset_t *set;

int sigfillset (set)
sigset_t *set;

int sigaddset (set, signo)
sigset_t *set;
int signo;

int sigdelset (set, signo)
sigset_t *set;
int signo;

int sigismember (set, signo)
sigset_t *set
int signo;
```

## Description

**sigaddset** - add a signal to the existing signal set  
**sigdelset** - remove a signal from the existing signal set  
**sigemptyset** - initialize the signal set to include no signals  
**sigfillset** - initialize the signal set to include all signals  
**sigismember** - test if a signal is in the existing signal set

These functions manipulate sets of signals. They operate on data objects addressable by the application. They do not operate on any set of signals known to the system, such as the set blocked from delivery to a process or the set pending for a process (see **signal(S)** or **sigaction(S)**.)

The **sigemptyset** routine initializes the signal set to which the argument *set* points, so that all signals are excluded.

The **sigfillset** routine initializes the signal set to which the argument *set* points, so that all signals are included.

Applications should call either the **sigemptyset** or the **sigfillset** routines at least once for each object of type *sigset\_t* before using that object. If an object of type *sigset\_t* is not initialized as described but is used as an argument in any **sigaddset**, **sigdelset**, **sigismember**, **sigaction**, **signal**, **sigprocmask**, **sigpending**, or **sigsuspend** routine, its behavior is undefined.

The **sigaddset** adds and **sigdelset** deletes the individual signal specified by the argument *signo* from the signal set to which the argument *set* points.

The **sigismember** function tests whether the signal specified by the argument *signo* is a member of the signal set to which the argument *set* points.

## *Return value*

---

Upon successful completion, the **sigismember** function returns a value of one if the specified signal is a member of the specified set, or a value of zero if it is not. Upon successful completion, the other functions return a value of zero. For all of the functions, if an error is detected, a value of -1 is returned and **errno** is set to indicate the error.

## *Diagnostics*

---

For the following condition, the **sigaddset**, **sigdelset**, and **sigismember** functions return a -1 and set **errno** to the corresponding value:

[EINVAL] The value of the signal argument is an invalid or unsupported signal number.

## *See also*

---

**sigaction(S)**, **signal(S)**, **sigpending(S)**, **sigprocmask(S)**, **sigsuspend(S)**, **sigsetjmp(S)** **sigsetv(S)**,

## *Standards conformance*

---

**sigset** conforms with:  
AT&T SVID Issue 2; and  
X/Open Portability Guide, Issue 3, 1989.

# sigset, sighold, sigrelse, sigignore, sigpause

signal management routines

---

## Syntax

---

```
cc ... -lc

#include <signal.h>

void (*sigset (sig, func))()
int sig;
void (*func)();

int sighold (sig)
int sig;

int sigrelse (sig)
int sig;

int sigignore (sig)
int sig;

int sigpause (sig)
int sig;
```

## Description

---

**sigset** - specifies signal action to be taken  
**sighold** - holds a signal until released or discarded  
**sigrelse** - release a held signal  
**sigignore** - sets the action for signal to SIG\_IGN  
**sigpause** - suspends the calling process until it receives a signal

These functions provide signal management for application processes. The **sigset** system call specifies the system signal action to be taken upon receipt of signal *sig*. This action is either calling a process signal-catching handler *func* or performing a system-defined action.

*sig* can be assigned any one of the following values except SIGKILL and SIGSTOP. Machine- or implementation-dependent signals are not included (see *Notes* below). Each value of *sig* is a macro, defined in *<signal.h>*, that expands to an integer constant expression.

| Signal    | Value             | Description                                 | XPG 3 | POSIX | ANSI |
|-----------|-------------------|---------------------------------------------|-------|-------|------|
| SIGHUP    | 01                | hangup                                      | •     | •     |      |
| SIGINT    | 02                | interrupt                                   | •     |       | •    |
| SIGQUIT   | 03 <sup>[1]</sup> | quit                                        | •     | •     |      |
| SIGILL    | 04 <sup>[1]</sup> | illegal instruction (not reset when caught) | •     | •     | •    |
| SIGTRAP   | 05 <sup>[1]</sup> | trace trap (not reset when caught)          |       |       |      |
| SIGIOT    | 06 <sup>[1]</sup> | IOT instruction                             |       |       |      |
| SIGABRT   | 06 <sup>[1]</sup> | used by abort, replaces SIGIOT              | •     | •     | •    |
| SIGEMT    | 07 <sup>[1]</sup> | EMT instruction                             |       |       |      |
| SIGFPE    | 08 <sup>[1]</sup> | floating point exception                    | •     | •     | •    |
| SIGKILL   | 09                | kill (cannot be caught or ignored)          | •     | •     |      |
| SIGBUS    | 10 <sup>[1]</sup> | bus error                                   |       |       |      |
| SIGSEGV   | 11 <sup>[1]</sup> | segmentation violation                      | •     | •     | •    |
| SIGSYS    | 12 <sup>[1]</sup> | bad argument to system call                 |       |       |      |
| SIGPIPE   | 13                | write on a pipe with no one to read it      | •     | •     |      |
| SIGALRM   | 14                | alarm clock                                 | •     | •     |      |
| SIGTERM   | 15                | software termination signal                 | •     | •     | •    |
| SIGUSR1   | 16                | user-defined signal 1                       | •     | •     |      |
| SIGUSR2   | 17                | user-defined signal 2                       | •     | •     |      |
| SIGCLD    | 18 <sup>[3]</sup> | death of a child                            |       |       |      |
| SIGCHLD   | 18 <sup>[3]</sup> | synonym for SIGCLD                          |       |       |      |
| SIGPWR    | 19 <sup>[3]</sup> | power fail                                  |       |       |      |
| SIGWINCH  | 20                | window change                               |       |       |      |
| SIGPOLL   | 22 <sup>[4]</sup> | selectable event pending                    |       |       |      |
| SIGSTOP   | 23 <sup>[2]</sup> | sendable stop signal not from tty           | •     | •     |      |
| SIGTSTP   | 24 <sup>[2]</sup> | stop signal from tty                        | •     | •     |      |
| SIGCONT   | 25 <sup>[2]</sup> | continue a stopped process                  | •     | •     |      |
| SIGTTIN   | 26 <sup>[2]</sup> | background tty read attempt                 | •     | •     |      |
| SIGTTOU   | 27 <sup>[2]</sup> | background tty write attempt                | •     | •     |      |
| SIGVTALRM | 28                | virtual timer alarm                         |       |       |      |
| SIGPROF   | 29                | profile alarm                               |       |       |      |

The following values for the system-defined actions of *func* are also defined in *<signal.h>*. Each is a macro that expands to a constant expression of type pointer to function returning *void* and has a unique value that matches no declarable function.

**SIG\_DFL** (*default system action*)

Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit(S)*. Those signals with a [1] or a [2] are exceptions to this rule. Their default behavior is described in the corresponding Notes section below.

**SIG\_IGN** (*ignore signal*)

Any pending signal *sig* is discarded and the system signal action is set to ignore future occurrences of this signal type.

**SIG\_HOLD** (*hold signal*)

The signal *sig* is to be held upon receipt. Any pending signal of this type remains held. Only one signal of each type is held.

Otherwise, *func* must be a pointer to a function, the signal-catching handler, that is to be called when signal *sig* occurs. In this case, *sigset* specifies that the process calls this function upon receipt of signal *sig*. Any pending signal of this type is released. This handler address is retained across calls to the other signal management functions listed here.

When a signal occurs, the signal number *sig* is passed as the only argument to the signal-catching handler. Before calling the signal-catching handler, the system signal action is set to **SIG\_HOLD**. During normal return from the signal-catching handler, the system signal action is restored to *func* and any held signal of this type released. If a non-local goto (*longjmp*) is taken, then the signal remains held, and *sigrelse* must be called to restore the system signal action and release any held signal of this type.

If a signal-catching function is invoked, how it effects the interrupted function is described by each individual function. See the X/Open Portability Guide, Issue 3, 1989 *sigaction(S)* section "Signal Effects on Other Functions", for further information.

*sighold* and *sigrelse* are used to establish critical regions of code. *sighold* is analogous to raising the priority level and deferring or holding a signal until the priority is lowered by *sigrelse*. *sigrelse* restores the system signal action to that specified previously by *sigset*.

*sigignore* sets the action for signal *sig* to **SIG\_IGN** (see above).



**sigpause** suspends the calling process until it receives a signal, the same as **pause(S)**. However, if the signal *sig* had been received and held, it is released and the system signal action taken. This system call is useful for testing variables that are changed on the occurrence of a signal. The correct usage is to use **sighold** to block the signal first, then test the variables. If they have not changed, then call **sigpause** to wait for the signal. **sigset** fails if one or more of the following is true:

[EINVAL] *sig* is an illegal signal number (including SIGKILL) or the default handling of *sig* cannot be changed.

[EINTR] A signal was caught during the system call **sigpause**.

## Diagnostics

---

Upon successful completion, **sigset** returns the previous value of the system signal action for the specified signal *sig*. Otherwise, a value of SIG\_ERR is returned and **errno** is set to indicate the error. SIG\_ERR is defined in *<signal.h>*.

For the other functions, upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

## Notes

---

- [1] If SIG\_DFL is assigned for these signals, in addition to the process being terminated, a "core image" is constructed in the current working directory of the process, if the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named *core* exists and is writable or can be created. If the file must be created, it has the following properties:

- a mode of 0666 modified by the file creation mask (see **umask(S)**).
- a file owner ID that is the same as the effective user ID of the receiving process.
- a file group ID that is the same as the effective group ID of the receiving process.

- [2] For the signals SIGSTOP, SIGTSTP, SIGTTIN, and SIGTTOU the default action is to stop the process rather than to terminate the process. The distinction is that a stopped process can be started again with a SIGCONT whereas a terminated process cannot be restarted under any circumstance.

For the SIGCONT signal, the default action is to continue the process if it is stopped, otherwise it is ignored.

- [3] For the signals SIGCLD and SIGPWR, the signal-handling function is, again, assigned one of three values: SIG\_DFL, SIG\_IGN, or a *function address*. The actions prescribed by these values are:

**SIG\_DFL** (*ignore signal*)

The signal is to be ignored.

**SIG\_IGN** (*ignore signal*)

The signal is to be ignored. Also, if *sig* is SIGCLD, the calling process's child processes does not create zombie processes when they terminate (see exit(S)).

**function address** (*catch signal*)

If the signal is SIGPWR, the action to be taken is the same as that described above for func equal to **function address**. The same is true if the signal is SIGCLD with one exception: while the process is executing the signal-catching function, any received SIGCLD signals are ignored. (This is the default action.)

In addition, SIGCLD affects the wait, waitpid, and exit system calls as follows:

- wait** If the func for SIGCLD is set to SIG\_IGN and a wait is executed, the wait blocks until all of the calling process's child processes terminate; it then returns a value of -1 with **errno** set to ECHILD
- waitpid** If the func for SIGCLD is set to SIG\_IGN and a waitpid is executed, the waitpid blocks until all of the calling process's child processes terminate; it then returns a value of -1 with **errno** set to ECHILD. However, if the WNOHANG option is specified, waitpid does not block.
- exit** If in the exiting process's parent process the func value of SIGCLD is set to SIG\_IGN, the exiting process does not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the preceding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set SIGCLD to be caught.

- [4] SIGPOLL is issued when a file descriptor corresponding to a STREAMS (see Intro(S)) file has a "selectable" event pending. A process must specifically request that this signal be sent using the I\_SETSIG ioctl call. Otherwise, the process never receives SIGPOLL. SIGPOLL is defined to have a value of 20 if used in cross-compiling for XENIX.

For portability, applications should use only the symbolic names of signals rather than their values and use only the set of signals defined here. The action for the signal SIGKILL cannot be changed from the default system action.

Other implementations of UNIX System V may have other implementation-defined signals. Also, additional implementation-defined arguments may be passed to the signal-catching handler for hardware-generated signals. For certain hardware-generated signals, it may not be possible to resume execution at the point of interruption.

The signal type SIGSEGV is reserved for the condition that occurs on an invalid access to a data object.

The other signal management functions, **signal(S)** and **pause(S)**, should not be used in conjunction with these routines for a particular signal type.

### ***See also***

---

**kill(S), pause(S), signal(S), setjmp(S), wait(S)**

### ***Standards conformance***

---

**sigsetv** is conformant with:

Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

# sigsuspend

wait for signal(s)

## Syntax

```
cc ... -lc

#include <signal.h>

int sigsuspend (sigmask)
sigset_t *sigmask;
```

## Description

A call to the **sigsuspend** routine replaces the process's signal mask with the signal set to which the argument *sigmask* points. The routine then suspends the process until it receives a signal whose associated action is to either terminate the process or execute a signal-catching function.

If the action is to execute a signal-catching function, the **sigsuspend** routine returns immediately after the signal-catching function does. The signal mask is restored to the value it held before the call to the **sigsuspend** routine. If the action calls for the termination of the process, the **sigsuspend** routine does not return.

As with all signal routines, un-ignorable signals cannot be blocked (see **signal(S)**. or **sigaction(S)**). The system enforces this restriction without indicating an error.

## Return value

Because the **sigsuspend** function suspends process execution indefinitely, no successful completion value is returned. A value of -1 is returned and **errno** is set to indicate the error.

## Diagnostics

If the following condition occurs, the **sigsuspend** function returns -1 and sets **errno** to the corresponding value:

[EINTR] A signal is caught by the calling process and control is returned from the signal-catching function.

## ***See also***

---

**pause(S), sigaction(S), signal(S), sigpending(S), sigprocmask(S), sigset(S), sigsetjmp(S) sigsetv(S),**

## ***Standards conformance***

---

**sigsuspend** is conformant with:

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);  
X/Open Portability Guide, Issue 3, 1989;  
and NIST FIPS 151-1.

# sinh, cosh, tanh

hyperbolic functions

## Syntax

```
cc ... -lm

#include <math.h>

double sinh (x)
double x;

double cosh (x)
double x;

double tanh (x)
double x;
```

## Description

**sinh** - returns hyperbolic sine of argument

**cosh** - returns hyperbolic cosine of argument

**tanh** - returns hyperbolic tangent of argument

The **sinh**, **cosh**, and **tanh** functions return, respectively, the hyperbolic sine, cosine and tangent of their argument.

## Return value

If the program was compiled in the default or **-svid** mode **sinh** and **cosh** functions return **HUGE** (and **sinh** may return **-HUGE** for negative *x*) when the correct value would overflow and set **errno** to **ERANGE**. If the program was compiled in **-ansi**, **-posix**, or **-xpg3** mode then **HUGE\_VAL** or **-HUGE\_VAL** are returned.

## ***Diagnostics***

---

The **sinh** function may fail if:

[EDOM]            The value of *x* is NaN.

[ERANGE]        The result would cause overflow.

## ***Notes***

---

These error-handling procedures may be changed with the function **matherr(S)**.

## ***See also***

---

**matherr(S)**

## ***Standards conformance***

---

**cosh**, **sinh** and **tanh** are conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

ANSI X3.159-1989 Programming Language — C;

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

# sleep

suspend execution for interval

## Syntax

```
cc ... -lc

unsigned sleep (seconds)
unsigned seconds;
```

## Description

The current process is suspended from execution for the number of *seconds* specified by the argument. The actual suspension time may be less than that requested for two reasons: (1) Because scheduled wakeups occur at fixed 1-second intervals, (on the second, according to an internal clock) and (2) because any caught signal terminates the *sleep* following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount due to the scheduling of other activity in the system. The value returned by *sleep* is the "unslept" amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested *sleep* time, or premature arousal due to another caught signal.

The routine is implemented by setting an alarm signal and pausing until it (or some other signal) occurs. The previous state of the alarm signal is saved and restored. The calling program may have set up an alarm signal before calling *sleep*. If the *sleep* time exceeds the time till such alarm signal, the process sleeps only until the alarm signal would have occurred. The caller's alarm catch routine is executed just before the *sleep* routine returns. But if the *sleep* time is less than the time till such alarm, the prior alarm time is reset to go off at the same time it would have without the intervening *sleep*.

## See also

*alarm(S)*, *pause(S)*, *signal(S)*

## Standards conformance

*sleep* is conformant with:

AT&T SVID Issue 2;  
X/Open Portability Guide, Issue 3, 1989;  
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);  
and NIST FIPS 151-1.



## **sputl, sgetl**

access long integer data in a machine-independent fashion

---

### ***Syntax***

---

```
cc ... -lld

void sputl (value, buffer)
long value;
char *buffer;

long sgetl (buffer)
char *buffer;
```

### ***Description***

---

**sputl** - puts long integer data in memory

**sgetl** - gets long integer data from memory

**sputl** takes the four bytes of the long integer *value* and places them in memory starting at the address pointed to by *buffer*. The ordering of the bytes is the same across all machines.

**sgetl** retrieves the four bytes in memory starting at the address pointed to by *buffer* and returns the long integer value in the byte ordering of the host machine.

The combination of **sputl** and **sgetl** provides a machine-independent way of storing long numeric data in a file in binary form without conversion to characters.

### ***Standards conformance***

---

**sgetl** and **sputl** are conformant with:

AT&T SVID Issue 2.

# ssignal, gsignal

software signals

## Syntax

```
cc ... -lc

#include <signal.h>

int (*ssignal (sig, action))()
int sig, (*action)();

int gsignal (sig)
int sig;
```

## Description

The **ssignal** and **gsignal** functions implement a software facility similar to **signal(S)**. This facility enables users to indicate the disposition of error conditions.

Software signals made available to users are associated with integers in the inclusive range 1 through 16. A call to **ssignal** associates a procedure, *action*, with the software signal *sig*; the software signal, *sig*, is raised by a call to **gsignal**. Raising a software signal causes the action established for that signal to be taken.

The first argument to **ssignal** is a number identifying the type of signal for which an action is to be established. The second argument defines the action; it is either the name of a (user-defined) action function or one of the manifest constants **SIG\_DFL** (default) or **SIG\_IGN** (ignore). The **ssignal** function returns the action previously established for that signal type; if no action has been established or the signal number is illegal, **ssignal** returns **SIG\_DFL**.

The **gsignal** function raises the signal identified by its argument, *sig*:

- If an action function has been established for *sig*, then that action is reset to **SIG\_DFL** and the action function is entered with argument *sig*. **gsignal** returns the value returned to it by the action function.
- If the action for *sig* is **SIG\_IGN**, **gsignal** returns the value 1 and takes no other action.
- If the action for *sig* is **SIG\_DFL**, **gsignal** returns the value 0 and takes no other action.

If *sig* has an illegal value or no action was ever specified for *sig*, **gsignal** returns the value 0 and takes no other action.

*ssignal(S)*

## ***See also***

---

**signal(S), sigset(S)**

## ***Standards conformance***

---

**gsignal** and **ssignal** are conformant with:

AT&T SVID Issue 2; however, both routines have been withdrawn from XPG3.

# stat, fstat, lstat, statlstat

returns file status

## Syntax

```
cc ... -lc

#include <sys/types.h>
#include <sys/stat.h>

int stat (path, buf)
char *path;
struct stat *buf;

int fstat (fildes, buf)
int fildes;
struct stat *buf;

int lstat (path, buf)
char *path;
struct stat *buf;

int statlstat (path, buf)
char *path;
struct stat *buf;

S_ISBLK (fmode)

S_ISCHR (fmode)

S_ISDIR (fmode)

S_ISFIFO (fmode)

S_ISNAM (fmode)

S_ISREG (fmode)
```

## Description

The **stat** system call obtains information about the named file. Both the **stat** and **fstat** functions update any time-related fields before writing into the **stat** structure.

**fstat** obtains information about an open file known by the file descriptor *fildes*, obtained from a successful **open**, **creat**, **dup**, **fcntl**, or **pipe** system call.

**lstat** obtains information about a symbolic link. Use the macro **S\_ISLNK** to determine if the file is a symbolic link.

Calls to **lstat** or **symlink** made on a kernel on which these calls are not supported, such as SCO UNIX System V Version 2.0, produce the signal SIGSYS. Programs can be made to operate correctly on kernels that do and do not support these routines by first calling **lstat(S)** and then calling **stat(S)** if SIGSYS is produced by the first call.

*path* points to a path name naming a file. Read, write, or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable. An implementation that provides extended security controls may, under implementation-defined conditions, cause the **stat** function to fail. In particular, the system may deny the existence of the file specified by *path*.

The **lstat** system call is supported by SCO UNIX System V Version 4.0, but not by earlier versions of SCO UNIX. A SIGSYS signal is sent to a program attempting an **lstat** system call on earlier versions, which typically results in a core dump. If you are developing a program to use **lstat**, and that program is also required to run on systems which do not support **lstat**, then you should use **statlstat** in place of **lstat**. On its first call to **lstat**, **statlstat** uses **signal(S)** to catch SIGSYS without core dumping, and uses **stat** instead and thereafter, if the system does not support **lstat**.

The **S\_IS\*** macros provide POSIX file testing capabilities. The *fmode* argument is the file mode. If a test is correct, a nonzero value is returned; otherwise, a value of zero is returned. The purpose of the update any time-related fields before writing into the macros is:

| Macro           | Purpose                                           |
|-----------------|---------------------------------------------------|
| <b>S_ISBLK</b>  | determine if file is a block special file         |
| <b>S_ISCHR</b>  | determine if file is a character special file     |
| <b>S_ISDIR</b>  | determine if file is a directory                  |
| <b>S_ISFIFO</b> | determine if file is a first-in, first-out (FIFO) |
| <b>S_ISNAM</b>  | determine if file is a special named file         |
| <b>S_ISREG</b>  | determine if file is a regular file               |

*buf* is a pointer to a **stat** structure into which information is placed concerning the file.

The contents of the structure pointed to by *buf* include the following members:

```

dev_t st_dev; /* ID of device containing */
 /* a directory entry for this file */
ino_t st_ino; /* Inode number */
mode_t st_mode; /* File mode (see mknod (S)) */
nlink_t st_nlink; /* Number of links */
uid_t st_uid; /* User ID of the file's owner */
gid_t st_gid; /* Group ID of the file's group */
dev_t st_rdev; /* ID of device */
 /* This entry is defined only for */
 /* character special or */
 /* block special files */
off_t st_size; /* File size in bytes */
time_t st_atime; /* Time of last access */
time_t st_mtime; /* Time of last data modification */
time_t st_ctime; /* Time of last file status change */
 /* Times measured in seconds since */
 /* 00:00:00 GMT, Jan. 1, 1970 */

```

- st\_mode** The mode of the file as described in the **mknod(S)** system call.
- st\_ino** This field uniquely identifies the file in a given file system. The pair **st\_ino** and **st\_dev** uniquely identifies regular files.
- st\_dev** This field uniquely identifies the file system that contains the file. Its value may be used as input to the **ustat(S)** system call to determine more information about this file system. No other meaning is associated with this value.
- st\_rdev** This field should be used only by administrative commands. It is valid only for block special or character special files and only has meaning on the system where the file was configured.
- st\_nlink** This field should be used only by administrative commands.
- st\_uid** The user ID of the file's owner.
- st\_gid** The group ID of the file's group.
- st\_size** For regular files, this is the address of the end of the file. For pipes or fifos, this is the count of the data currently in the file. For block special or character special, this is not defined.
- st\_atime** Time when file data was last accessed. Changed by the following system calls: **creat(S)**, **mknod(S)**, **pipe(S)**, **utime(S)**, and **read(S)**.
- st\_mtime** Time when data was last modified. Changed by the following system calls: **creat(S)**, **link(S)**, **mknod(S)**, **pipe(S)**, **unlink(S)**, **utime(S)**, and **write(S)**.
- st\_ctime** Time when file status was last changed. Changed by the following system calls: **chmod(S)**, **chown(S)**, **creat(S)**, **link(S)**, **mknod(S)**, **pipe(S)**, **unlink(S)**, **utime(S)**, and **write(S)**.

The **stat** system call fails if one or more of the following is true:

**[EACCES]** Search permission is denied for a component of the path prefix.

**[ENAMETOOLONG]**

The length of the *path* argument exceeds **[PATH\_MAX]** or a pathname component is longer than **[NAME\_MAX]** while **[\_POSIX\_NO\_TRUNC]** is in effect.

**[ENOENT]** The named file does not exist.

**[ENOTDIR]** A component of the path prefix is not a directory.

**fstat** fails if:

**[EBADF]** *fildes* is not a valid open file descriptor.

## *Diagnostics*

---

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

## *See also*

---

**chmod(S), chown(S), creat(S), link(S), mknod(S), pipe(S), read(S), readlink(S), symlink(S), time(S), unlink(S), utime(S), write(S)**

## *Standards conformance*

---

**fstat** and **stat** are conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1); and NIST FIPS 151-1.

**lstat** and **statlstat** are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

# statfs, fstatfs

get file system information

## Syntax

```
cc ... -lc

#include <sys/types.h>
#include <sys/statfs.h>

int statfs (path, buf, len, fstyp)
char *path;
struct statfs *buf;
int len, fstyp;

int fstatfs (fildes, buf, len, fstyp)
int fildes;
struct statfs *buf;
int len, fstyp;
```

## Description

**statfs** - get file system information

**fstatfs** - get file system information

The **statfs** system call returns a “generic superblock” describing a file system. It can be used to acquire information about mounted as well as unmounted file systems, and usage is slightly different in the two cases. In all cases, *buf* is a pointer to a structure (described below) which is filled by the system call, and *len* is the number of bytes of information which the system should return in the structure. *len* must be no greater than `sizeof (struct statfs)` and ordinarily it contains exactly that value; if it holds a smaller value, the system fills the structure with that number of bytes. (This allows future versions of the system to grow the structure without invalidating older binary programs.)

If the file system of interest is currently mounted, *path* should name a file which resides on that file system. In this case the file system type is known to the operating system and the *fstyp* argument must be zero. For an unmounted file system *path* must name the block special file containing it and *fstyp* must contain the (non-zero) file system type. In both cases read, write, or execute permission of the named file is not required, but all directories listed in the *path* name leading to the file must be searchable.



The **statfs** structure pointed to by *buf* includes the following members:

```

short f_fstyp; /* File system type */
long f_bsize; /* Block size */
long f_frsize; /* Fragment size */
long f_blocks; /* Total number of blocks */
long f_bfree; /* Count of free blocks */
long f_files; /* Total number of file nodes */
long f_ffree; /* Count of free file nodes */
char f_fname[6]; /* Volume name */
char f_fpack[6]; /* Pack name */

```

The **fstatfs** system call is similar, except that the file named by *path* in **statfs** is instead identified by an open file descriptor *fildes* obtained from a successful **open(S)**, **creat(S)**, **dup(S)**, **fcntl(S)**, or **pipe(S)** system call.

The **statfs** system call obsoletes **ustat(S)** and should be used in preference to it in new programs.

## Return value

---

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

## Diagnostics

---

The **statfs** and **fstatfs** system calls fails if one or more of the following is true:

- [EACCES]      Search permission is denied for a component of the path prefix.
- [EBADF]      *fildes* is not a valid open file descriptor.
- [EFAULT]      *buf* or *path* points to an invalid address.
- [EINVAL]      *fstyp* is an invalid file system type; *path* is not a block special file and *fstyp* is nonzero; *len* is negative or is greater than **sizeof (struct statfs)**.
- [EMULTIHOP]   Components of *path* require hopping to multiple remote machines.
- [ENOENT]      The named file does not exist.
- [ENOLINK]      *path* points to a remote machine, and the link to that machine is no longer active.
- [ENOTDIR]      A component of the path prefix is not a directory.

## *See also*

---

**chmod(S), chown(S), creat(S), filesystem(FP), link(S), mknod(S), pipe(S), read(S), time(S), unlink(S), utime(S), write(S)**

## *Standards conformance*

---

**statfs** and **fstatfs** are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

# stdio

standard buffered input/output package

## Syntax

```
#include <stdio.h>

FILE *stdin, *stdout, *stderr;
```

## Description

The functions described in this manual entry constitute an efficient, user-level I/O buffering scheme. The in-line macros `getc(S)` and `putc(S)` handle characters quickly. The macros `getchar` and `putchar`, and the higher-level routines `fgetc`, `fgets`, `fprintf`, `fputc`, `fputs`, `fread`, `fscanf`, `fwrite`, `gets`, `getw`, `printf`, `puts`, `putw`, and `scanf` all use or act as if they use `getc` and `putc`; they can be freely intermixed.

If the program is compiled in the default mode, the following are defined only as macros: `getc`, `getchar`, `putc`, `putchar`, `error`, `feof`, `clearerr`, and `fileno`. If the program is compiled in `-ansi`, `-posix`, or `-xpg3` mode, they exist as macros and functions. The function form can be used if the macro form is `#undef`ed.

A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type `FILE`. The `fopen(S)` function creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the `<stdio.h>` header file and associated with the standard open files:

`stdin` - standard input file  
`stdout` - standard output file  
`stderr` - standard error file

A constant `NULL` (0) designates a nonexistent pointer.

An integer-constant `EOF` (-1) is returned upon end-of-file or error by most integer functions that deal with streams (see the individual descriptions for details).

An integer constant `BUFSIZ` specifies the size of the buffers used by the particular implementation.

Any program that uses this package must include the header file of pertinent macro definitions, as follows:

```
#include <stdio.h>
```

The constants and the following “functions” are implemented as macros (redeclaration of these names is perilous): **getc**, **getchar**, **putc**, **putchar**, **ferror**, **feof**, **clearerr**, and **fileno**.

Output streams, with the exception of the standard error stream *stderr*, are by default buffered if the output refers to a file, and line-buffered if the output refers to a terminal. The standard error output stream *stderr* is by default unbuffered, but use of **freopen** (see **fopen(S)**) cause it to become buffered or line-buffered. When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as written. When it is buffered, many characters are saved up and written as a block. When it is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new-line character is written or terminal input is requested). The **setbuf** or **setvbuf** functions in **setbuf(S)** may be used to change the stream’s buffering strategy.

The default setting for the open file limit (**FOPEN\_MAX**) is 128. For code compiled to be iBCS2 conformant, the default is 60. The existing shared C library is iBCS2 conformant and, thus, is restricted to an open file limit of 60. The static archive (libc.a) is restricted to 128.

## Diagnostics

---

Invalid *stream* pointers usually cause disorders, possibly including program termination. Individual function descriptions describe the possible error conditions.

## See also

---

**close(S)**, **ctermid(S)**, **cuserid(S)**, **fclose(S)**, **ferror(S)**, **fopen(S)**, **fread(S)**, **fseek(S)**, **getc(S)**, **gets(S)**, **lseek(S)**, **open(S)**, **pipe(S)**, **popen(S)**, **printf(S)**, **putc(S)**, **puts(S)**, **read(S)**, **scanf(S)**, **setbuf(S)**, **system(S)**, **tmpfile(S)**, **tmpnam(S)**, **ungetc(S)** **write(S)**,

## Standards conformance

---

**stdio** is conformant with:

ANSI X3.159-1989 Programming Language — C;  
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);  
AT&T SVID Issue 2;  
and X/Open Portability Guide, Issue 3, 1989.

# stime

---

set time

## Syntax

---

```
cc ... -lc

int stime (tp)
long *tp;
```

## Description

---

The **stime** system call sets the system's idea of the time and date. *tp* points to the value of time as measured in seconds from 00:00:00 GMT January 1, 1970.

[EPERM] **stime** fails if the effective user ID of the calling process is not super-user.

## Diagnostics

---

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

## See also

---

**ctime(S)**, **difftime(S)**, **mktime(S)**, **nl\_cxtime(S)**, **time(S)**

## Standards conformance

---

**stime** is conformant with:

AT&T SVID Issue 2; however, this routine has been withdrawn from XPG3.

# stopio

stop further I/O to an open file

## Syntax

```
cc ... -lprot

#include <sys/types.h>
#include <sys/security.h>
#include <sys/audit.h>
#include <prot.h>

int stopio (path)
char *path;
```

## Description

The **stopio** routine prevents further I/O on all file descriptors now having the argument *path* open. The next time a **read(S)**, **write(S)** or **ioctl(S)** is invoked by any process on any file descriptor of *path*, the system call fails with the [EBADF] error and the **SIGHUP** signal is sent to the process. **stopio** may only be invoked by the super user or owner of file *path*.

The **stopio** routine is used to isolate successive user sessions on a single terminal. At the time **stopio** is invoked with the terminal *path* as the argument, background processes left by previous sessions may continue to run on the terminal, but any further input or output to the terminal fails as previously described. As the terminal gets re-opened after the **stopio** invocation, I/O using those new file descriptors succeeds until the next **stopio** call on the same *path*. This system call is intended to be used immediately before the open of the terminal line.

## Return value

Upon successful completion, the **stopio** routine returns a value of 0. Otherwise, a value of -1 is returned and **errno** is set to indicate the appropriate error.

## Diagnostics

If one of the following conditions occurs, the **stopio** routine fails and **errno** is set to the corresponding value:

- [EACCES] Search permission is denied on a component of the path prefix.
- [EFAULT] Path points outside the allocated address space of the process.
- [ENOENT] The named file does not exist.

- [ENOTDIR] A component of the path prefix is not a directory.
- [ENOTTY] *path* is not a local character special file. (This may go away - see following *Note*).
- [EPERM] The effective user ID does not match the owner of the file and the effective user ID is not super-user.
- [EROFS] The named file resides on a read-only file system. (This may go away - see following *Note*).

## *Note*

---

The [EROFS] error is not really appropriate for this routine.

**stopio** should work for all file types, but a security study has not yet been made as to the utility or feasibility in such an assignment. For now, it works only for character special files to specifically address the problem of sharing a single terminal through multiple login sessions.

**stopio**'s functionality should really be a command within **fcntl(S)**, but **fcntl** expects the file to be open already and this call only requires a path name.

## *See also*

---

**creat(S)**, **ioctl(S)**, **open(S)**, **read(S)**, **write(S)**

## *Standards conformance*

---

The **stopio** routine is an extension of AT&T System V provided by the Santa Cruz Operation.

# strcoll, strncoll, strnxfrm, strxfrm

handles collation of strings

## Syntax

```
cc ... -lintl

int strcoll (str1, str2)
char *str1, *str2;

int strncoll (str1, str2, n)
char *str1, *str2;
int n;

int strnxfrm (to, from, maxsize, n)
char *to, *from;
int maxsize, n;

int strxfrm (to, from, maxsize)
char *to, *from;
int maxsize;
```

## Description

These functions operate on null-terminated strings.

The **strxfrm** routine transforms the string *from* according to the collation environment of the current locale. The result is placed in the character array *to*. The maximum size of the resulting string is the value contained in the *maxsize* variable. The result can then be compared with another transformed string to establish the collating order of the two original strings. The **strnxfrm** routine transforms at most *n* characters in the *from* string, where *n* is defined after the following collation rules have been applied:

- 1 → 2 character transformations : for example, the german 'ß' counts as two characters.
- 2 → 1 character transformations : for example, the spanish 'ch' counts as a single character.
- 1 → 1 character transformations : for example, 'a', 'b', etc. count as single characters.
- 1 → 0 character transformations : for example, the hyphen, '-', is not counted.

Both functions return the size of the array required to hold the transformed string. If the return value is greater than *maxsize* then the contents of *to* is undefined.



The **strcoll** function is used to collate the two strings *str1* and *str2* according to the collation environment of the current locale. The returned value is equal to, less than or greater than 0, according to whether *str1* is equal to, less than or greater than *str2*. **strncoll** collates the two strings until the *n*th character in *str1* is reached. The *n*th character is defined in the same way as **strnxfm**.

## See also

---

**coltbl(M), nl\_strcmp(S), string(S)**

## Standards conformance

---

**strcoll** and **strxfm** are conformant with:  
ANSI X3.159-1989 Programming Language — C.

## Examples

---

It is assumed that the value of *maxsize* is large enough to contain the transformed string in the **strxfm** and the **strnxfm** examples.

### Example 1:

This is an example of the **strxfm** routine:

```
char *to1, *to2;
int ret;

strxfm(to1, "Straße", maxsize);
strxfm(to2, "Strasse", maxsize);
ret= strcmp(to1, to2)
```

**ret** contains 0 because "Straße" and "Strasse" collate as equal.

### Example 2:

This is an example of the **strcoll** routine:

```
int ret;

ret = strcoll("Straße", "Strasse");
```

**ret** contains 0 because "Straße" and "Strasse" collate as equal.

**Example 3:**

This is an example of the **strnxfm** routine:

```
char *to;

strnxfm(to, "Straße", maxsize, 6);
```

This transforms only the "Straß" portion of the string, as ß counts as two characters.

**Example 4:**

This is an example of the **strncoll** routine:

```
strncoll("Straße", "Bahn", 6);
```

This compares only the "Straß" portion of the string, as ß counts as two characters.

## **strerror**

---

gets error message pointer from last routine call error

### ***Syntax***

---

```
cc ... -lc

#include <string.h>

char *strerror(errnum)
int errnum;
```

### ***Description***

---

The **strerror** routine maps *errnum* to an error-message string, returning a pointer to the string. The function itself does not actually print the message; for that, you need to call an output function such as **printf**.

### ***Return value***

---

The **strerror** function returns a pointer to the error-message string. The string can be overwritten by subsequent calls to **strerror**.

### ***See also***

---

**clearerr(S)**, **ferror(S)**, **perror(S)**

### ***Standards conformance***

---

**strerror** is conformant with: ANSI X3.159-1989 Programming Language — C.

## Example

---

```

#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

extern int errno;
int errnum;
int fh1, fh2;

main()
{
 errnum=0;
 if ((fh1=open("xxxx",O_RDONLY)) == -1)
 errnum=errno;
 fh2=open("yyyy",O_RDONLY);

 /* Other code that may set the errno value.*/
 if (errnum != 0)
 printf(strerror(errnum));
}

```

The program shown above tries to open files *xxxx* and *yyyy*. If an error occurs opening *xxxx*, the variable *errnum* is set to the *errno* value returned by *open*. Other code that may alter the *errno* value is then executed. Later, the saved *errno* value in *errnum* is checked and, if nonzero, an error message assigned to it by *strerror* is printed. If file *xxxx* does not exist, the example prints the following message:

No such file or directory

# strftime

format date/time string

---

## Syntax

---

```
cc ... -lintl

#include <time.h>

int strftime(s, maxsize, format, timeptr)
char *s;
int maxsize;
char *format;
struct tm *timeptr;
```

## Description

---

The **strftime** function places its output (according to the format string *format* and the time values contained in the structure pointed to by *timeptr*), followed by the null character (\0) in consecutive bytes starting at *s*. The maximum length of the output string is specified by *maxsize*, the maximum number of characters including the terminating null character.

The **strftime** function converts and formats the time values contained in the structure pointed to by *timeptr* under control of the format string *format*. The string *format* is a character string can contain two types of object. These are: plain characters, which are simply copied to the output string, and directives.

Each directive is introduced by the percent character (%). The following directives are independent of the program's current locale (see *locale(M)*):

- %d** is replaced by the day of the month as a decimal number (01-31)
- %H** is replaced by the hour (24-hour clock) as a decimal number (00-23)
- %I** is replaced by the hour (12-hour clock) as a decimal number (01-12)
- %j** is replaced by the day of the year as a decimal number (001-366)
- %m** is replaced by the month as a decimal number (01-12)
- %M** is replaced by the minute as a decimal number (00-59)
- %S** is replaced by the second as a decimal number (00-59)
- %U** is replaced by the week number of the year (Sunday as the first day of the week) as a decimal number (00-52)
- %w** is replaced by the weekday as a decimal number [0(Sunday)-6]
- %W** is replaced by the week number of the year (Monday as the first day of the week) as a decimal number (00-52)

- %y** is replaced by the year without century as a decimal number (00-99)
- %Y** is replaced by the year with century as a decimal number
- %Z** is replaced by the timezone name, or by no characters if no timezone exists
- %%** is replaced by %

The strings used to replace the following directives are dependent on the program's current locale, and are defined using the utility `timtbl(M)`:

- %a** is replaced by the locale's abbreviated weekday name
- %A** is replaced by the locale's full weekday name
- %b** is replaced by the locale's abbreviated month name
- %B** is replaced by the locale's full month name
- %c** is replaced by the locale's appropriate date and time representation
- %p** is replaced by the locale's equivalent of AM or PM
- %x** is replaced by the locale's appropriate date representation
- %X** is replaced by the locale's appropriate time representation

## Return value

---

The function returns the number of characters placed in the string *s* (not including the terminating null character), or zero if the length of the output string would exceed *maxsize*. In the latter case, the content of the output string *s* is undefined.

Zero is also returned if the function detects a situation which would lead to infinite recursion; for example, if a format string refers to itself.

## See also

---

`ctime(S),`  
`nl_cxtime(S), timtbl(M)`

## Standards conformance

---

**strftime** is conformant with:

ANSI X3.159-1989 Programming Language — C;  
 IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);  
 and NIST FIPS 151-1.

## **string: strcat, strchr, strcmp, strcpy, strcspn, strdup, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strstr, strtok**

---

string operations

### ***Syntax***

---

```
cc ... -lc

#include <string.h>
#include <sys/types.h>

char *strcat (s1, s2)
char *s1, *s2;

char *strchr (s, c)
char *s;
int c;

int strcmp (s1, s2)
char *s1, *s2;

char *strcpy (s1, s2)
char *s1, *s2;

size_t strcspn (s1, s2)
char *s1, *s2;

char *strdup (s1)
char *s1;

size_t strlen (s)
char *s;

char *strncat (s1, s2, n)
char *s1, *s2;
size_t n;

int strncmp (s1, s2, n)
char *s1, *s2;
size_t n;

char *strncpy (s1, s2, n)
char *s1, *s2;
size_t n;

char *strpbrk (s1, s2)
char *s1, *s2;
```

```

char *strrchr (s, c)
char *s;
int c;

size_t strspn (s1, s2)
char *s1, *s2;

char *strstr(s1, s2)
char *s1, *s2;

char *strtok (s1, s2)
char *s1, *s2;

```

## ***SVID Syntax***

```

int strcspn (s1, s2)
char *s1, *s2;

int strlen (s)
char *s;

int strspn (s1, s2)
char *s1, *s2;

```

## ***Description***

---

The arguments *s1*, *s2*, and *s* point to strings (arrays of characters terminated by a null character). The functions **strcat**, **strncat**, **strcpy**, and **strncpy** all alter *s1*. These functions do not check for overflow of the array pointed to by *s1*.

**strcat** appends a copy of string *s2* to the end of string *s1*. If unsuccessful completion occurs **strcat** returns a null pointer. **strncat** appends at most *n* characters. Each returns a pointer to the null-terminated result. The initial character of *s2* overwrites the null character at the end of *s1*.

**strcmp** compares its arguments and returns an integer less than, equal to, or greater than 0, according as *s1* is lexicographically less than, equal to, or greater than *s2*. **strncmp** makes the same comparison but looks at most *n* characters.

**strcpy** copies string *s2* to *s1*, stopping after the null character has been copied. **strncpy** copies exactly *n* characters, truncating *s2* or adding null characters to *s1* if necessary. The result is not null-terminated if the length of *s2* is *n* or more. Each function returns *s1*, or null on an unsuccessful completion. If the array pointed to by *s2* is a string shorter than *n* characters, null characters are appended to the copy in the array pointed to by *s1*, until *n* characters in all have been written.

**strchr** returns a pointer to the first occurrence of character *c* in string *s*, or a null pointer if *c* does not occur in the string. **strrchr** returns a pointer to the last occurrence of character *c* in string *s*, or a null pointer if *c* does not occur in the string. In both routines, the null character terminating a string is considered to be part of the string.



**strdup** returns a pointer to a new string that is a duplicate of the string pointed to by *s1*. The space for the new string is obtained using **malloc(S)**. If the new string cannot be created, null is returned.

**strlen** returns the number of characters in *s*, not including the terminating null character.

**strpbrk** returns a pointer to the first occurrence in string *s1* of any character from string *s2*, or a null pointer if no character from *s2* exists in *s1*.

**strspn** and **strcspn** return the length of the initial segment of string *s1* that consists entirely of characters from (or not from in the case of **strcspn**) string *s2*.

**strstr** finds the first occurrence in string *s1* of string *s2*. If *s2* has zero length, a pointer to *s1* is returned. If *s2* is not found, a null pointer is returned. Otherwise, a pointer to the located substring in *s1* is returned.

**strtok** considers the string *s1* to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *s2*. The first call (with pointer *s1* specified) returns a pointer to the first character of the first token, and has written a null character into *s1* immediately following the returned token. The function keeps track of its position in the string between separate calls, so that subsequent calls (which must be made with the first argument a null pointer) work through the string *s1* immediately following that token. In this way subsequent calls work through the string *s1* until no tokens remain. The separator string *s2* may be different from call to call. When no token remains in *s1*, a null pointer is returned.

For user convenience, all these functions are declared in the optional *<string.h>* header file.

## Notes

---

**strcmp** and **strncmp** are implemented by using the most natural character comparison on the machine. Thus the sign of the value returned when one of the characters has its high-order bit set is not the same in all implementations and should not be relied upon. For XPG3 and ANSI, the sign of a nonzero value returned by **strcmp** or **strncmp** is determined by the sign of the difference between the values of the first pair of characters that differ in the objects being compared. The first pair of characters are interpreted as *unsigned char*.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

## See also

---

**malloc(S)**

## Standards conformance

---

**strcat, strchr, strcmp, strcpy, strcspn, strlen, strncat, strncmp, strpbrk, strspn** and **strtok** are conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

ANSI X3.159-1989 Programming Language — C;

Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1); and NIST FIPS 151-1.

**strdup** is conformant with:

AT&T SVID Issue 2.

**strstr** is conformant with:

X/Open Portability Guide, Issue 3, 1989.

**strncpy** and **strrchr** are conformant with:

X/Open Portability Guide, Issue 3, 1989;

ANSI X3.159-1989 Programming Language — C;

Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1); and NIST FIPS 151-1.

## strtod

---

convert string to double-precision number

### Syntax

---

```
cc ... -lc

#include <stdlib.h>

double strtod (str, ptr)
char *str, **ptr;
```

### Description

---

The **strtod** function returns as a double-precision floating-point number the value represented by the character string pointed to by *str*. The string is scanned up to the first unrecognized character.

The **strtod** function recognizes an optional string of “white-space” characters (as defined by *isspace* in *ctype(S)*), then an optional sign, then a string of digits optionally containing a decimal point, then an optional e or E followed by an optional sign or space, followed by an integer.

If the value of *ptr* is not (char \*\*)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no number can be formed, *\*ptr* is set to *str*, and zero is returned. If the correct value would cause overflow, **±HUGE\_VAL** (as defined in *<math.h>*) is returned (according to the sign of the value), and **errno** is set to **ERANGE**.

If the correct value would cause underflow, zero is returned and **errno** is also set to **ERANGE**.

### See also

---

**atof(S)**, **ctype(S)**, **scanf(S)**, **strtoul(S)**

### Standards conformance

---

**strtod** is conformant with:

AT&T SVID Issue 2;  
X/Open Portability Guide, Issue 3, 1989;  
and ANSI X3.159-1989 Programming Language — C.

# strtol

convert string to integer

## Syntax

```
cc ... -lc

#include <stdlib.h>
#include <limits.h> /* ANSI-only */

long strtol (str, ptr, base)
char *str, **ptr;
int base;
```

## Description

The **strtol** function returns as a long integer the value represented by the character string pointed to by *str*. The string is scanned up to the first character inconsistent with the base. Leading "white-space" characters (as defined by *isspace* in *ctype(S)*) are ignored.

If the value of *ptr* is not (char \*\*)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no integer can be formed, that location is set to *str*, and zero is returned.

If *base* is positive (and not greater than 36), it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and "0x" or "0X" is ignored if *base* is 16.

If *base* is zero, the string itself determines the base. After an optional leading sign a leading zero indicates octal conversion, and a leading "0x" or "0X" hexadecimal conversion. Otherwise, decimal conversion is used.

Truncation from long to int can take place upon assignment or by an explicit cast.

Upon successful completion, **strtol** returns the converted value, if any. If the correct value causes overflow, and you are compiling with the **cc -ansi** flag, **errno** is set to **ERANGE** and **LONG\_MIN** is returned if the value is negative, and **LONG\_MAX** is returned if the value is positive. (**LONG\_MIN** and **LONG\_MAX** are defined in *limits.h*.) For all other conformance standards, the value returned is either a positive or negative long integer and **ERANGE** is returned in **errno**. For all other errors, zero is returned and **errno** indicates the error. **EINVAL** is returned when the value of *base* is not supported.

## ***See also***

---

**atof(S), ctype(S), scanf(S), strtod(S)**

## ***Standards conformance***

---

**strtol** is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

and ANSI X3.159-1989 Programming Language — C.

# strtoul

convert a string to an unsigned long integer

## Syntax

```
cc ... -l
#include <stdlib.h>

unsigned long strtoul (const char *sptr, char **eptr, int base)
```

## Description

The **strtoul** routine converts a string to an unsigned long integer. *sptr* points to the start of the string, *eptr* points to the end of the string, and *base* defines the input radix, for example, base 8 is octal. *base* can be in the range of a value of zero to 36.

If *base* is a value of zero, then the string is assumed to contain the key for the *base*. The string can start with plus (+) or minus (-), or without a sign. The next character, or the first if a sign is not specified, specifies the input radix. If zero (0), then decimal is assumed. If the character is either lowercase "x" or "X", then hexadecimal is assumed; otherwise, octal is assumed.

## Return values

On success, an unsigned long integer value for the string is returned; otherwise, on overflow, **errno** is set to **ERANGE** and the value of the **ULONG\_MAX** constant is returned.

## See also

**strtod(S)**, **strtol(S)**

## Standards conformance

**strtoul** is an SCO extension. It is conformant with:

X/Open Portability Guide, Issue 3, 1989;  
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);  
and ANSI X3.159-1989 Programming Language — C.

# subsystems

---

manipulation routines for Subsystems database

## *Syntax*

---

```
cc ... -lprot -lx

int authorized_user (auth)
char *auth;

int primary_auth (name)
char *name;

char *primary_of_secondary_auth (name)
char *name;

int secondary_auth (name)
char *name;

int total_auths()

int widest_auth()

int write_authorizations (user, auth_list, list_len)
char *user;
char **auth_list;
int list_len;
```

## *Description*

---

These routines manipulate and refer to the Subsystems database, which is extracted from the Protected Password entries for all users. The Subsystems database stores primary and secondary authorizations for users with respect to protected subsystems on the Security Module Package. A protected subsystem is associated with a special group, and provides some service to users. It is called protected because all its programs run SGID to that group, and the files and devices that the subsystem references are only accessible to that group, and therefore through the protected subsystem's programs.

A *primary authorization* for a protected subsystem allows a user to assume the administrator role for that subsystem. The primary authorization name is the group name whose effective identity is used by the subsystem programs to protect subsystem files. If a user possesses a primary authorization for a subsystem, he/she can do all actions within that subsystem. The list of primary authorizations is as follows:

|          |                                                                          |
|----------|--------------------------------------------------------------------------|
| audit    | Audit administrator.                                                     |
| auth     | Authentication database administrator.                                   |
| backup   | Backup and file system maintenance administrator.                        |
| cron     | Cron subsystem administrator (at, batch, crontab).                       |
| lp       | Line printer spooling subsystem administrator.                           |
| mem      | Authorization to view information about other users (memory devices).    |
| sysadmin | System administrator functions that require root privilege.              |
| terminal | The authorization to send unfiltered information between user terminals. |
| uucp     | The ability to run uucp (not currently supported).                       |

A *secondary authorization* allows finer grain operations within protected subsystems. These authorizations are often granted either to specific users or to the entire user community by appropriate setup of user Protected Password entries and the Defaults database. Each secondary authorization is associated with exactly one protected subsystem, and only allows operations with respect to that subsystem. The secondary authorizations are as follows:

|             |                                                                                                   |
|-------------|---------------------------------------------------------------------------------------------------|
| printqueue  | Authorization to see other user's jobs in the print queues (lp).                                  |
| printerstat | Authorization to change printer status with <b>enable(C)</b> and <b>disable(C)</b> commands (lp). |
| queryspace  | Authorization to use <b>df(C)</b> to query file system space (backup).                            |

The **authorized\_user** routine returns non-zero if the login user ID associated with the current process has the specified authorization in the subsystem that is currently running. This subsystem is identified by the effective group under which the program began execution. The primary authorization for any subsystem grants all secondary authorizations for that subsystem.

The rest of the routines are for programs that manipulate user entries directly, or that have to validate authorizations input by users. They may not be supported in future releases of the SMP. The **primary\_auth** routine maps a primary authorization name to a bit offset in a mask. This is used for checking a user's authorizations against the Protected Password entry. The **primary\_of\_secondary\_auth** routine returns the primary authorization name associated with the secondary authorization. The primary authorization name is also the subsystem name in which the secondary authorization is recognized. The **secondary\_auth** routine maps a secondary authorization to a



bit offset. The **total\_auths** routine returns the number of primary and secondary authorizations recognized by the system. The **widest\_auth** routine returns the longest string name of any authorization.

The **write\_authorizations** routine updates the Subsystems database for a given user. It takes *list\_len* authorizations from the *auth\_list* array of string pointers and associates those authorizations with the given user. If the first character string pointer references the string "default," the user is given default authorizations. This routine is used by the screen-oriented routine which updates the user's Protected Password entry, and then must propagate the user's authorizations to the Subsystems database.

## *Diagnostics*

---

The **authorized\_user** routine returns non-zero if the user possesses the specified authorization, otherwise zero. The **total\_auths** routine returns the sum of the number of primary authorizations and the number of secondary authorizations recognized by the system. The **widest\_auth** routine returns the length of the longest string name of an authorization. This length does not include the trailing null character. Both **primary\_auth** and **secondary\_auth** return a negative value if the authorization name is not supported; otherwise, they return the bit offset of the authorization. The Protected Password database stores both types of authorizations in the same mask. The **primary\_of\_secondary\_auth** routine returns a pointer to a static area containing the primary authorization name associated with the secondary authorization. The string must be copied if it is to be modified. The **write\_authorizations** routine returns 0 on success, non-zero on permission failures, I/O errors, etc.

## *See also*

---

**getprpwent(S)**

## *Standards conformance*

---

**subsystems** is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

# swab

---

swap bytes

## Syntax

---

```
cc ... -lc

void swab (src, dest, nbytes)
char *src, *dest;
int nbytes;
```

## Description

---

The **swab** function copies *nbytes* bytes pointed to by *src* to the object pointed to by *dest*, exchanging adjacent bytes. The *nbytes* argument should be even and non-negative. If *nbytes* is odd and positive, **swab** copies and exchanges *nbytes* - 1 and the disposition for the last byte is undefined. If *nbytes* is negative, **swab** does nothing.

## Standards conformance

---

**swab** is conformant with:

AT&T SVID Issue 2;  
and X/Open Portability Guide, Issue 3, 1989.

# symlink

creates symbolic link to a file

## Syntax

---

**cc ... -lc**

```
int symlink(path, newpath)
char *path, *newpath;
```

## Description

---

The **symlink** function creates a symbolic link. The argument *path* is the name and location of the object to which the symbolic link points. The argument *newpath* is the name and location of the symbolic link being created.

The object, *path*, that the symbolic link, *newpath*, points to, must be specified either as an absolute path, or as a relative path leading from the directory containing the symbolic link to the object *path*. *path* and *newpath* need not be on the same file systems. The object *path* need not exist for the symbolic link to be created. Neither *path* nor *newpath* can be a null string.

Symbolic links are used only to find objects, therefore they can point to any object supported by the filesystem, including, but not limited to, regular files, directories, block, character and FIFO special files and other symbolic links.

A call to **stat(S)** with the symbolic link *newpath* returns information about the file, *path*, as if **stat(S)** had been called on *path* directly. A call to **lstat(S)** returns information about the symbolic link, *newpath*. Both of these routines are described on the **stat(S)** manual page.

Calls to **symlink(S)** or **lstat(S)** made on a kernel on which these calls are not supported, such as SCO UNIX System V Release 3.2 Version 2.0, produce the signal SIGSYS. Programs can be made to operate correctly on kernels that do and do not support these routines by first calling **lstat(S)** and then calling **stat(S)** if SIGSYS is produced by the first call.

**readlink(S)** can be used to read the contents of a symbolic link.

An error is returned if any of the following is true:

- |          |                                                                                     |
|----------|-------------------------------------------------------------------------------------|
| [EACCES] | Search permission denied for a component of the path prefix of <i>newpath</i> .     |
| [EEXIST] | The file specified in <i>newpath</i> already exists.                                |
| [EFAULT] | <i>path</i> or <i>newpath</i> points outside the process's allocated address space. |

**[EINVAL]**

1. The filesystem on which *newpath* is created does not support symbolic links
2. The *path* argument is a null string

**[ELOOP]**

Too many symbolic links were encountered in translating the *newpath* file argument. The maximum number of symbolic links, that can be found in a pathname, is defined by `_POSIX_SYMLINK` in the `<sys/unistd.h>` header file. This value is compiled into the kernel so changing this value does not affect the limit unless the kernel is recompiled.

**[ENAMETOOLONG]**

1. The length of a component of *newpath* exceeds the allowable length of a pathname component on the filesystem on which it is to be created, for example, 255 characters on EAFS and ES51K filesystems.
2. The length of either *path* or *newpath* exceeds 1024 characters.

**[ENOENT]**

A component of the path prefix of the *newpath* file argument does not exist.

**[ENOSPC]**

1. The directory in which the entry for the new symbolic link is being placed cannot be extended because no space is left on the filesystem containing the directory.
2. The new symbolic link cannot be created because no space is left on the filesystem containing the link.
3. No free inodes are available on the filesystem containing the link.

**[ENOTDIR]**

A component of the path prefix of the *newpath* file argument is not a directory.

**[EROFS]**

The file specified in *newpath* would reside on a read-only filesystem.

## ***Return value***

---

Upon successful completion, zero (0) is returned. If an error occurs, an error is returned in `errno` and -1 is returned to the calling routine.

## ***See also***

---

`link(S)`, `ln(C)`, `lstat(S)`, `readlink(S)`, `stat(S)`, `unlink(S)`

## ***Standards conformance***

---

`symlink` is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

# sync

---

update super block

## *Syntax*

---

```
cc ... -lc
```

```
void sync ()
```

## *Description*

---

The `sync` system call causes all information in memory that should be on disk to be written out. This includes modified super blocks, modified inodes, and delayed block I/O.

It should be used by programs which examine a filesystem, for example, `fsck`, `df`, and so on. It is mandatory before a reboot.

The writing, although scheduled, is not necessarily complete upon return from `sync`.

## *Standards conformance*

---

`sync` is conformant with:

AT&T SVID Issue 2; however, this routine has been withdrawn from XPG3.

# sysconf

get configurable system variables

## Syntax

```
cc ... -lc

#include <unistd.h>
long sysconf (name)
int name ;
```

## Description

The `sysconf` function provides a method for the application to determine the current value of a configurable system limit or option (*variable*).

The *name* argument represents the system variable to be queried. In the table that follows, the values in the left column are defined in `<limits.h>`, except for `CLK_TCK`, which is defined in `<time.h>`. The values in the right column are defined in `<unistd.h>`. The information in the right column comprise the possible values for the *name* argument.

| Variable                          | <i>name</i> Value              |
|-----------------------------------|--------------------------------|
| <code>{ARG_MAX}</code>            | <code>{_SC_ARG_MAX}</code>     |
| <code>{CHILD_MAX}</code>          | <code>{_SC_CHILD_MAX}</code>   |
| <code>{CLK_TCK}</code>            | <code>{_SC_CLK_TCK}</code>     |
| <code>{NGROUPS_MAX}</code>        | <code>{_SC_NGROUPS_MAX}</code> |
| <code>{OPEN_MAX}</code>           | <code>{_SC_OPEN_MAX}</code>    |
| <code>{PASS_MAX}</code>           | <code>{_SC_PASS_MAX}</code>    |
| <code>{_POSIX_JOB_CONTROL}</code> | <code>{_SC_JOB_CONTROL}</code> |
| <code>{_POSIX_SAVED_IDS}</code>   | <code>{_SC_SAVED_IDS}</code>   |
| <code>{_POSIX_VERSION}</code>     | <code>{_SC_VERSION}</code>     |

The value returned by `sysconf` for `_SC_CLK_TCK` is the same as `CLK_TCK`.

## Return value

If *name* is an invalid value, `sysconf` returns a -1. If the variable corresponding to *name* is not defined on the system, `sysconf` returns -1 without changing the value of `errno`.

Otherwise, `sysconf` returns the current variable value on the system. The value returned is not more restrictive than the corresponding value described to the application when it was compiled with the implementation's `<time.h>` or `<unistd.h>`. The value does not change during the lifetime of the calling process.

## Diagnostics

---

If any of the following conditions occur, the `sysconf` function returns -1 and sets `errno` to the corresponding value:

[EINVAL] The value of the name argument is invalid.

## Notes

---

PASS\_MAX and \_SC\_PASS\_MAX are XPG3 extensions over POSIX.

## Standards conformance

---

`sysconf` is conformant with:

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);  
and NIST FIPS 151-1.



# sysfs

get file system type information

## Syntax

```
cc ... -lc

#include <sys/fstyp.h>
#include <sys/fsid.h>

int sysfs (opcode, fsname)
int opcode;
char *fsname;

int sysfs (opcode, fs_index, buf)
int opcode;
int fs_index;
char *buf;

int sysfs (opcode)
int opcode;
```

## Description

The **sysfs** system call returns information about the file system types configured in the system. The number of arguments accepted by **sysfs** varies and depends on the *opcode*. The currently recognized *opcodes* and their functions are described below:

|                   |                                                                                                                                                                                                                                                              |
|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>GETFSIND</b>   | translates <i>fsname</i> , a null-terminated file-system identifier, into a file-system type index.                                                                                                                                                          |
| <b>GETFSTYP</b>   | translates <i>fs_index</i> , a file-system type index, into a null-terminated file-system identifier and writes it into the buffer pointed to by <i>buf</i> ; this buffer must be at least of size <b>FSTYPSZ</b> as defined in <i>&lt;sys/fstyp.h&gt;</i> . |
| <b>GETNFSSTYP</b> | returns the total number of file system types configured in the system.                                                                                                                                                                                      |

The **sysfs** system call fails if one or more of the following is true:

|                 |                                                                                                                           |
|-----------------|---------------------------------------------------------------------------------------------------------------------------|
| <b>[EFAULT]</b> | <i>buf</i> or <i>fsname</i> point to an invalid user address.                                                             |
| <b>[EINVAL]</b> | <i>fsname</i> points to an invalid file-system identifier; <i>fs_index</i> is zero, or invalid; <i>opcode</i> is invalid. |

## Diagnostics

---

Upon successful completion, `sysfs` returns the file-system type index if the *opcode* is `GETFSIND`, a value of 0 if the *opcode* is `GETFSTYP`, or the number of file system types configured if the *opcode* is `GETNFSSTYP`. Otherwise, a value of -1 is returned, and `errno` is set to indicate the error.

## Standards conformance

---

`sysfs` is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

# sysi86

machine-specific functions

## Syntax

```
cc ... -lc

#include <sys/sysi86.h>

long sysi86 (cmd, arg)
int cmd;
```

## Description

The **sysi86** system call implements machine-specific functions. The *cmd* argument determines the function to be performed. The types of the arguments expected depend on the function.

**RTODC** When *cmd* is **RTODC**, the expected argument is the address of a structure **rtc\_t** (from the header file *<sys/rtc.h>*):

```
struct rtc_t {
 char rtc_sec, rtc_asec, rtc_min, rtc_amin,
 rtc_hr, rtc_ahr, rtc_dow, rtc_dom,
 rtc_mon, rtc_yr, rtc_statusg,
 rtc_statusb, rtc_statusc, rtc_statusd;
};
```

This function reads the hardware time-of-day clock and returns the data in the structure referenced by the argument. This command is available only to root.

**RDUBLK** This command reads the u-block (per process user information as defined by the **user** structure in the *<sys/user.h>* header file) for a given process. When *cmd* is **RDUBLK**, **sysi86** takes three additional arguments: the process ID, the address of a buffer, and the number of bytes to read:

```
sysi86(RDUBLK, pid, buf, n)
int pid;
char *buf;
ind n;
```

**SI86FPHW** This command expects the address of an integer as its argument. After successful return from the system call, the integer specifies how floating-point computation is supported.

The low-order byte of the integer contains the value of *fpkind*, a variable specifying whether an 80287 or 80387 floating-point co-processor is present, emulated in software, or not supported. The values are defined in the header file *<sys/fp.h>*.

|       |                                         |
|-------|-----------------------------------------|
| FP_NO | no fp chip, no emulator (no fp support) |
| FP_SW | no fp chip, using software emulator     |
| FP_HW | chip present bit                        |
| FP_28 | 80287 chip present                      |
| FP_38 | 80387 chip present                      |

**SETNAME** This *cmd*, which is only available to root, expects an argument of type `char *` pointing to a NULL terminated string of at most 7 characters. The command changes the running system's *sysname* and *nodename* to this string. See `uname(S)`.

**STIME** When *cmd* is *STIME*, an argument of type `long` is expected. This function sets the system time and date (not the hardware clock). The argument contains the time as measured in seconds from 00:00:00 GMT January 1, 1970. Note that this command is only available to root.

**SI86DSCR** This command sets a segment or gate descriptor in the kernel. The following descriptor types are accepted:

- executable and data segments in the LDT at DPL 3
- a call gate in the GDT at DPL 3 pointing to a segment in the LDT

The argument is a pointer to a request structure that contains the values to be placed in the descriptor. The request structure is declared in the `<sys/sysi86.h>` header file.

**SI86MEM** This command returns the size of available memory in bytes.

**SI86SETPIPE** The argument is the name of a file. If the filesystem on which that file resides is mounted read/write, then that filesystem becomes the "pipe filesystem." Anonymous pipes created with `pipe(S)` use i-nodes and blocks on this pipe filesystem for temporary storage of the data passing through the pipe.

If the argument is the NULL pointer, then the pipe filesystem is disabled. This causes all subsequent `pipe()` calls to fail with error `ENODEV`. Existing anonymous pipes are not affected.

Named pipes use the filesystem on which they reside, not the pipe filesystem, and so are never affected by `SI86SETPIPE`.

Only root can set or disable the pipe filesystem. Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned and `errno` is set to indicate the error:

- [`EPERM`] The effective user ID is not root (the superuser).  
 [`ENOENT`] The named file does not exist.

[ENOTDIR] A component of file's path prefix is not a directory.  
 [EROFS] The filesystem is mounted read-only.  
 [ENODEV] The filesystem is incapable of supporting pipes.

**SI86GETPIPE** This command, which does not take an argument, returns the device number of the block special file containing the current pipe filesystem. (See command **SI86SETPIPE** for the definition of the "pipe filesystem".) Otherwise, -1 is returned and `errno` set to indicate the error:

[ENODEV] No pipe filesystem currently exists.

**SI86SWPI** When *cmd* is **SI86SWPI**, individual swapping areas may be added, deleted or the current areas determined. The address of an appropriately primed swap buffer is passed as the only argument. Refer to `<sys/swap.h>` header file for details on loading the buffer.

The format of the swap buffer is:

```
typedef struct swapint {
 char si_cmd; /* One of the command codes */
 /* listed below. */
 char *si_buf; /* For an SI_LIST function, this*/
 /* is a pointer to a buffer of */
 /* sizeof(swpt_t)*MSFILES bytes.*/
 /* For the other cases, it is a */
 /* pointer to a pathname of a */
 /* swap file. */
 int si_swpl0; /* The first block number of the*/
 /* swap file. Used only for */
 /* SI_ADD and SI_DEL. */
 int si_nblks; /* The size of the swap file in */
 /* blocks. Used only for an */
 /* SI_ADD request. */
} swpi_t;

/* The following are the possible values for si_cmd. */

#define SI_LIST 0 /* List the currently active */
 /* swap files. */
#define SI_ADD 1 /* Add a new swap file. */
#define SI_DEL 2 /* Delete one of the currently */
 /* active swap files. */
```

Note that the add and delete options of the command may only be used by root.

Typically, a swap area is added by a single call to `sysi86`. First, the swap buffer is primed with appropriate entries for the structure members. Then `sysi86` is invoked.

```
#include <sys/sysi86.h>
#include <sys/swap.h>

struct swapint swapbuf; /*swap into buffer ptr*/

sysi86(SI86SWPI, &swapbuf);
```

If this command succeeds, it returns 0 to the calling process. This command fails, returning -1, if one or more of the following is true:

- [EEXIST] Swap area specified has already been added
- [EFAULT] 1. *swapbuf* points to an invalid address  
2. *swapbuf.si\_buf* points to an invalid address
- [EINVAL] Bad arguments
- [ENOMEM] 1. Tried to delete last remaining swap area  
2. No place to put swapped pages when deleting a swap area
- [ENOSPC] Too many swap areas in use (if adding).
- [ENOTBLK] Swap area specified is not a block special device

## Return value

---

Upon error, -1 is returned and `errno` is set to indicate the error. When the *cmd* is invalid, `errno` is set to `EINVAL`.

## Notes

---

Normally after booting, the pipe filesystem is initially and automatically set by the system to be the root filesystem. However, if the system is booted with a read-only root, which is normally done only for installation (see `boot(HW)`), then there is no initial pipe filesystem and the `pipe(ADM)` command should be used once a suitable filesystem is mounted.

`pathconf(S)` can be used to determine whether or not a filesystem supports pipes.

## See also

---

`pathconf(S)`, `pipe(S)`, `uname(S)`, `pipe(ADM)`, `swap(ADM)`

## Standards conformance

---

The `sysi86` system call is not part of any currently supported standard; it is an extension of AT&T System V provided by The Santa Cruz Operation, Inc.

# system

issue a shell command

## Syntax

---

```
cc ... -lc

#include <stdio.h>
#include <stdlib.h>

int system (string)
char *string;
```

## Description

---

The **system** function causes the **string** to be given to **sh(C)** as input, as if the string had been typed as a command at a terminal. The current process waits until the shell has completed.

## Files

---

*/bin/sh*

## Return value

---

If **string** is not a null pointer, **system** returns the exit status of the shell in the same format as specified by the **waitpid(S)** function. The meaning of the return value, if it is not -1, can be decoded via macros like **WIFEXITED** defined in *<sys/wait.h>*, as described in **waitpid(S)**.

## Diagnostics

---

The **system** function forks to create a child process that in turn executes (using **exec(S)**) */bin/sh* in order to execute **string**. If the fork fails, **system** returns -1 and sets **errno**. If the fork succeeds but **exec** fails, **system** returns a value as if **sh(C)** had terminated using **\_exit(127)**.

The **system** function fails if:

- |          |                                                                                                                                                         |
|----------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| [EAGAIN] | The system-imposed limit on the total number of processes under execution, system-wide, or by a single user ID ( <b>CHILD_MAX</b> ), would be exceeded. |
| [ENOMEM] | Insufficient storage space is available.                                                                                                                |

## ***See also***

---

**exec(S), sh(C)**

## ***Standards conformance***

---

**system** is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

and ANSI X3.159-1989 Programming Language — C.

(S)



## **t\_accept**

accept a connect request

---

### **Syntax**

---

#### ***TLI syntax***

```
cc ... -lnsl_s

#include <sys/tiuser.h>

int t_accept (fd, resfd, call)
int fd, resfd;
struct t_call *call;
```

#### ***XTI syntax***

```
cc ... -lxti

#include <xti.h>

int t_accept (fd, resfd, call)
int fd, resfd;
struct t_call *call;
```

### **Description**

---

This function is issued by a transport user to accept a connect request. *fd* identifies the local transport endpoint where the connect indication arrived, *resfd* specifies the local transport endpoint where the connection is to be established, and *call* contains information required by the transport provider to complete the connection. *call* points to a **t\_call** structure containing the following members:

```
 struct netbuf addr; /* caller's address */
 struct netbuf opt; /* options */
 struct netbuf udata; /* user data returned by caller */
 int sequence; /* value returned by t_listen */
```

Refer to *netbuf(FP)* for information about the **netbuf** structure. In *call*, *addr* is the address of the caller, *opt* indicates any protocol-specific parameters associated with the connection, *udata* points to any user data to be returned to the caller, and *sequence* is the value returned by **t\_listen** that uniquely associates the response with a previously received connect indication.

A transport user may accept a connection on either the same, or on a different, local transport endpoint than the one on which the connect indication arrived. If the same endpoint is specified (that is, *resfd=fd*), the connection can be accepted unless the following condition is true: the user has received other

indications on that endpoint but has not responded to them (with t\_accept or t\_snddis). For this condition, t\_accept fails and sets t\_errno to TBAADF.

If a different transport endpoint is specified (*resfd!=fd*), the endpoint must be bound to a protocol address and must be in the T\_IDLE state (see t\_getstate(S)) before the t\_accept is issued.

For both types of endpoints, t\_accept fails and set t\_errno to TLOOK if there are indications (for example, a connect or disconnect) waiting to be received on that endpoint.

The values of parameters specified by opt and the syntax of those values are protocol specific. The udata argument enables the called transport user to send user data to the caller and the amount of user data must not exceed the limits supported by the transport provider as returned by t\_open or t\_getinfo. If the len field of udata is zero, no data is sent to the caller.

On failure, t\_errno may be set to one of the following:

|               |                                                                                                                                                                                               |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [TACCES]      | The user does not have permission to accept a connection on the responding transport endpoint or use the specified options.                                                                   |
| [TBADDDATA]   | The amount of user data specified was not within the bounds allowed by the transport provider.                                                                                                |
| [TBAADF]      | The specified file descriptor does not refer to a transport endpoint, or the user is illegally accepting a connection on the same transport endpoint on which the connect indication arrived. |
| [TBADOPT]     | The specified options were in an incorrect format or contained illegal information.                                                                                                           |
| [TBADSEQ]     | An invalid sequence number was specified.                                                                                                                                                     |
| [TFLOW]       | A system error occurred while sending the connect request.                                                                                                                                    |
| [TLOOK]       | An asynchronous event has occurred on the transport endpoint referenced by <i>fd</i> and requires immediate attention.                                                                        |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider.                                                                                                                          |

*t\_accept(S)*

|             |                                                                                                                                                                                     |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [TOUTSTATE] | The function was issued in the wrong sequence on the transport endpoint referenced by <i>fd</i> , or the transport endpoint referred to by <i>resfd</i> is not in the T_IDLE state. |
| [TSYSERR]   | A system error occurred during execution of this function.                                                                                                                          |

## ***See also***

---

**netbuf(FP),**

**t\_connect(S), t\_getstate(S), t\_listen(S), t\_open(S), t\_rcvconnect(S)**

## ***Diagnostics***

---

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and **t\_errno** is set to indicate the error.

## ***Standards conformance***

---

**t\_accept** is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

# t\_alloc

allocate a library structure

## Syntax

### TLI syntax

```
cc ... -lnsl_s

#include <sys/tiuser.h>

char * t_alloc (fd, struct_type, fields)
int fd, struct_type, fields;
```

### XTI syntax

```
cc ...-lxti

#include <xti.h>

char * t_alloc (fd, struct_type, fields)
int fd, struct_type, fields;
```

## Description

The **t\_alloc** function dynamically allocates memory for the various transport function argument structures as specified below. This function allocates memory for the specified structure, and also allocates memory for buffers referenced by the structure.

The structure to allocate is specified by *struct\_type*, and can be one of the following:

| TLI <i>struct_type</i><br>Value | XTI <i>struct_type</i><br>Value | Allocated<br>Structure |
|---------------------------------|---------------------------------|------------------------|
| T_BIND                          | T_BIND_STR                      | struct t_bind          |
| T_CALL                          | T_CALL_STR                      | struct t_call          |
| T_DIS                           | T_DIS_STR                       | struct t_discon        |
| T_INFO                          | T_INFO_STR                      | struct t_info          |
| T_OPTMGMT                       | T_OPTMGMT_STR                   | struct t_optmgmt       |
| T_UDERROR                       | T_UDERROR_STR                   | struct t_uderr         |
| T_UNITDATA                      | T_UNITDATA_STR                  | struct t_unitdata      |

Each of these structures may subsequently be used as an argument to one or more transport functions.

Each of the previously listed structures, except **T\_INFO** (TLI) or **T\_INFO\_STR** (XTI), contains at least one field of type **struct netbuf**. **netbuf** is described in *netbuf(FP)*. For each field of this type, the user may specify that the buffer for that field should be allocated as well. The *fields* argument specifies this option, where the argument is the bitwise-OR of any of the following:

- T\_ALL**      All relevant fields of the given structure.
- T\_ADDR**    The **addr** field of the **t\_bind**, **t\_call**, **t\_unitdata**, or **t\_uderr** structures.
- T\_OPT**      The **opt** field of the **t\_optmgmt**, **t\_call**, **t\_unitdata**, or **t\_uderr** structures.
- T\_UDATA**    The **udata** field of the **t\_call**, **t\_discon**, or **t\_unitdata** structures.

For each field specified in *fields*, **t\_alloc** allocates memory for the buffer associated with the field, and initialize the **buf** pointer and **maxlen** (see **netbuf** in *netbuf(FP)* for a description of the **buf** and **maxlen**) fields accordingly. The length of the buffer allocated is based on the same size information that is returned to the user on **t\_open** and **t\_getinfo**. Thus, *fd* must refer to the transport endpoint through which the newly allocated structure is passed, so that the appropriate size information can be accessed. If the size value associated with any specified field is -1 or -2 (see **t\_open** or **t\_getinfo**), **t\_alloc** is unable to determine the size of the buffer to allocate and fails, setting **t\_errno** to **TSYSERR** and **errno** to **EINVAL**. For any field not specified in *fields*, **buf** is set to **NULL** and **maxlen** is set to zero.

Use of **t\_alloc** to allocate structures helps ensure the compatibility of user programs with future releases of the transport interface.

On failure, **t\_errno** may be set to one of the following:

- [TBADF]**                      The specified file descriptor does not refer to a transport endpoint.
- [TNOSTRUCTYPE]**          For XTI, an incorrect value was specified in the *struct\_type* argument.

**[TSYSERR]**

A system error has occurred during execution of this function. The following conditions accompany this error:

| <b>t_errno Value</b> | <b>errno Value</b> | <b>Cause</b>                                                           |
|----------------------|--------------------|------------------------------------------------------------------------|
| <b>TSYSERR</b>       | not set            | Device or system failure                                               |
| <b>TSYSERR</b>       | <b>EINVAL</b>      | <i>struct_type</i> out of range, or internal error                     |
| <b>TSYSERR</b>       | <b>ENOMEM</b>      | Memory could not be allocated for specified <i>struct_type</i> request |

## See also

---

netbuf(FP), t\_free(S), t\_getinfo(S), t\_open(S), t\_info(FP)

## Diagnostics

---

On successful completion, t\_alloc returns a pointer to the newly allocated structure. On failure, NULL is returned.

## Standards conformance

---

t\_alloc is conformant with:

AT&T SVID Issue 2;  
X/Open Portability Guide, Issue 3, 1989;  
and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

## **t\_bind**

bind an address to a transport endpoint

---

### **Syntax**

---

#### ***TLI syntax***

```
cc ... -lnsl_s

#include <sys/tiuser.h>

int t_bind (fd, req, ret)
int fd;
struct t_bind *req;
struct t_bind *ret;
```

#### ***XTI syntax***

```
cc ... -lxti

#include <xti.h>

int t_bind (fd, req, ret)
int fd;
struct t_bind *req;
struct t_bind *ret;
```

### **Description**

---

This function associates a protocol address with the transport endpoint specified by *fd* and activates that transport endpoint. In connection mode, the transport provider may begin accepting or requesting connections on the transport endpoint. In connectionless mode, the transport user may send or receive data units through the transport endpoint.

The *req* and *ret* arguments point to a **t\_bind** structure containing the following members:

```
 struct netbuf addr;
 unsigned qlen;
```

*netbuf* is described in *netbuf*(FP). The *addr* field of the **t\_bind** structure specifies a protocol address and the *qlen* field is used to indicate the maximum number of outstanding connect indications.

*req* is used to request that an address, represented by the *netbuf* structure, be bound to the given transport endpoint. *len* (see *netbuf* in *netbuf*(FP); also for *buf* and *maxlen*) specifies the number of bytes in the address and *buf* points to the address buffer. *maxlen* has no meaning for the *req* argument. On return,

*ret* contains the address that the transport provider actually bound to the transport endpoint; this may be different from the address specified by the user in *req*. In *ret*, the user specifies *maxlen* which is the maximum size of the address buffer and *buf* which points to the buffer where the address is to be placed. On return, *len* specifies the number of bytes in the bound address and *buf* points to the bound address. If *maxlen* is not large enough to hold the returned address, an error results.

If the requested address is not available, or if no address is specified in *req* (the *len* field of *addr* in *req* is zero) the transport provider assigns an appropriate address to be bound, and returns that address in the *addr* field of *ret*. The user can compare the addresses in *req* and *ret* to determine whether the transport provider bound the transport endpoint to a different address than that requested.

*req* may be NULL if the user does not wish to specify an address to be bound. Here, the value of *qlen* is assumed to be zero, and the transport provider must assign an address to the transport endpoint. Similarly, *ret* may be NULL if the user does not care what address was bound by the provider and is not interested in the negotiated value of *qlen*. It is valid to set *req* and *ret* to NULL for the same call, in which case the provider chooses the address to bind to the transport endpoint and does not return that information to the user.

The *qlen* field has meaning only when initializing a connection-mode service. It specifies the number of outstanding connect indications the transport provider should support for the given transport endpoint. An outstanding connect indication is one that has been passed to the transport user by the transport provider. A value of *qlen* greater than zero is only meaningful when issued by a passive transport user that expects other users to call it. The value of *qlen* is negotiated by the transport provider and may be changed if the transport provider cannot support the specified number of outstanding connect indications. On return, the *qlen* field in *ret* contains the negotiated value.

This function allows more than one transport endpoint to be bound to the same protocol address (however, the transport provider must support this capability also), but it is not allowable to bind more than one protocol address to the same transport endpoint. If a user binds more than one transport endpoint to the same protocol address, only one endpoint can be used to listen for connect indications associated with that protocol address. In other words, only one *t\_bind* for a given protocol address may specify a value of *qlen* greater than zero. In this way, the transport provider can identify which transport endpoint should be notified of an incoming connect indication. If a user attempts to bind a protocol address to a second transport endpoint with a value of *qlen* greater than zero, the transport provider assigns another address to be bound to that endpoint. If a user accepts a connection on the transport endpoint that is being used as the listening endpoint, the bound pro-



to col address is found to be busy for the duration of that connection. No other transport endpoints may be bound for listening while that initial listening endpoint is in the data transfer phase. This prevents more than one transport endpoint bound to the same protocol address from accepting connect indications.

## *Return value*

---

The `t_bind` function returns 0 on success and -1 on failure, and `t_errno` is set to indicate the error.

## *Diagnostics*

---

On failure, `t_errno` may be set to one of the following:

|             |                                                                                                                                                                                                                                |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [TACCES]    | The user does not have permission to use the specified address.                                                                                                                                                                |
| [TBADADDR]  | The specified protocol address was in an incorrect format or contained illegal information.                                                                                                                                    |
| [TBADEF]    | The specified file descriptor does not refer to a transport endpoint.                                                                                                                                                          |
| [TBUFOVFLW] | The number of bytes allowed for an incoming argument is not sufficient to store the value of that argument. The provider's state changes to <code>T_IDLE</code> and the information to be returned in <i>ret</i> is discarded. |
| [TNOADDR]   | The transport provider could not allocate an address.                                                                                                                                                                          |
| [TOUTSTATE] | The function was issued in the wrong sequence.                                                                                                                                                                                 |
| [TSYSERR]   | A system error has occurred during execution of this function.                                                                                                                                                                 |
| TADDRBUSY   | The address requested is in use and the transport provider could not allocate a new address.                                                                                                                                   |

## *See also*

---

`netbuf(FP)`, `t_open(S)`, `t_optmgmt(S)`, `t_unbind(S)`,

## *Standards conformance*

---

`t_bind` is conformant with:

AT&T SVID Issue 2;  
X/Open Portability Guide, Issue 3, 1989;  
and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

# t\_close

close a transport endpoint

## Syntax

### TLI syntax

```
cc...-lnsl_s

#include <sys/tiuser.h>

int t_close (fd)
int fd;
```

### XTI syntax

```
cc...-lxti

#include <xti.h>

int t_close (fd)
int fd;
```

## Description

The **t\_close** function informs the transport provider that the user is finished with the transport endpoint specified by *fd*, and frees any local library resources associated with the endpoint. In addition, **t\_close** closes the file associated with the transport endpoint.

The **t\_close** function should be called from the **T\_UNBND** state (see **t\_getstate(S)**). However, this function does not check state information, so it may be called from any state to close a transport endpoint. If this occurs, the local library resources associated with the endpoint will be freed automatically. In addition, **close(S)** will be issued for that file descriptor; the close will be abortive if no other process has that file open, and will break any transport connection that may be associated with that endpoint.

On failure, **t\_errno** may be set to the following:

|         |                                                                       |
|---------|-----------------------------------------------------------------------|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
|---------|-----------------------------------------------------------------------|

`t_close(S)`

## *See also*

---

`t_getstate(S)`, `t_open(S)`, `t_unbind(S)`,

## *Diagnostics*

---

The `t_close` function returns 0 on success and -1 on failure, and `t_errno` is set to indicate the error.

## *Standards conformance*

---

`t_close` is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

# t\_connect

establish a connection with another transport user

## Syntax

### TLI syntax

```
cc ... -lnsl_s

#include <sys/tiuser.h>

int t_connect (fd, sndcall, rcvcall)
int fd;
struct t_call *sndcall;
struct t_call *rcvcall;
```

### XTI syntax

```
cc ... -lxti

#include <xti.h>

int t_connect (fd, sndcall, rcvcall)
int fd;
struct t_call *sndcall;
struct t_call *rcvcall;
```

## Description

This function enables a transport user to request a connection to the specified destination transport user. *fd* identifies the local transport endpoint where communication will be established, while *sndcall* and *rcvcall* point to a *t\_call* structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

*sndcall* specifies information needed by the transport provider to establish a connection, and *rcvcall* specifies information that is associated with the newly established connection.

*netbuf* is described in *netbuf*(FP). In *sndcall*, *addr* specifies the protocol address of the destination transport user, *opt* presents any protocol-specific information that might be needed by the transport provider, *udata* points to optional user data that may be passed to the destination transport user during connection establishment, and *sequence* has no meaning for this function.

On return in *rcvcall*, *addr* returns the protocol address associated with the responding transport endpoint; *opt* presents any protocol-specific information associated with the connection; *udata* points to optional user data that may be returned by the destination transport user during connection establishment; and *sequence* has no meaning for this function.

The *opt* argument implies no structure on the options that may be passed to the transport provider. The transport provider is free to specify the structure of any options passed to it. These options are specific to the underlying protocol of the transport provider. The user may choose not to negotiate protocol options by setting the *len* field of *opt* to zero. In this case, the provider may use default options.

The *udata* argument enables the caller to pass user data to the destination transport user and receive user data from the destination user during connection establishment. However, the amount of user data must not exceed the limits supported by the transport provider as returned by *t\_open(S)* or *t\_getinfo(S)*. If the *len* (see *netbuf* in *netbuf(FP)*) field of *udata* is zero in *sndcall*, no data will be sent to the destination transport user.

On return, the *addr*, *opt*, and *udata* fields of *rcvcall* will be updated to reflect values associated with the connection. Thus, the *maxlen* (see *netbuf* in *netbuf(FP)*) field of each argument must be set before issuing this function to indicate the maximum size of the buffer for each. However, *rcvcall* may be NULL, in which case no information is given to the user on return from *t\_connect*.

By default, *t\_connect* executes in synchronous mode, and will wait for the destination user's response before returning control to the local user. A successful return (that is, a return value of zero) indicates that the requested connection has been established. However, if *O\_NONBLOCK* is set (via *t\_open* or *fcntl*), *t\_connect* executes in asynchronous mode. In this case, the call will not wait for the remote user's response, but will return control immediately to the local user and return -1 with *t\_errno* set to *TNODATA* to indicate that the connection has not yet been established. In this way, the function simply initiates the connection establishment procedure by sending a connect request to the destination transport user.

On failure, *t\_errno* may be set to one of the following:

|            |                                                                                                |
|------------|------------------------------------------------------------------------------------------------|
| [TACCES]   | The user does not have permission to use the specified address or options.                     |
| [TBADADDR] | The specified protocol address was in an incorrect format or contained illegal information.    |
| [TBADDATA] | The amount of user data specified was not within the bounds allowed by the transport provider. |
| [TBADEF]   | The specified file descriptor does not refer to a transport endpoint.                          |

|               |                                                                                                                                                                                                                                                                                                        |
|---------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [TBADOPT]     | The specified protocol options were in an incorrect format or contained illegal information.                                                                                                                                                                                                           |
| [TBUFOVFLW]   | The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. If executed in synchronous mode, the provider's state, as seen by the user, changes to T_DATAXFER, and the connect indication information to be returned in <i>rcvcall</i> is discarded. |
| [TLOOK]       | An asynchronous event has occurred on this transport endpoint and requires immediate attention.                                                                                                                                                                                                        |
| [TNODATA]     | O_NONBLOCK was set, so the function successfully initiated the connection establishment procedure, but did not wait for a response from the remote user.                                                                                                                                               |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider.                                                                                                                                                                                                                                   |
| [TOUTSTATE]   | The function was issued in the wrong sequence.                                                                                                                                                                                                                                                         |
| [TSYSERR]     | A system error has occurred during execution of this function.                                                                                                                                                                                                                                         |

## Diagnostics

---

The `t_connect` function returns 0 on success and -1 on failure, and `t_errno` is set to indicate the error.

## See also

---

`netbuf(FP)`, `t_accept(S)`, `t_getinfo(S)`, `t_listen(S)`, `t_open(S)`, `t_optmgmt(S)`, `t_rcvconnect(S)`

## Standards conformance

---

`t_connect` conforms to:

AT&T SVID Issue 2; the X/Open Portability Guide, Issue 3, 1989 and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

## **t\_error**

produce error message

---

### ***Syntax***

---

#### ***TLI syntax***

```
cc ... -lnsl_s

#include <sys/tiuser.h>

void t_error (errmsg)
char *errmsg;
extern int t_errno;
extern char *t_errlist[];
extern int t_nerr;
```

#### ***XTI syntax***

```
cc ... -lxti

#include <xti.h>

void t_error (errmsg)
char *errmsg;
extern int t_errno;
extern char *t_errlist[];
extern int t_nerr;
```

### ***Description***

---

**t\_error** produces a message on the standard error output which describes the last error encountered during a call to a transport function. The argument string *errmsg* is a user-supplied error message that gives context to the error.

**t\_error** prints the user-supplied error message followed by a colon and the standard transport function error message for the current value contained in **t\_errno**. If **t\_errno** is **TSYSERR**, **t\_error** will also print the standard error message for the current value contained in **errno** (see *netbuf(FP)*).

**t\_errlist** is the array of message strings, to allow user message formatting. **t\_errno** can be used as an index into this array to retrieve the error message string (without a terminating newline). **t\_nerr** is the maximum index value for the **t\_errlist** array.

**t\_errno** is set when an error occurs and is not cleared on subsequent successful calls.

## ***Example***

---

If a `t_connect` function fails on transport endpoint *fd2* because a bad address was given, the following call might follow the failure:

```
t_error("t_connect failed on fd2");
```

The diagnostic message would print as:

```
t_connect failed on fd2: Incorrect transport address format
```

where `t_connect failed on fd2` tells the user which function failed on which transport endpoint, and `Incorrect transport address format` identifies the specific error that occurred.

## ***Standards conformance***

---

`t_error` is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

(S)



## **t\_free**

free a library structure

### **Syntax**

#### ***TLI syntax***

```
cc...-lnsl_s
#include <sys/tiuser.h>

int t_free (ptr, struct_type)
char *ptr;
int struct_type;
```

#### ***XTI syntax***

```
cc...-lxti
#include <xti.h>

int t_free (ptr, struct_type)
char *ptr;
int struct_type;
```

### **Description**

The **t\_free** function frees memory previously allocated by **t\_alloc**. This function frees memory for the specified structure and also frees memory for buffers referenced by the structure.

*ptr* points to one of the six structure types described for **t\_alloc**, and *struct\_type* identifies the type of that structure which can be one of the following, where each of these structures is used as an argument to one or more transport functions.

| TLI <i>struct_type</i><br>Value | XTI <i>struct_type</i><br>Value | Allocated<br>Structure |
|---------------------------------|---------------------------------|------------------------|
| T_BIND                          | T_BIND_STR                      | struct t_bind          |
| T_CALL                          | T_CALL_STR                      | struct t_call          |
| T_DIS                           | T_DIS_STR                       | struct t_discon        |
| T_INFO                          | T_INFO_STR                      | struct t_info          |
| T_OPTMGMT                       | T_OPTMGMT_STR                   | struct t_optmgmt       |
| T_UDERROR                       | T_UDERROR_STR                   | struct t_uderr         |
| T_UNITDATA                      | T_UNITDATA_STR                  | struct t_unitdata      |

The **t\_free** function checks the **addr**, **opt**, and **udata** fields of the given structure (as appropriate) and frees the buffers pointed to by the **buf** field of the **netbuf** (see **netbuf(FP)**) structure. If **buf** is **NULL**, **t\_free** does not attempt to free memory. After all buffers are freed, **t\_free** frees the memory associated with the structure pointed to by **ptr**.

Undefined results occur if **ptr** or any of the **buf** pointers points to a block of memory that was not previously allocated by **t\_alloc**.

On failure, **t\_errno** may be set to the following:

#### TNOSTRUCTYPE

**struct\_type** is out of the range. This error code only occurs with XTI.

#### [TSYSERR]

A system error has occurred during execution of this function. For TLI, this error code is also returned when **struct\_type** is out of range; when this error occurs, **errno** is set to **EINVAL**.

## Diagnostics

---

The **t\_free** function returns 0 on success and -1 on failure, and **t\_errno** is set to indicate the error.

## See also

---

**netbuf(FP)**, **t\_alloc(S)**

## Standards conformance

---

**t\_free** is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

## **t\_getinfo**

---

get protocol-specific service information

### ***Syntax***

---

#### ***TLI syntax***

```
cc...-lnsl_s

#include <sys/tiuser.h>

int t_getinfo (fd, info)
int fd;
struct t_info *info;
```

#### ***XTI syntax***

```
cc...-lxti

#include <xti.h>

int t_getinfo (fd, info)
int fd;
struct t_info *info;
```

### ***Description***

---

This function returns the current characteristics of the underlying transport protocol associated with file descriptor *fd*. The *info* structure is used to return the same information returned by *t\_open*. This function enables a transport user to access this information during any phase of communication.

This argument points to the *t\_info*(FP) structure.

On failure, *t\_errno* may be set to one of the following:

- |           |                                                                                                                              |
|-----------|------------------------------------------------------------------------------------------------------------------------------|
| [TBADF]   | The specified file descriptor does not refer to a transport endpoint.                                                        |
| [TSYSERR] | A system error has occurred during execution of this function. When this error code is set, <i>errno</i> is also set to EIO. |

### ***Diagnostics***

---

The *t\_getinfo* function returns 0 on success and -1 on failure, and *t\_errno* is set to indicate the error.

## ***See also***

---

**t\_info(FP), t\_open(S)**

## ***Standards conformance***

---

**t\_getinfo** is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

## **t\_getstate**

---

get the current state

### ***Syntax***

---

#### ***TLI syntax***

```
cc ... -lnsl_s

#include <sys/tiuser.h>

int t_getstate (fd)
int fd;
```

#### ***XTI syntax***

```
cc ... -lxti

#include <xti.h>

int t_getstate (fd)
int fd;
```

### ***Description***

---

The **t\_getstate** function returns the current state of the provider associated with the transport endpoint specified by *fd*.

On failure, **t\_errno** may be set to one of the following:

|                      |                                                                       |
|----------------------|-----------------------------------------------------------------------|
| <b>[TBADF]</b>       | The specified file descriptor does not refer to a transport endpoint. |
| <b>[TSTATECHNG]</b>  | The transport provider is undergoing a state change.                  |
| <b>[TSYSERR]</b>     | A system error has occurred during execution of this function.        |
| <b>[TNOTSUPPORT]</b> | This function is not supported by the current implementation of XTI.  |

## Diagnostics

---

The **t\_getstate** function returns the current state on successful completion and -1 on failure, and **t\_errno** is set to indicate the error. The current state may be one of the following:

|                   |                                                                      |
|-------------------|----------------------------------------------------------------------|
| <b>T_DATAXFER</b> | data transfer                                                        |
| <b>T_IDLE</b>     | idle                                                                 |
| <b>T_INCON</b>    | incoming connection pending                                          |
| <b>T_INREL</b>    | incoming orderly release (waiting for an orderly release request)    |
| <b>T_OUTCON</b>   | outgoing connection pending                                          |
| <b>T_OUTREL</b>   | outgoing orderly release (waiting for an orderly release indication) |
| <b>T_UNBND</b>    | unbound                                                              |

If the provider is undergoing a state transition when **t\_getstate** is called, the function fails.

## See also

---

**t\_open(S)**

## Standards conformance

---

**t\_getstate** is conformant with:

AT&T SVID Issue 2;  
X/Open Portability Guide, Issue 3, 1989;  
and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

## **t\_listen**

listen for a connect request

---

### ***Syntax***

---

#### ***TLI syntax***

```
cc...-lnsl_s

#include <sys/tiuser.h>

int t_listen (fd, call)
int fd;
struct t_call *call;
```

#### ***XTI syntax***

```
cc...-lxti

#include <xti.h>

int t_listen (fd, call)
int fd;
struct t_call *call;
```

### ***Description***

---

This function listens for a connect request from a calling transport user. *fd* identifies the local transport endpoint where connect indications arrive, and on return, *call* contains information describing the connect indication. *call* points to a **t\_call** structure which contains the following members:

```
 struct netbuf addr;
 struct netbuf opt;
 struct netbuf udata;
 int sequence;
```

**netbuf** is described in *netbuf(FP)*. In *call*, *addr* returns the protocol address of the calling transport user; *opt* returns protocol-specific parameters associated with the connect request; *udata* returns any user data sent by the caller on the connect request; and *sequence* is a number that uniquely identifies the returned connect indication. The value of *sequence* enables the user to listen for multiple connect indications before responding to any of them.

Since this function returns values for the *addr*, *opt*, and *udata* fields of *call*, the *maxlen* (see **netbuf** in *netbuf(FP)*) field of each must be set before issuing the **t\_listen** to indicate the maximum size of the buffer for each.

By default, `t_listen` executes in synchronous mode and waits for a connect indication to arrive before returning to the user. However, if `O_NONBLOCK` is set (via `t_open` or `fcntl`), `t_listen` executes asynchronously, reducing to a poll for existing connect indications. If none are available, it returns -1 and sets `t_errno` to `TNODATA`.

On failure, `t_errno` may be set to one of the following:

|               |                                                                                                                                                                                                                                                                                                                                                                             |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [TBADF]       | The specified file descriptor does not refer to a transport endpoint.                                                                                                                                                                                                                                                                                                       |
| [TBADQLEN]    | (XTI only.) The number of outstanding connect indications is equal to <i>qlen</i> .                                                                                                                                                                                                                                                                                         |
| [TBUFOVFLW]   | The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. The provider's state, as seen by the user, changes to <code>T_INCON</code> , and the connect indication information to be returned in <i>call</i> is discarded.                                                                                               |
| [TLOOK]       | An asynchronous event has occurred on this transport endpoint and requires immediate attention.                                                                                                                                                                                                                                                                             |
| [TNODATA]     | <code>O_NONBLOCK</code> was set, but no connect indications had been queued.                                                                                                                                                                                                                                                                                                |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider.                                                                                                                                                                                                                                                                                                        |
| [TOUTSTATE]   | (XTI only.) The function was called out with sequence.                                                                                                                                                                                                                                                                                                                      |
| [TSYSERR]     | A system error occurred during execution of this function. <code>errno</code> may have any of the following values:<br><b>EFAULT</b><br>The area specified in the <code>t_call</code> structure is not (or not completely) located in the user address space.<br><b>EIO</b><br>The transport provider is no longer operable.<br><b>EPROTO</b><br>A protocol error occurred. |

## Diagnostics

---

The `t_listen` function returns 0 on success and -1 on failure, and `t_errno` is set to indicate the error.



## **Warning**

---

If a user issues **t\_listen** in synchronous mode on a transport endpoint that was not bound for listening (that is, **qlen** was zero on **t\_bind**), the call waits forever because no connect indications arrive on that endpoint.

## **See also**

---

**netbuf(FP), t\_accept(S), t\_bind(S), t\_connect(S), t\_open(S), t\_rcvconnect(S)**

## **Standards conformance**

---

**t\_listen** is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

# t\_look

look at the current event on a transport endpoint

## Syntax

### *TLI syntax*

```
cc ... -lnsl_s
#include <sys/tiuser.h>

int t_look (fd)
int fd;
```

### *XTI syntax*

```
cc ... -lxti
#include <xti.h>

int t_look (fd)
int fd;
```

## Description

This function returns the current event on the transport endpoint specified by *fd*. This function enables a transport provider to notify a transport user of an asynchronous event when the user is issuing functions in synchronous mode. Certain events require immediate notification of the user and are indicated by a specific error, **TLOOK**, on the current or next function to be executed.

This function also enables a transport user to poll a transport endpoint periodically for asynchronous events.

## Diagnostics

On failure, **t\_errno** may be set to one of the following:

- |                  |                                                                                                                                                                               |
|------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>[TBADF]</b>   | The specified file descriptor does not refer to a transport endpoint.                                                                                                         |
| <b>[TSYSERR]</b> | A system error occurred during execution of this function. When this error code, <b>errno</b> is set to <b>EPROTO</b> indicating a protocol error such as an illegal message. |

## ***Return value***

---

Upon success, **t\_look** returns a value that indicates which of the allowable events has occurred, or returns zero if no event exists. One of the following events is returned:

|                     |                                |
|---------------------|--------------------------------|
| <b>T_CONNECT</b>    | connect confirmation received  |
| <b>T_DATA</b>       | normal data received           |
| <b>T_DISCONNECT</b> | disconnect received            |
| <b>T_ERROR</b>      | fatal error indication         |
| <b>T_EXDATA</b>     | expedited data received        |
| <b>T_LISTEN</b>     | connection indication received |
| <b>T_ORDREL</b>     | orderly release indication     |
| <b>T_UDERR</b>      | datagram error indication      |

On failure, -1 is returned, and **t\_errno** is set to indicate the error.

## ***See also***

---

**t\_open(S),**

## ***Standards conformance***

---

**t\_look** is conformant with:

AT&T SVID Issue 2;  
X/Open Portability Guide, Issue 3, 1989;  
and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

# t\_open

establish a transport endpoint

## Syntax

### TLI syntax

```
cc ... -lnsl_s

#include <sys/tiuser.h>

int t_open (path, oflag, info)
char *path;
int oflag;

struct t_info *info;
```

### XTI syntax

```
cc ... -lxti

#include <xti.h>

int t_open (path, oflag, info)
char *path;
int oflag;

struct t_info *info;
```

## Description

The **t\_open** function must be called as the first step in the initialization of a transport endpoint. This function establishes a transport endpoint by opening a UNIX system file that identifies a particular transport provider (that is, transport protocol) and returning a file descriptor that identifies that endpoint. For example, opening the file */dev/iso\_cots* identifies an OSI connection-oriented transport layer protocol as the transport provider.

**path** points to the pathname of the file to open, and **oflag** identifies any open flags (as in **open(S)**). **t\_open** returns a file descriptor that is used by all subsequent functions to identify the particular local transport endpoint.

This function also returns various default characteristics of the underlying transport protocol by setting fields in the **t\_info** structure. This argument points to the **t\_info(FP)** structure.

The **servtype** member of **t\_info** may specify one of the following values on return:

|                   |                                                                                                                                                                        |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>T_CLTS</b>     | The transport provider supports a connectionless-mode service. For this service type, <b>t_open</b> returns -2 for <b>etsdu</b> , <b>connect</b> , and <b>discon</b> . |
| <b>T_COTS</b>     | The transport provider supports a connection-mode service but does not support the optional orderly release facility.                                                  |
| <b>T_COTS_ORD</b> | The transport provider supports a connection-mode service with the optional orderly release facility.                                                                  |

A single transport endpoint may support only one of the above services at one time.

If **info** is set to **NULL** by the transport user, no protocol information is returned by **t\_open**.

On failure, **t\_errno** may be set to the following:

|                     |                                                                                                                                                |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| [ <b>TBADFLAG</b> ] | (XTI only.) <i>oflag</i> is invalid. The acceptable values are <b>O_RDWR</b> or <b>O_RDWR</b> Or'ed with <b>O_NONBLOCK</b> .                   |
| [ <b>TBADNAME</b> ] | (XTI only.) The <b>open</b> of the specified <i>path</i> failed. <b>errno</b> is set to <b>ENOENT</b> .                                        |
| [ <b>TSYSERR</b> ]  | A system error occurred during execution of this function. One reason for this error is an I/O error; <b>errno</b> is then set to <b>EIO</b> . |

## *Diagnostics*

---

The **t\_open** function returns a valid file descriptor on success and -1 on failure, and **t\_errno** is set to indicate the error.

## *See also*

---

**open(S), t\_info(FP)**

## *Standards conformance*

---

**t\_open** is conformant with:

AT&T SVID Issue 2;  
X/Open Portability Guide, Issue 3, 1989;  
and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

# t\_optmgmt

manage options for a transport endpoint

## Syntax

### TLI syntax

```
cc ...-lnsl_s

#include <sys/tiuser.h>

int t_optmgmt (fd, req, ret)
int fd;
struct t_optmgmt *req;
struct t_optmgmt *ret;
```

## Description

The **t\_optmgmt** function enables a transport user to retrieve, verify, or negotiate protocol options with the transport provider. *fd* identifies a bound transport endpoint.

The *req* and *ret* arguments point to a **t\_optmgmt** structure containing the following members:

```
struct netbuf opt;
long flags;
```

The *opt* field identifies protocol options, and the *flags* field is used to specify the action to take with those options.

The options are represented by a **netbuf** (see *netbuf*(FP); also for *len*, *buf*, and *maxlen*) structure in a manner similar to the address in **t\_bind**. *req* is used to request a specific action of the provider and to send options to the provider. *len* specifies the number of bytes in the options, *buf* points to the options buffer, and *maxlen* has no meaning for the *req* argument. The transport provider may return options and flag values to the user through *ret*. For *ret*, *maxlen* specifies the maximum size of the options buffer and *buf* points to the buffer where the options are to be placed. On return, *len* specifies the number of bytes of options returned. *maxlen* has no meaning for the *req* argument, but must be set in the *ret* argument to specify the maximum number of bytes the options buffer can hold. The actual structure and content of the options is imposed by the transport provider.

The *flags* field of *req* can specify one of the following actions:

|                |                                                                                                                                                                                                                                                                             |
|----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>T_CHECK</b> | This action enables the user to verify whether the options specified in <i>req</i> are supported by the transport provider. On return, the <i>flags</i> field of <i>ret</i> either has <b>T_SUCCESS</b> or <b>T_FAILURE</b> set to indicate to the user whether the options |
|----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

are supported. These flags are only meaningful for the **T\_CHECK** request.

|                    |                                                                                                                                                                                                                                                         |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>T_DEFAULT</b>   | This action enables a user to retrieve the default options supported by the transport provider into the <b>opt</b> field of <i>ret</i> . In <i>req</i> , the <b>len</b> field of <b>opt</b> must be zero, and the <b>buf</b> field may be <b>NULL</b> . |
| <b>T_NEGOTIATE</b> | This action enables the user to negotiate the values of the options specified in <i>req</i> with the transport provider. The provider evaluates the requested options and negotiates the values, returning the negotiated values through <i>ret</i> .   |

If issued as part of the connectionless-mode service, **t\_optmgmt** may block due to flow control constraints. The function does not complete until the transport provider has processed all previously sent data units.

## Return value

---

The **t\_optmgmt** function returns 0 on success and -1 on failure, and **t\_errno** is set to indicate the error.

## Note

---

This routine is not supported by XTI. If this routine is called under XTI, the **TNOTSUPPORT** error code is returned and a -1 is returned.

## Diagnostics

---

On failure, **t\_errno** may be set to one of the following:

|                    |                                                                                                                                                                        |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>[TACCES]</b>    | The user does not have permission to negotiate the specified options.                                                                                                  |
| <b>[TBADF]</b>     | The specified file descriptor does not refer to a transport endpoint.                                                                                                  |
| <b>[TBADFLAG]</b>  | An invalid flag was specified.                                                                                                                                         |
| <b>[TBADOPT]</b>   | The specified protocol options were in an incorrect format or contained illegal information.                                                                           |
| <b>[TBUFOVFLW]</b> | The number of bytes allowed for an incoming argument is not sufficient to store the value of that argument. The information to be returned in <i>ret</i> is discarded. |
| <b>[TOUTSTATE]</b> | The function was issued in the wrong sequence.                                                                                                                         |
| <b>[TSYSERR]</b>   | A system error occurred during execution of this function.                                                                                                             |

## ***See also***

---

**netbuf(FP), t\_getinfo(S), t\_open(S)**

## ***Standards conformance***

---

**t\_optmgmt** is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

(S)



## **t\_rcv**

---

receive data or expedited data sent over a connection

### ***Syntax***

---

#### ***TLI syntax***

**cc...-lnsl\_s**

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

```
int t_rcv (fd, buf, nbytes, flags)
int fd;
char *buf;
unsigned nbytes;
int *flags;
```

#### ***XTI syntax***

**cc...-lxti**

**#include <xti.h>**

```
int t_rcv (fd, buf, nbytes, flags)
int fd;
char *buf;
unsigned nbytes;
int *flags;
```

### ***Description***

---

This function receives either normal or expedited data. *fd* identifies the local transport endpoint through which data arrives; *buf* points to a receive buffer where user data is placed; and *nbytes* specifies the size of the receive buffer. *flags* may be set on return from *t\_rcv* and specifies optional flags as described below.

By default, *t\_rcv* operates in synchronous mode and waits for data to arrive if none is currently available. However, if *O\_NONBLOCK* is set (via *t\_open* or *fcntl*), *t\_rcv* executes in asynchronous mode and fails if no data is available. (See *TNODATA* below.)

On return from the call, if *T\_MORE* is set in *flags*, this indicates that there is more data and the current transport service data unit (TSDU) or expedited transport service data unit (ETSDU) must be received in multiple *t\_rcv* calls. Each *t\_rcv* with the *T\_MORE* flag set indicates that another *t\_rcv* must follow immediately to get more data for the current TSU. The end of the TSU is

identified by the return of a `t_rcv` call with the `T_MORE` flag not set. If the transport provider does not support the concept of a TSDU as indicated in the *info* argument on return from `t_open` or `t_getinfo`, the `T_MORE` flag is not meaningful and should be ignored.

On return, the data returned is expedited data if `T_EXPEDITED` is set in *flags*. If the number of bytes of expedited data exceeds *nbytes*, `t_rcv` sets `T_EXPEDITED` and `T_MORE` on return from the initial call. Subsequent calls to retrieve the remaining ETSDU do not have `T_EXPEDITED` set on return. The end of the ETSDU is identified by the return of a `t_rcv` call with the `T_MORE` flag not set.

If expedited data arrives after part of a TSDU has been retrieved, receipt of the remainder of the TSDU is suspended until the ETSDU has been processed. Only after the full ETSDU has been retrieved (`T_MORE` not set) is the remainder of the TSDU available to the user.

On failure, `t_errno` may be set to one of the following:

|               |                                                                                                                                                                              |
|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [TBADF]       | The specified file descriptor does not refer to a transport endpoint.                                                                                                        |
| [TLOOK]       | An asynchronous event has occurred on this transport endpoint and requires immediate attention.                                                                              |
| [TNODATA]     | <code>O_NONBLOCK</code> was set, but no data is currently available from the transport provider. <code>errno</code> is set to <code>EAGAIN</code> .                          |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider.                                                                                                         |
| [TOUTSTATE]   | (XTI only.) The state was incorrect when this routine was called.                                                                                                            |
| [TSYSERR]     | A system error occurred during execution of this function. One possible event is a protocol error; when this is the case, <code>errno</code> is set to <code>EPROTO</code> . |

## Diagnostics

---

On successful completion, `t_rcv` returns the number of bytes received, and it returns -1 on failure, and `t_errno` is set to indicate the error.

## See also

---

`t_open(S)`, `t_snd(S)`,

## ***Standards conformance***

---

**t\_rcv** is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

# t\_rcvconnect

receive the confirmation from a connect request

## Syntax

### TLI syntax

```
cc...-lnsl_s

#include <sys/tiuser.h>

int t_rcvconnect (fd, call)
int fd;
struct t_call *call;
```

### XTI syntax

```
cc...-lxti

#include <xti.h>

int t_rcvconnect (fd, call)
int fd;
struct t_call *call;
```

## Description

This function enables a calling transport user to determine the status of a previously sent connect request and is used in conjunction with `t_connect` to establish a connection in asynchronous mode. The connection is established on successful completion of this function.

`fd` identifies the local transport endpoint where communication is established, and `call` contains information associated with the newly established connection. `call` points to a `t_call` structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

`netbuf` is described in `netbuf(FP)`. In `call`, `addr` returns the protocol address associated with the responding transport endpoint, `opt` presents any protocol-specific information associated with the connection, `udata` points to optional user data that may be returned by the destination transport user during connection establishment, and `sequence` has no meaning for this function.

The `maxlen` (see `netbuf` in `netbuf(FP)`) field of each argument must be set before issuing this function to indicate the maximum size of the buffer for each. However, *call* may be `NULL`, in which case no information is given to the user on return from `t_rcvconnect`. By default, `t_rcvconnect` executes in synchronous mode and waits for the connection to be established before returning. On return, the `addr`, `opt`, and `udata` fields reflect values associated with the connection.

If `O_NONBLOCK` is set (via `t_open` or `fcntl`), `t_rcvconnect` executes in asynchronous mode and reduces to a poll for existing connect confirmations. If none are available, `t_rcvconnect` fails and returns immediately without waiting for the connection to be established. (See `TNODATA` below.) `t_rcvconnect` must be re-issued at a later time to complete the connection establishment phase and retrieve the information returned in *call*.

## Return value

---

`t_rcvconnect` returns 0 on success and -1 on failure, and `t_errno` is set to indicate the error.

## Diagnostics

---

On failure, `t_errno` may be set to one of the following:

|               |                                                                                                                                                                                                                                                                       |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [TBADF]       | The specified file descriptor does not refer to a transport endpoint.                                                                                                                                                                                                 |
| [TBUFOVFLW]   | The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument and the connect information to be returned in <i>call</i> is discarded. The provider's state, as seen by the user, is changed to <code>DATAXFER</code> . |
| [TLOOK]       | An asynchronous event has occurred on this transport connection and requires immediate attention.                                                                                                                                                                     |
| [TNODATA]     | <code>O_NONBLOCK</code> was set, but a connect confirmation has not yet arrived.                                                                                                                                                                                      |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider.                                                                                                                                                                                                  |
| [TOUTSTATE]   | (XTI only.) The state was incorrect when this routine was called.                                                                                                                                                                                                     |
| [TSYSERR]     | A system error has occurred during execution of this function.                                                                                                                                                                                                        |

## *See also*

---

**netbuf(FP), t\_accept(S), t\_bind(S), t\_connect(S), t\_listen(S), t\_open(S)**

## *Standards conformance*

---

**t\_rcvconnect** is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

## **t\_rcvudata**

receive a data unit

### ***Syntax***

---

#### ***TLI syntax***

```
cc... -lnsl_s

#include <sys/tiuser.h>

int t_rcvudata (fd, unitdata, flags)
int fd;
struct t_unitdata *unitdata;
int *flags;
```

#### ***XTI syntax***

```
cc... -lxti

#include <xti.h>

int t_rcvudata (fd, unitdata, flags)
int fd;
struct t_unitdata *unitdata;
int *flags;
```

### ***Description***

---

This function is used in connectionless mode to receive a data unit from another transport user. *fd* identifies the local transport endpoint through which data is received, *unitdata* holds information associated with the received data unit, and *flags* is set on return to indicate that the complete data unit was not received. *unitdata* points to a *t\_unitdata* structure containing the following members:

```
struct netbuf addr; /* caller's address */
struct netbuf opt; /* options */
struct netbuf udata; /* user data returned by caller */
```

The *maxlen* field of *addr*, *opt*, and *udata* must be set before issuing this function to indicate the maximum size of the buffer for each.

On return from this call, *addr* specifies the protocol address of the sending user, *opt* identifies protocol-specific options that were associated with this data unit, and *udata* specifies the user data that was received.

By default, *t\_rcvudata* operates in synchronous mode and waits for a data unit to arrive if none is currently available. However, if *O\_NONBLOCK* is set

(via `t_open` or `fcntl`), `t_rcvudata` executes in asynchronous mode and fails if no data units are available.

If the buffer defined in the `udata` field of *unitdata* is not large enough to hold the current data unit, the buffer is filled and `T_MORE` is set in *flags* on return to indicate that another `t_rcvudata` should be issued to retrieve the rest of the data unit. Subsequent `t_rcvudata` call(s) return zero for the length of the address and options until the full data unit has been received.

On failure, `t_errno` may be set to one of the following:

|               |                                                                                                                                                                                                  |
|---------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [TBADF]       | The specified file descriptor does not refer to a transport endpoint.                                                                                                                            |
| [TBUFOVFLW]   | The number of bytes allocated for the incoming protocol address or options is not sufficient to store the information. The unit data information to be returned in <i>unitdata</i> is discarded. |
| [TLOOK]       | An asynchronous event has occurred on this transport endpoint and requires immediate attention.                                                                                                  |
| [TNODATA]     | <code>O_NONBLOCK</code> was set, but no data units are currently available from the transport provider.                                                                                          |
| [TNOTSUPPORT] | The <code>T_CLTS</code> service type is not supported by the underlying transport provider.                                                                                                      |
| [TOUTSTATE]   | (XTI only.) The state should have been <code>T_IDLE</code> when this routine was called.                                                                                                         |
| [TSYSERR]     | A system error occurred during execution of this function. One possibility, indicated by <code>errno</code> set to <code>EPROTO</code> , is a protocol error.                                    |

## Diagnostics

---

The `t_rcvudata` function returns 0 on successful completion and -1 on failure, and `t_errno` is set to indicate the error.

## See also

---

`netbuf(FP)`, `t_rcvuderr(S)`, `t_sndudata(S)`

## Standards conformance

---

`t_rcvudata` is conformant with:

AT&T SVID Issue 2;  
X/Open Portability Guide, Issue 3, 1989;  
and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).



## **t\_rcvuderr**

receive a unit data error indication

---

### ***Syntax***

---

#### ***TLI syntax***

```
cc ... -lnsl_s

#include <sys/tiuser.h>

int t_rcvuderr (fd, uerr)
int fd;
struct t_uderr *uerr;
```

#### ***XTI syntax***

```
cc ... -lxti

#include <xti.h>

int t_rcvuderr (fd, uerr)
int fd;
struct t_uderr *uerr;
```

### ***Description***

---

This function is used in connectionless mode to receive information concerning an error on a previously sent data unit, and should only be issued following a unit data error indication. It informs the transport user that a data unit with a specific destination address and protocol options produced an error. *fd* identifies the local transport endpoint through which the error report is received, and *uerr* points to a *t\_uderr* structure containing the following members:

```
 struct netbuf addr;
 struct netbuf opt;
 long error;
```

*netbuf* is described in *netbuf*(FP). The *maxlen* (see *netbuf*(FP)) field of *addr* and *opt* must be set before issuing this function to indicate the maximum size of the buffer for each.

On return from this call, the *addr* structure specifies the destination protocol address of the erroneous data unit; the *opt* structure identifies protocol-specific options that were associated with the data unit; and *error* specifies a protocol-dependent error code.

If the user does not care to identify the data unit that produced an error, *uderr* may be set to NULL and *t\_rcvuderr* simply clears the error indication without reporting any information to the user.

## Return value

---

The *t\_rcvuderr* function returns 0 on successful completion and -1 on failure, and *t\_errno* is set to indicate the error.

## Diagnostics

---

On failure, *t\_errno* may be set to one of the following:

|               |                                                                                                                                                                                                                                                                                                             |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [TBADF]       | The specified file descriptor does not refer to a transport endpoint.                                                                                                                                                                                                                                       |
| [TBUFOVFLW]   | The number of bytes allocated for the incoming protocol address or options is not sufficient to store the information. The unit data error information to be returned in <i>uderr</i> is discarded.                                                                                                         |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider.                                                                                                                                                                                                                                        |
| [TNOUDERR]    | No unit data error indication currently exists on the specified transport endpoint.                                                                                                                                                                                                                         |
| [TSYSERR]     | A system error occurred during execution of this function. One possibility is an I/O error where not all of the message was received; when this error occurs, <i>errno</i> is set to <i>EIO</i> . Another is that a protocol error occurred; when this error occurs, <i>errno</i> is set to <i>EPROTO</i> . |

## See also

---

*netbuf*(FP), *t\_rcvudata*(S), *t\_sndudata*(S)

## Standards conformance

---

*t\_rcvuderr* is conformant with:

AT&T SVID Issue 2;  
X/Open Portability Guide, Issue 3, 1989;  
and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

## **t\_rcvdis**

retrieve information from disconnect

---

### *Syntax*

---

#### *TLI syntax*

```
cc ... -lnsl_s

#include <sys/tiuser.h>

t_rcvdis (fd,discon)
int fd;
struct t_discon *discon;
```

#### *XTI syntax*

```
cc ... -lxti

#include <xti.h>

t_rcvdis (fd,discon)
int fd;
struct t_discon *discon;
```

### *Description*

---

This function is used to identify the cause of a disconnect, and to retrieve any user data sent with the disconnect. *fd* identifies the local transport endpoint where the connection existed, and *discon* points to a **t\_discon** structure containing the following members:

```
 struct netbuf udata;
 int reason;
 int sequence;
```

**netbuf** is described in *netbuf*(FP). *reason* specifies the reason for the disconnect through a protocol-dependent reason code, *udata* identifies any user data that was sent with the disconnect, and *sequence* may identify an outstanding connect indication with which the disconnect is associated. *sequence* is only meaningful when **t\_rcvdis** is issued by a passive transport user who has executed one or more **t\_listen** functions and is processing the resulting connect indications. If a disconnect indication occurs, *sequence* can be used to identify which of the outstanding connect indications is associated with the disconnect.

If a user does not care if there is incoming data and does not need to know the value of *reason* or *sequence*, *discon* may be NULL and any user data associated with the disconnect is discarded. However, if a user has retrieved more

than one outstanding connect indication (via `t_listen`) and `discon` is `NULL`, the user is unable to identify with which connect indication the disconnect is associated.

On failure, `t_errno` may be set to one of the following:

|               |                                                                                                                                                                                                                                                                |
|---------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [TBADF]       | The specified file descriptor does not refer to a transport endpoint.                                                                                                                                                                                          |
| [TBUFOVFLW]   | The number of bytes allocated for incoming data is not sufficient to store the data. The provider's state, as seen by the user, changes to <code>T_IDLE</code> , and the disconnect indication information to be returned in <code>discon</code> is discarded. |
| [TNODIS]      | No disconnect indication currently exists on the specified transport endpoint.                                                                                                                                                                                 |
| [TNOTSUPPORT] | The <code>T_CLTS</code> service type is not supported by the underlying transport provider.                                                                                                                                                                    |
| [TOUTSTATE]   | (XTI only.) The state was incorrect when this routine was called.                                                                                                                                                                                              |
| [TSYSERR]     | A system error occurred during execution of this function. One possibility is an I/O error indicated by <code>errno</code> set to <code>EIO</code> . Another possibility is a protocol error indicated by <code>errno</code> set to <code>EPROTO</code> .      |

## Diagnostics

---

The `t_rcvdis` function returns 0 on success and -1 on failure, and `t_errno` is set to indicate the error.

## See also

---

`netbuf(FP)`, `t_connect(S)`, `t_listen(S)`, `t_open(S)`, `t_snddis(S)`

## Standards conformance

---

`t_rcvdis` is conformant with:

AT&T SVID Issue 2;  
X/Open Portability Guide, Issue 3, 1989;  
and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

## **t\_rcvrel**

acknowledge receipt of an orderly release indication

---

### ***Syntax***

---

#### ***TLI syntax***

```
cc ... -lnsl_s
#include <sys/tiuser.h>

t_rcvrel (fd)
int fd;
```

#### ***XTI syntax***

```
cc ... -lxti
#include <xti.h>

t_rcvrel (fd)
int fd;
```

### ***Description***

---

This function is used to acknowledge receipt of an orderly release indication. *fd* identifies the local transport endpoint where the connection exists. After receipt of this indication, the user may not attempt to receive more data because such an attempt will block forever. However, the user may continue to send data over the connection if *t\_sndrel* has not been issued by the user.

This function is an optional service of the transport provider, and is only supported if the transport provider returned service type *T\_COTS\_ORD* on *t\_open* or *t\_getinfo*.

On failure, *t\_errno* may be set to one of the following:

|          |                                                                                                 |
|----------|-------------------------------------------------------------------------------------------------|
| [TBADE]  | The specified file descriptor does not refer to a transport endpoint.                           |
| [TLOOK]  | An asynchronous event has occurred on this transport endpoint and requires immediate attention. |
| [TNOREL] | No orderly release indication currently exists on the specified transport endpoint.             |

|               |                                                                                                                                                                                                                   |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [TNOTSUPPORT] | Without the T_COTS_ORD service type being set, this function is not supported by the underlying transport provider.                                                                                               |
| [TOUTSTATE]   | (XTI only.) The state was incorrect when this routine was called.                                                                                                                                                 |
| [TSYSERR]     | A system error occurred during execution of this function. One possibility is an I/O error and is indicated by errno being set to EIO. Another is a protocol error and is indicated by errno being set to EPROTO. |

## Diagnostics

---

The t\_rcvrel function returns 0 on success and -1 on failure, and t\_errno is set to indicate the error.

## See also

---

t\_open(S), t\_sndrel(S)

## Standards conformance

---

t\_rcvrel is conformant with:

AT&T SVID Issue 2;  
X/Open Portability Guide, Issue 3, 1989;  
and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

## **t\_snd**

send data or expedited data over a connection

---

### ***Syntax***

---

#### ***TLI syntax***

```
cc ... -lnsl_s

#include <sys/tiuser.h>

int t_snd (fd, buf, nbytes, flags)
int fd;
char *buf;
unsigned nbytes;
int flags;
```

#### ***XTI syntax***

```
cc ... -lxti

#include <xti.h>

int t_snd (fd, buf, nbytes, flags)
int fd;
char *buf;
unsigned nbytes;
int flags;
```

### ***Description***

---

This function is used to send either normal or expedited data. *fd* identifies the local transport endpoint over which data should be sent, *buf* points to the user data, *nbytes* specifies the number of bytes of user data to be sent, and *flags* specifies any optional flags described below.

By default, *t\_snd* operates in synchronous mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if *O\_NONBLOCK* is set (via *t\_open* or *fcntl*), *t\_snd* executes in asynchronous mode, and fails immediately if there are flow control restrictions.

Even when there are no flow control restrictions, *t\_snd* waits if STREAMS internal resources are not available, regardless of the state of *O\_NONBLOCK*.

On successful completion, *t\_snd* returns the number of bytes accepted by the transport provider. Normally this equals the number of bytes specified in *nbytes*. However, if *O\_NONBLOCK* is set, it is possible that only part of the

data is accepted by the transport provider. In this case, `t_snd` sets `T_MORE` for the data that was sent (see below) and returns a value less than `nbytes`. If `nbytes` is zero, no data is passed to the provider and `t_snd` returns zero.

If `T_EXPEDITED` is set in `flags`, the data is sent as expedited data, and is subject to the interpretations of the transport provider.

If `T_MORE` is set in `flags`, or is set as described above, an indication is sent to the transport provider that the transport service data unit (TSDU) or expedited transport service data unit (ETSDU) is being sent through multiple `t_snd` calls. Each `t_snd` with the `T_MORE` flag set indicates that another `t_snd` follows with more data for the current TSUD. The end of the TSUD (or ETSDU) is identified by a `t_snd` call with the `T_MORE` flag not set. Use of `T_MORE` enables a user to break up large logical data units without losing the boundaries of those units at the other end of the connection. The flag implies nothing about how the data is packaged for transfer below the transport interface. If the transport provider does not support the concept of a TSUD as indicated in the `info` argument on return from `t_open` or `t_getinfo`, the `T_MORE` flag is not meaningful and should be ignored.

The size of each TSUD or ETSDU must not exceed the limits of the transport provider as returned by `t_open` or `t_getinfo`. If the size is exceeded, a `TSYSERR` with system error `EPROTO` occurs. However, the `t_snd` may not fail because `EPROTO` errors may not be reported immediately. In this case, a subsequent call that accesses the transport endpoint fails with the associated `TSYSERR`.

If `t_snd` is issued from the `T_IDLE` state, the provider may silently discard the data. If `t_snd` is issued from any state other than `T_DATAXFER`, `T_INREL` or `T_IDLE`, the provider generates a `TSYSERR` with system error `EPROTO` (which may be reported in the manner described above).

## Return value

---

On successful completion, `t_snd` returns the number of bytes accepted by the transport provider, and it returns -1 on failure and `t_errno` is set to indicate the error.

## Diagnostics

---

On failure, `t_errno` may be set to one of the following:

|             |                                                                                                                                    |
|-------------|------------------------------------------------------------------------------------------------------------------------------------|
| [TBADDDATA] | Bad data was encountered with <code>flags</code> set to <code>T_EXPEDITED</code> .                                                 |
| [TBADF]     | The specified file descriptor does not refer to a transport endpoint.                                                              |
| [TFLOW]     | <code>O_NONBLOCK</code> was set, but the flow control mechanism prevented the transport provider from accepting data at this time. |



|               |                                                                                                        |
|---------------|--------------------------------------------------------------------------------------------------------|
| [TBADFLAG]    | The <i>flags</i> argument must be either T_END, T_EXPEDITED, T_MORE, or T_EXPEDITED Or'ed with T_MORE. |
| [TLOOK]       | A disconnect or an orderly release request is occurring on the transport endpoint.                     |
| [TNOTSUPPORT] | The use of the T_CLTS service type is not supported by the underlying transport provider.              |
| [TOUTSTATE]   | The function was issued in the wrong sequence on the transport endpoint referenced by fd               |
| [TSYSERR]     | A system error occurred during execution of this function.                                             |

### *See also*

---

**t\_open(S), t\_rcv(S),**

### *Standards conformance*

---

**t\_snd** is conformant with:

AT&T SVID Issue 2;  
X/Open Portability Guide, Issue 3, 1989;  
and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

# t\_snddis

send user-initiated disconnect request

## Syntax

### TLI syntax

```
cc... -lnsl_s

#include <sys/tiuser.h>

int t_snddis (fd, call)
int fd;
struct t_call *call;
```

### XTI syntax

```
cc... -lxti

#include <xti.h>

int t_snddis (fd, call)
int fd;
struct t_call *call;
```

## Description

This function is used to initiate an abortive release on an already established connection or to reject a connect request. *fd* identifies the local transport endpoint of the connection, and *call* specifies information associated with the abortive release. *call* points to a *t\_call* structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

*netbuf* is described in *netbuf*(FP). The values in *call* have different semantics, depending on the context of the call to *t\_snddis*. When rejecting a connect request, *call* must be non-NULL and contain a valid value of sequence to uniquely identify the rejected connect indication to the transport provider. The *addr* and *opt* fields of *call* are ignored. In all other cases, *call* need only be used when data is being sent with the disconnect request. The *addr*, *opt*, and *sequence* fields of the *t\_call* structure are ignored. If the user does not wish to send data to the remote user, the value of *call* may be NULL.

udata specifies the user data to be sent to the remote user. The amount of user data must not exceed the limits supported by the transport provider as returned by `t_open` or `t_getinfo`. If the `len` field of `udata` is zero, data is not sent to the remote user.

On failure, `t_errno` may be set to one of the following:

|               |                                                                                                                                                                                             |
|---------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [TBADF]       | The specified file descriptor does not refer to a transport endpoint.                                                                                                                       |
| [TBADDATA]    | The amount of user data specified was not within the bounds allowed by the transport provider. The transport provider's outgoing queue is flushed, so data may be lost.                     |
| [TBADSEQ]     | An invalid sequence number was specified, or a NULL call structure was specified when rejecting a connect request. The transport provider's outgoing queue is flushed, so data may be lost. |
| [TLOOK]       | An asynchronous event has occurred on this transport endpoint and requires immediate attention.                                                                                             |
| [TNOTSUPPORT] | The <code>T_CLTS</code> service type is not supported by the underlying transport provider.                                                                                                 |
| [TOUTSTATE]   | The function was issued in the wrong sequence. The transport provider's outgoing queue may be flushed, so data may be lost.                                                                 |
| [TSYSERR]     | A system error has occurred during execution of this function.                                                                                                                              |

## *Diagnostics*

---

The `t_snddis` function returns 0 on success and -1 on failure, and `t_errno` is set to indicate the error.

## *See also*

---

`netbuf(FP)`, `t_connect(S)`, `t_getinfo(S)`, `t_listen(S)`, `t_open(S)`

## *Standards conformance*

---

`t_snddis` is conformant with:

AT&T SVID Issue 2;  
X/Open Portability Guide, Issue 3, 1989;  
and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

# t\_sndrel

initiate an orderly release

## Syntax

### TLI syntax

```
cc... -lnsl_s
#include <sys/tiuser.h>

int t_sndrel (fd)
int fd;
```

### XTI syntax

```
cc... -lxti
#include <xti.h>

int t_sndrel (fd)
int fd;
```

## Description

This function is used to initiate an orderly release of a transport connection and indicates to the transport provider that the transport user has no more data to send. *fd* identifies the local transport endpoint where the connection exists. After issuing `t_sndrel`, the user may not send any more data over the connection. However, a user may continue to receive data if an orderly release indication has been received.

This function is an optional service of the transport provider and is only supported if the transport provider returned service type `T_COTS_ORD` on `t_open` or `t_getinfo`.

On failure, `t_errno` may be set to one of the following:

|         |                                                                                                                                            |
|---------|--------------------------------------------------------------------------------------------------------------------------------------------|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint.                                                                      |
| [TFLOW] | <code>O_NONBLOCK</code> was set, but the flow control mechanism prevented the transport provider from accepting the function at this time. |
| [TLOOK] | A disconnect event has occurred on this transport endpoint.                                                                                |

|               |                                                                                                                                                               |
|---------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [TOUTSTATE]   | (XTI only.) The state was incorrect when this routine was called.                                                                                             |
| [TNOTSUPPORT] | This error is generated if the T_COTS_ORD service type is not returned by a previous call to the <code>t_open(S)</code> or <code>t_getinfo(S)</code> routine. |
| [TSYSERR]     | A system error has occurred during execution of this function.                                                                                                |

## ***Diagnostics***

---

The `t_sndrel` function returns 0 on success and -1 on failure, and `t_errno` is set to indicate the error.

## ***See also***

---

`t_open(S)`, `t_rcvrel(S)`

## ***Standards conformance***

---

`t_sndrel` is conformant with:

AT&T SVID Issue 2;  
X/Open Portability Guide, Issue 3, 1989;  
and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

# t\_sndudata

send a data unit

## Syntax

### TLI syntax

```
cc... -lnsl_s

#include <sys/tiuser.h>

int t_sndudata (fd, unitdata)
int fd;
struct t_unitdata *unitdata;
```

### XTI syntax

```
cc... -lxti

#include <xti.h>

int t_sndudata (fd, unitdata)
int fd;
struct t_unitdata *unitdata;
```

## Description

This function is used in connectionless mode to send a data unit to another transport user. *fd* identifies the local transport endpoint through which data is sent, and *unitdata* points to a **t\_unitdata** structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

**netbuf** is described in *netbut(FP)*. In *unitdata*, *addr* specifies the protocol address of the destination user, *opt* identifies protocol-specific options that the user wants associated with this request, and *udata* specifies the user data to be sent. The user may choose not to specify what protocol options are associated with the transfer by setting the *len* field of *opt* to zero. In this case, the provider may use default options.

If the *len* field of *udata* is zero, no data unit is passed to the transport provider; **t\_sndudata** does not send zero-length data units.

By default, **t\_sndudata** operates in synchronous mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if **O\_NONBLOCK** is set

(via **t\_open** or **fcntl**), **t\_sndudata** executes in asynchronous mode and fails under such conditions.

If **t\_sndudata** is issued from an invalid state, or if the amount of data specified in **udata** exceeds the TSDU size as returned by **t\_open** or **t\_getinfo**, the provider generates an EPROTO protocol error. (See TSYSERR below.)

## *Return value*

---

The **t\_sndudata** function returns 0 on successful completion and -1 on failure, and **t\_errno** is set to indicate the error.

## *Diagnostics*

---

On failure, **t\_errno** may be set to one of the following:

|                      |                                                                                                                                                                                                                                               |
|----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>[TBADDDATA]</b>   | (XTI only.) Bad data was detected.                                                                                                                                                                                                            |
| <b>[TBAADF]</b>      | The specified file descriptor does not refer to a transport endpoint.                                                                                                                                                                         |
| <b>[TFLOW]</b>       | O_NONBLOCK was set, but the flow control mechanism prevented the transport provider from accepting data at this time.                                                                                                                         |
| <b>[TLOOK]</b>       | (XTI only.) A datagram error occurred on this transport endpoint.                                                                                                                                                                             |
| <b>[TNOTSUPPORT]</b> | The use of the T_CLTS service type is not supported by the underlying transport provider.                                                                                                                                                     |
| <b>[TOUTSTATE]</b>   | The function was issued in the wrong sequence on the transport endpoint referenced by <b>fd</b> .                                                                                                                                             |
| <b>[TSYSERR]</b>     | A system error occurred during execution of this function. One possibility is a protocol error and is indicated by EPROTO being set in <b>errno</b> . Another possibility is an I/O error and is indicated by EIO being set in <b>errno</b> . |

## *See also*

---

**netbuf(FP)**, **t\_rcvudata(S)**, **t\_rcvuderr(S)**

## *Standards conformance*

---

**t\_sndudata** is conformant with:

AT&T SVID Issue 2;  
X/Open Portability Guide, Issue 3, 1989;  
and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

# t\_sync

synchronize transport library

---

## Syntax

---

### TLI syntax

```
cc ... -lnsl_s

#include <sys/tiuser.h>

int t_sync (fd)
int fd;
```

### XTI syntax

```
cc ... -lxti

#include <xti.h>

int t_sync (fd)
int fd;
```

## Description

---

For the transport endpoint specified by *fd*, **t\_sync** synchronizes the data structures managed by the transport library with information from the underlying transport provider. In doing so, it can convert a raw file descriptor (obtained via **open(S)**, **dup(S)**, or as a result of a **fork(S)** and **exec(S)**) to an initialized transport endpoint, assuming that file descriptor referenced a transport provider. This function also allows two cooperating processes to synchronize their interaction with a transport provider.

For example, if a process **forks** a new process and issues an **exec**, the new process must issue a **t\_sync** to build the private library data structure associated with a transport endpoint and to synchronize the data structure with the relevant provider information.

It is important to remember that the transport provider treats all users of a transport endpoint as a single user. If multiple processes are using the same endpoint, they should coordinate their activities so as not to violate the state of the provider. **t\_sync** returns the current state of the provider to the user, thereby enabling the user to verify the state before taking further action. This coordination is only valid among cooperating processes; it is possible that a process or an incoming event could change the provider's state after a **t\_sync** is issued.



If the provider is undergoing a state transition when `t_sync` is called, the function fails.

On failure, `t_errno` may be set to one of the following:

|              |                                                                                                                                                                   |
|--------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [TBADF]      | The specified file descriptor is a valid open file descriptor but does not refer to a transport endpoint.                                                         |
| [TSTATECHNG] | The transport provider is undergoing a state change.                                                                                                              |
| [TSYSERR]    | A system error occurred during execution of this function. One possibility is an I/O error and is indicated by <code>EIO</code> being set in <code>errno</code> . |

## *Diagnostics*

---

The `t_sync` function returns the state of the transport provider on successful completion and -1 on failure, and `t_errno` is set to indicate the error. The state returned may be one of the following:

|                         |                                                                      |
|-------------------------|----------------------------------------------------------------------|
| <code>T_DATAXFER</code> | data transfer                                                        |
| <code>T_IDLE</code>     | idle                                                                 |
| <code>T_INCON</code>    | incoming connection pending                                          |
| <code>T_INREL</code>    | incoming orderly release (waiting for an orderly release request).   |
| <code>T_OUTCON</code>   | outgoing connection pending                                          |
| <code>T_OUTREL</code>   | outgoing orderly release (waiting for an orderly release indication) |
| <code>T_UNBND</code>    | unbound                                                              |

## *See also*

---

`dup(S)`, `exec(S)`, `fork(S)`, `open(S)`

## *Standards conformance*

---

`t_sync` is conformant with:

AT&T SVID Issue 2;  
X/Open Portability Guide, Issue 3, 1989;  
and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

# t\_unbind

---

disable a transport endpoint

## Syntax

---

### TLI syntax

```
cc ... -lnsl_s

#include <sys/tiuser.h>

int t_unbind (fd)
int fd;
```

### XTI syntax

```
cc ... -lxti

#include <xti.h>

int t_unbind (fd)
int fd;
```

## Description

---

The **t\_unbind** function disables the transport endpoint specified by *fd*, which was previously bound by **t\_bind**. On completion of this call, no further data or events destined for this transport endpoint is accepted by the transport provider.

On failure, **t\_errno** may be set to one of the following:

- |             |                                                                                    |
|-------------|------------------------------------------------------------------------------------|
| [TBADF]     | The specified file descriptor does not refer to a transport endpoint.              |
| [TLOOK]     | An asynchronous event has occurred on this transport endpoint.                     |
| [TOUTSTATE] | The function was issued in the wrong sequence; the state should be <b>T_IDLE</b> . |
| [TSYSERR]   | A system error occurred during execution of this function.                         |

## Diagnostics

---

The **t\_unbind** function returns 0 on success and -1 on failure, and **t\_errno** is set to indicate the error.

## ***See also***

---

**t\_bind(S)**

## ***Standards conformance***

---

**t\_unbind** is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

and Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2).

# tai\_end, tai\_get, tai\_init

get MMDF site tailoring information

## Syntax

```
cc ... -lmmdf

tai_end ()

tai_get (maxargs, argv)
int maxargs;
char *argv[];

tai_init (file)
char file[];
```

## Description

The **tai** package is designed to acquire site-dependent information, from an external, clear-text file. It supports a simple file syntax.

### File format

Tailoring information comprises a set of lines. Lines comprise a set of fields. Usually the first field on the line is treated as the name of the line. Fields may be separated by a space, horizontal tab, comma, semicolon, colon, or an equal sign. These separators are treated similarly. However, a space and a tab are skipped at the beginning of a field. Also, if two fields are separated by an equal sign, they are assumed to be a key/value pair and are treated somewhat differently by **tai\_get**.

Fields are formed according to a syntax that is similar to C programs. A sequence of printable, nonspecial characters is a field. To include a special character, such as a space or a comma, the character must be preceded by a backslash. The sequence backslash octal-string specifies a single eight-bit character (for example, \040 indicates space). If a field is surrounded by quotation marks, only the quotation mark and the backslash are treated as special.

The following shows how different argument lines are parsed. A file containing the following is parsed into four sets of argument lists:

```
loc UDel-Relay
que /usr/mmdf/que/home
chan a,, arpa\040net\ foo=bar
chan b backup blat='gelp,blat=boop'
```

The parsed argument lists are:

```
'loc' 'UDel-Relay'
'que' '/usr/mmdf/que/home'
'chan' 'a' '' '=' 'arpa net foo' 'bar'
'chan' 'b' 'backup' '=' 'blat' 'gelp,blat=boop'
```

The formal syntax for a tailoring file is given below, in modified BNF. An asterisk indicates a repetition of zero or more times. Parentheses surround alternatives.

```
file = *line
line = *field '\n'
field = *(SPACE / TAB) value *value SPECIAL
value = CHAR / echar / qstring
qstring = ''' *qvalue '''
qvalue = CHAR / DELIM / echar
echar = "\\\" (SPECIAL / FORMAT / octal)
SPECIAL = DELIM / ''' / "\\\"
DELIM = SPACE / TAB / ',' / ';' / ':'
CHAR = {Any ASCII character not a SPECIAL}
FORMAT = 'n' / 'r' / 'f' / 'b'
octal = ochar [ochar [ochar]]
ochar = {Digit 0-7}
```

## *Procedure calls*

The `tai_init` call initializes the tailoring package. Its argument is the name of the file that contains tailoring information. Since the name is not built into the package, it is possible to have several different tailoring files. The `tai_init` call returns 0 on normal termination.

The `tai_get` call acquires and parses the next line of the tailoring file. The first argument to the procedure is the maximum number of tailoring arguments (fields) that can be accepted from the line. The second procedure argument is an array which will hold character pointers to the tailoring arguments. The list of pointers is terminated by a zero-valued pointer.

The `tai_get` call returns the actual number of arguments that were parsed and returns 0 when it has reached the end of the file. The data are in permanent storage, so that they do not have to be copied before the next call to `tai_get`.

The argument list is built in the same sequence as the fields are listed in the argument line. Separator characters are not returned, except for an equal-sign. When an equal-sign is used, three entries are consumed in the argument pointer array. The first entry points to the string "=", the second entry is for the key and the third entry is for the value. (An equal-sign is treated as an infix operator, but the parsed fields are stored in prefix form.)

When initialization is complete, call `tai_end` for cleanup. The `tai_end` call returns 0 on normal termination.

## Usage

The user program calls `tai_init` and then calls `tai_get` until all of the tailoring information has been processed. If there are several different software modules that need to be initialized, then the information returned by `tai_get` can be passed consecutively to the modules' initialization routines, until one of the routines returns a value indicating that it has "consumed" the information. The following code demonstrates the use of this package:

```
#include "util.h"

main ()
{
 char *taiargs[20];
 int nargs;
 register int ind;

 if (tai_init ("init.tai") < OK)
 {
 perror ("tai_init"); /* initialize */
 exit (-1);
 }

 while ((nargs = tai_get (20, taiargs)) > 0)
 for (ind = 0; ind < nargs; ind++)
 {
 if (user_init (taiargs)) /* who uses it? */
 continue; /* user-level information */
 if (util_init (taiargs))
 continue; /* the utilities package */
 /* was not consumed; ignore it */
 }

 if (nargs < 0) /* we had some problem */
 perror ("tai_get");
 tai_end ();
 exit ((nargs < 0) ? -1 : 0);
}
```

## Diagnostics

---

All routines return -1 on error.

## See also

---

`llog(S)`, `ml_send(S)`, `mmdf(S)`, `phs(S)`

## Standards conformance

---

The `tai_init`, `tai_get`, and `tai_end` calls are not part of any currently supported standard; they were developed at the University of Delaware and are used by permission.

# **tam**

TAM transition libraries

---

## ***Syntax***

---

```

cc ... -ltam -lcurses

#include <tam.h>

int addch(c)
char c;

int addstr(s)
char *s;

int adf_gttok (ptr, tbl)
char *ptr;
struct s_kwtbl *tbl;

char *adf_gtwrd (sptr, dptr)
char *sptr, *dptr;

char *adf_gtxcd (sptr, dptr)
char *sptr, *dptr;

int attroff(attrs)
chtype attrs;

int attron(attrs)
chtype attrs;

int baudrate()
int beep()
int cbreak()
int clear()

int clearok(dum1, dum2)
int dum1, dum2;

int clrtoebot()
int clrtoeol()
int delch()
int deleteln()
int echo()
int endwin()

int erase()

int exhelp (hfile, htitle)
char *hfile, *htitle;

int fixterm()

```

```

int flash()

int flushing()

int form (form, op)
form_t *form;
int op;

int getch()

int getyx(win, r, c)
int win, r, c;

int initscr()

int insch(ch)
char ch;

int insertln()
int iswind()

char *kcodemap (code)
unsigned char code;

int keypad (dummy, flag)
int dummy, flag;

leaveok(dum1, dum2)
int dummy;

int menu (menu, op)
menu_t *menu;
int op;

int message (mtype, hfile, htitle, format [, arg ...])
int mtype;
char *hfile, *htitle, *format;

int move(r, c)
int r, c;

int mvaddch(r, c, ch)
int r, c;
char ch;

int mvaddstr(r, c, s)
int r, c;
char *s;

int mvinch(r, c)
int r, c;

nl()

int nocbreak()

```



```

int nodelay(dummy, bool)
int dummy, bool;

int noecho()

nonl()

int pb_check (stream)
FILE *stream;

int pb_empty (stream)
FILE *stream;

int pb_gbuf (ptr, n, fn, stream)
char *ptr;
int n;
int (*fn) ();
FILE *stream;

char *pb_gets (ptr, n, stream)
char *ptr;
int n;
FILE *stream;

char *pb_name()

FILE *pb_open()

int pb_puts (ptr, stream)
char *ptr;
FILE *stream;

int pb_seek (stream)
FILE *stream;

int pb_wEOF (stream)
FILE *stream;

int printw (fmt[, arg1 ... argn])
char *fmt;

int refresh()
int resetterm()
int resetty()
int savetty()

int track (w, trk, op, butptr, whyptr)
int w, op, *butptr, *whyptr;
track_t *trk;

int wcmd (wn, cp)
short wn;
char *cp;

int wcreate (row, col, height, width, flags)
short row, col, height, width;

```

```

unsigned short flags;

int wdelete (wn)
short wn;

void wexit(ret)
int ret;

int wgetc (wn)
short wn;

int wgetmouse (wn, ms)
short wn;
struct umdata *ms;

int wgetpos (wn, rowp, colp)
short wn;
int *rowp, *colp;

int wgetsel()

int wgetstat (wn, wstatp)
short wn;
WSTAT *wstatp;

int wgoto (wn, row, col)
short wn, row, col;

void wicoff (wn, row, col, icp)
short wn, row, col;
struct icon *icp;

void wicon (wn, row, col, icp)
short wn, row, col;
struct icon *icp;

int wind (type, height, width, flags, pfont)
int type, height, width;
short flags;
char *pfont[];

void winit()

int wlabel (wn, cp)
short wn;
char *cp;

int wndelay (wn, bool)
int wn, bool;

void wnl (wn, flag)
short wn;
int flag;

int wpostwait()

```

```

int wprexec()

int wprintf (wn, fmt[, arg1 ... argn])
short wn;
char *fmt;

int wprompt (wn, cp)
short wn;
char *cp;

int wputc (wn, c)
short wn;
char c;

int wputs (wn, cp)
short wn;
char *cp;

int wrastop (w, srcbase, srcwidth, dstbase,
 dstwidth, srcx, srcy, dstx,
 dsty, width, height, srcop,
 dstop, pattern)

int w;
unsigned short *srcbase, *dstbase, *pattern;
unsigned short srcwidth, dstwidth, width, height;
unsigned short srcx, srcy, dstx, dsty;
char srcop, dstop;

int wreadmouse (wn, xp, yp, bp, rp)
short wn;
int *xp, *yp, *bp, *rp;

int wrefresh (wn)
short wn;

int wselect (wn)
short wn;

int wsetmouse (wn, ms)
short wn;
struct umdata *ms;

int wsetstat (wn, wstatp)
short wn;
WSTAT *wstatp;

int wslk (wn, 0, slong1, slong2, sshort)
short wn;
char *slong1, *slong2, *sshort;

int wslk (wn, kn, llabel, slabel)
short wn, kn;
char *llabel, *slabel;

int wuser (wn, cp)

```

```
short wn;
char *cp;
```

## Description

---

These routines allow you to run character mode applications that run under the Terminal Access Method (TAM) on any display device supported by the `curses(S)` library.

Once you change a TAM program to remove machine-specific code, you can then recompile it including the standard TAM header file `tam.h` and link it with the `tam` and `curses` libraries.

The TAM transition library translates between TAM function calls and `curses` functions. For example, the TAM function `wcreate()` is mapped to the conversion library function `TAMwcreate()`, which consists of a series of `curses` calls, such as `newwin()` and `subwin()`.

To enable the code in your TAM application program to run smoothly under `curses`, you should do the following:

- remove code that would be executed if a low-level `iswind()` function call returned a non-zero value, i.e., *true*. Under the TAM transition library, `iswind()` always returns *false*.
- remove all TAM calls to mouse management routines and the calls `wicon()`, `wicoff()`, and `wrastop()`, because they will translate to null operations.
- remove all machine-specific code, because the TAM transition library does not translate system calls specifically tailored to the UNIX System PC or calls [such as `ioctl(S)`] that have no meaning under `curses`. These calls fail under the TAM transition library on all machines except the UNIX System PC.
- note that all calls to `track(S)` map to the low-level function `wgetc()`.
- remove all references to TAM calls that bear the same name as `curses` calls because calls that have the same names in both systems have different effects.
- remove all arbitrary ANSI escape sequences for display output. For example, the TAM transition library does not recognize the escape sequence used on the UNIX System PC in the command `echo "\033[J"`, which clears the screen. Instead, you should use equivalent `curses` routines (here, `clear()`).

Eliminating the superfluous code in the first three cases reduces your program's size and execution time.

## Functions

The following table summarizes the translation of TAM function calls to **curses** functions. You can use this table as a guide if you want to rewrite your TAM application to make **curses** calls directly and run more efficiently.

Routines marked with a dagger (†) are macros and don't return any value.

| TAM Function | curses(S) Equivalent                                                  |
|--------------|-----------------------------------------------------------------------|
| addch()†     | Calls <b>addch()</b> .                                                |
| addstr()†    | Calls <b>addstr()</b> .                                               |
| attron()     | Calls <b>attron()</b> .                                               |
| attroff()    | Calls <b>attroff()</b> .                                              |
| baudrate()   | Calls <b>baudrate()</b> .                                             |
| beep()       | Calls <b>beep()</b> .                                                 |
| cbreak()     | Calls <b>cbreak()</b> .                                               |
| clear()      | Calls <b>clear()</b> .                                                |
| clearok()†   | Null operation.                                                       |
| clrtoebot()  | Calls <b>clrtoebot()</b> .                                            |
| clrtoeol()   | Calls <b>clrtoeol()</b> .                                             |
| delch()      | Calls <b>delch()</b> .                                                |
| deleteln()   | Calls <b>deleteln()</b> .                                             |
| echo()       | Calls <b>echo()</b> .                                                 |
| erase()†     | Calls <b>erase()</b> .                                                |
| endwin()     | Calls <b>endwin()</b> .                                               |
| fixterm()    | Calls <b>fixterm()</b> .                                              |
| flash()†     | Calls <b>flash()</b> .                                                |
| flushinp()   | Calls <b>flushinp()</b> .                                             |
| getch()      | Calls <b>getch()</b> .                                                |
| getyx()†     | Calls <b>wgetyx()</b> .                                               |
| initscr()    | Calls <b>initscr()</b> .                                              |
| insch()      | Calls <b>insch()</b> .                                                |
| insertln()   | Calls <b>insertln()</b> .                                             |
| iswind()     | Returns <b>FALSE</b> .                                                |
| kcodemap()   | This functionality is not necessary. Remove this from your code.      |
| keypad()     | Calls <b>keypad()</b> .                                               |
| leaveok()†   | Null operation.                                                       |
| move()†      | Calls <b>move()</b> .                                                 |
| mvaddch()†   | Calls <b>move()</b> and <b>addch()</b> .                              |
| mvaddstr()†  | Calls <b>move()</b> and <b>addstr()</b> .                             |
| mvinch()     | Calls <b>move()</b> and <b>inch()</b> .                               |
| nl()†        | The functionality of this routine is not supported by <b>curses</b> . |
| nocbreak()   | Calls <b>nocbreak()</b> .                                             |
| nodelay()    | Calls <b>nodelay()</b> .                                              |
| noecho()     | Calls <b>noecho()</b> .                                               |

*(Continued on next page)*

(Continued)

| TAM Function | curses(S) Equivalent                                                                                                           |
|--------------|--------------------------------------------------------------------------------------------------------------------------------|
| nonl()†      | The functionality of this routine is not supported by curses.                                                                  |
| printw()     | Calls <code>printw()</code> .                                                                                                  |
| refresh()†   | Calls <code>refresh()</code> .                                                                                                 |
| resetterm()  | Calls <code>resetterm()</code> .                                                                                               |
| resetty()    | Calls <code>resetty()</code> .                                                                                                 |
| savetty()    | Calls <code>savetty()</code> .                                                                                                 |
| track()      | Calls <code>wgetch()</code> .                                                                                                  |
| wcmd()       | Copies the character string passed by <code>wcmd()</code> to the bottom of the screen.                                         |
| wcreate()    | Calls <code>newwin()</code> or <code>new_panel()</code> .                                                                      |
| wdelete()    | Calls <code>delwin()</code> or <code>del_panel()</code> .                                                                      |
| wexit()      | Calls <code>endwin()</code> and <code>exit()</code> .                                                                          |
| wgetsel()    | Calls <code>top_panel()</code> or <code>bottom_panel()</code> with NULL pointer.                                               |
| wgetstat()   | Calls <code>getyx()</code> , <code>getmaxyx()</code> , or <code>getbegyx()</code> .                                            |
| wgetc()      | Calls <code>wgetch()</code> . Character translation from curses to ANSI may be required, depending on the current keypad mode. |
| wgetpos()    | Calls <code>getyx()</code> .                                                                                                   |
| wgoto()      | Calls <code>wmove()</code> .                                                                                                   |
| winit()      | Calls <code>initscr()</code> .                                                                                                 |
| wlabel()     | The character string is printed in the upper left corner of the specified window.                                              |
| wndelay()    | Calls <code>nodelay()</code> .                                                                                                 |
| wselect()    | Calls <code>touchwin()</code> and <code>wrefresh()</code> , then updates the list of windows to indicate the new ordering.     |
| wsetstat()   | Calls <code>del_panel()</code> , then <code>new_panel()</code> .                                                               |
| wputc()      | Calls <code>waddch()</code> .                                                                                                  |
| wputs()      | Calls <code>waddstr()</code> .                                                                                                 |
| wprintf()    | Calls <code>wprintw()</code> .                                                                                                 |
| wprompt()    | The character string passed by <code>wprompt()</code> is copied to the bottom of the screen.                                   |
| wrefresh()   | Calls <code>wrefresh()</code> . If the window index is -1, all windows should be refreshed in the appropriate order.           |
| wslk()       | Creates small window at bottom and uses curses routines with <code>wprintw()</code> .                                          |
| wuser()      | This functionality is not necessary. Remove this from your code.                                                               |
| wsetmouse()  | This is a null operation.                                                                                                      |
| wgetmouse()  | This is a null operation.                                                                                                      |
| wreadmouse() | This is a null operation.                                                                                                      |
| wprexec()    | Calls <code>erase()</code> and <code>refresh()</code> .                                                                        |
| wpostwait()  | Calls <code>wrefresh()</code> for each window in the window list.                                                              |
| wnl()        | The functionality of this routine is not supported by curses.                                                                  |

(Continued on next page)

*(Continued)*

| TAM Function | curses(S) Equivalent      |
|--------------|---------------------------|
| wicon()      | This is a null operation. |
| wicoff()     | This is a null operation. |
| wrastop()    | This is a null operation. |

Because the high-level TAM functions in the previous table make calls only to the low-level functions in the previous table, you can continue to use those high-level TAM functions in your application programs as well. However, with **curses** you cannot use other TAM high-level functions such as **wtargeton()**.

#### Usable TAM high-level functions

|            |             |             |
|------------|-------------|-------------|
| form()     | menu()      | message()   |
| pb_empty() | pb_gets()   | adf_gttok() |
| pb_open()  | pb_check()  | pb_seek()   |
| pb_name()  | pb_puts()   | pb_wEOF()   |
| pb_gbuf()  | adf_gtwrd() | adf_gtxcd() |
| wind()     | exhelp()    |             |

Both TAM and **curses** use a set of virtual function keys that translate between an escape character sequence entered at the keyboard and a bit pattern inside the machine. Under the TAM transition library, the TAM virtual key values are translated into **curses** virtual key values.

The following table lists these equivalent virtual key values. The escape sequence listed in the left column generates the corresponding TAM virtual function key value given in the middle column. The right column lists the **curses** equivalent of the TAM virtual key and is for reference only.

| TAM Escape Sequence | Virtual Key Value |           |
|---------------------|-------------------|-----------|
|                     | TAM               | curses    |
| ESC-!               | s_F1              | KEY_F(8)  |
| ESC-@               | s_F2              | KEY_F(9)  |
| ESC-#               | s_F3              | KEY_F(10) |
| ESC-\$              | s_F4              | KEY_F(11) |
| ESC-%               | s_F5              | KEY_F(12) |
| ESC-^               | s_F6              | KEY_F(13) |
| ESC-&               | s_F7              | KEY_F(14) |
| ESC-*               | s_F8              | KEY_F(15) |
| ESC-f1              | PF1               | KEY_F(16) |
| ESC-f2              | PF2               | KEY_F(17) |
| ESC-f3              | PF3               | KEY_F(18) |
| ESC-f4              | PF4               | KEY_F(19) |

*(Continued on next page)*

(Continued)

| TAM Escape Sequence | Virtual Key Value |              |
|---------------------|-------------------|--------------|
|                     | TAM               | curses       |
| ESC-f5              | PF5               | KEY_F(20)    |
| ESC-f6              | PF6               | KEY_F(21)    |
| ESC-f7              | PF7               | KEY_F(22)    |
| ESC-f8              | PF8               | KEY_F(23)    |
| ESC-f9              | PF9               | KEY_F(24)    |
| ESC-f0              | PF10              | KEY_F(25)    |
| ESC-f-              | PF11              | KEY_F(26)    |
| ESC-f=              | PF12              | KEY_F(27)    |
| ESC-1               | F1                | KEY_F(0)     |
| ESC-2               | F2                | KEY_F(C)     |
| ESC-3               | F3                | KEY_F(S)     |
| ESC-4               | F4                | KEY_F(S)     |
| ESC-5               | F5                | KEY_F(F)     |
| ESC-6               | F6                | KEY_F(M)     |
| ESC-7               | F7                | KEY_F(6)     |
| ESC-8               | F8                | KEY_F(7)     |
| ESC-bg              | Beg               | KEY_BEG      |
| ESC-BG              | s_Beg             | KEY_SBEG     |
| ESC-br              | Break             | KEY_BREAK    |
| ESC-bw              | Back              | KEY_LEFT     |
| ESC-BW              | s_Back            | KEY_SLEFT    |
| ESC-ce              | Clear             | KEY_CLEAR    |
| ESC-CE              | Clear             | KEY_CLEAR    |
| ESC-ci              | ClearLine         | KEY_EOL      |
| ESC-CI              | s_ClearLine       | KEY_SEOL     |
| ESC-cl              | Close             | KEY_CLOSE    |
| ESC-CL              | Close             | KEY_CLOSE    |
| ESC-cm              | Cmd               | KEY_COMMAND  |
| ESC-CM              | s_Cmd             | KEY_SCOMMAND |
| ESC-cn              | Cancl             | KEY_CANCEL   |
| ESC-CN              | s_Cancl           | KEY_SCANCEL  |
| ESC-cp              | Copy              | KEY_COPY     |
| ESC-CP              | s_Copy            | KEY_SCOPY    |
| ESC-cr              | Creat             | KEY_CREATE   |
| ESC-CR              | s_Creat           | KEY_SCREATE  |
| ESC-dc              | DleteChar         | KEY_DC       |
| ESC-Del             | DleteChar         | KEY_DC       |
| ESC-DC              | s_DleteChar       | KEY_SDC      |
| ESC-dl              | Dlete             | KEY_DL       |
| ESC-DL              | s_Dlete           | KEY_SDL      |
| ESC-dn              | Down              | KEY_DOWN     |
| ESC-DN              | RollDn            | KEY_SF       |
| ESC-en              | End               | KEY_END      |

(Continued on next page)



(Continued)

| TAM Escape Sequence | Virtual Key Value |               |
|---------------------|-------------------|---------------|
|                     | TAM               | courses       |
| ESC-EN              | s_End             | KEY_SEND      |
| ESC-ESC             | Esc               | none          |
| ESC-ex              | Exit              | KEY_EXIT      |
| ESC-EX              | s_Exit            | KEY_SEXIT     |
| ESC-fi              | Find              | KEY_FIND      |
| ESC-FI              | s_Find            | KEY_SFIND     |
| ESC-fw              | Forward           | KEY_RIGHT     |
| ESC-FW              | s_Forward         | KEY_SRIGHT    |
| ESC-hl              | Help              | KEY_HELP      |
| ESC-?               | Help              | KEY_HELP      |
| ESC-HL              | s_Help            | KEY_SHELP     |
| ESC-hm              | Home              | KEY_HOME      |
| ESC-HM              | s_Home            | KEY_SHOME     |
| ESC-im              | InputMode         | KEY_IC        |
| ESC-NJ              | s_InputMode       | KEY_SIC       |
| ESC-mk              | Mark              | KEY_MARK      |
| ESC-MK              | Slect             | KEY_SELECT    |
| ESC-ms              | Msg               | KEY_MESSAGE   |
| ESC-MS              | s_Msg             | KEY_SMESSAGE  |
| ESC-mv              | Move              | KEY_MOVE      |
| ESC-MV              | s_Move            | KEY_SMOVE     |
| ESC-nx              | Next              | KEY_NEXT      |
| ESC-NX              | s_Next            | KEY_SNEXT     |
| ESC-op              | Open              | KEY_OPEN      |
| ESC-OP              | Close             | KEY_CLOSE     |
| ESC-ot              | Opts              | KEY_OPTIONS   |
| ESC-OT              | s_Opts            | KEY_SOPTIONS  |
| ESC-pg              | Page              | KEY_NPAGE     |
| ESC-PG              | s_Page            | KEY_PPAGE     |
| ESC-pr              | Print             | KEY_PRINT     |
| ESC-PR              | s_Print           | KEY_SPRINT    |
| ESC-pv              | Prev              | KEY_PREVIOUS  |
| ESC-PV              | s_Prev            | KEY_SPREVIOUS |
| ESC-rd              | RollDn            | KEY_SF        |
| ESC-RD              | RollDn            | KEY_SF        |
| ESC-re              | Ref               | KEY_REFERENCE |
| ESC-RE              | Rstrt             | KEY_RESTART   |
| ESC-rf              | Rfrsh             | KEY_REFRESH   |
| ESC-RF              | Clear             | KEY_CLEAR     |
| ESC-rm              | Rsume             | KEY_RESUME    |
| ESC-RM              | s_Rsume           | KEY_SRSUME    |
| ESC-ro              | Redo              | KEY_REDO      |
| ESC-RO              | s_Redo            | KEY_SREDO     |

(Continued on next page)

(Continued)

| TAM Escape Sequence | Virtual Key Value |               |
|---------------------|-------------------|---------------|
|                     | TAM               | curses        |
| ESC-rp              | Rplac             | KEY_REPLACE   |
| ESC-RP              | s_Rplac           | KEY_SREPLACE  |
| ESC-rs              | Rstrt             | KEY_REFERENCE |
| ESC-RS              | Rstrt             | KEY_RESTART   |
| ESC-ru              | RollUp            | KEY_SR        |
| ESC-RU              | RollUp            | KEY_SR        |
| ESC-sl              | Slect             | KEY_SELECT    |
| ESC-SL              | Slect             | KEY_SELECT    |
| ESC-ss              | Suspd             | KEY_SUSPEND   |
| ESC-SS              | s_Suspd           | KEY_SSUSPEND  |
| ESC-sv              | Save              | KEY_SAVE      |
| ESC-SV              | s_Save            | KEY_SSAVE     |
| ESC-ud              | Undo              | KEY_UNDO      |
| ESC-UD              | s_Undo            | KEY_SUNDO     |
| ESC-up              | Up                | KEY_UP        |
| ESC-UP              | RollUp            | KEY_SR        |

Some keyboards have one or more keys that emit escape sequences that are identical to the UNIX System PC keyboard sequences but do not match in terms of functionality. The function of an operationally incompatible key will always map to its **terminfo** specification. The TAM specific function implied by the same escape sequence will be accessible through the technique described above. Mechanisms in **curses(S)** automatically handle timing conflicts between actual keyboard function keys and UNIX System PC keyboard escape sequences.

## See also

---

**curses(S), field(S), fieldtype(S), form(S), item(S), menu(S), panel(S)**

## Standards conformance

---

**tam** is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

# tcgetattr, tcsetattr

state functions

## Syntax

```
cc ... -lc

#include <termios.h>

int tcgetattr (fildes, termios_p);
int fildes;
struct termios *termios_p;

int tcsetattr (fildes, optional_actions, termios_p);
int fildes, optional_actions;
struct termios *termios_p;
```

## Description

The **tcgetattr** function gets the parameters associated with the object referred to by *fildes* and stores them in the **termios** structure referenced by *termios\_p*. This function is allowed from a background process; however the terminal attributes may subsequently be changed by a foreground process. The *fildes* argument is an open file descriptor associated with a terminal. If the terminal device does not support split baud rates, the input baud rate stored in the **termios** structure is set to zero.

The **tcsetattr** function sets the parameters associated with the terminal referred to by the open file descriptor *fildes* and stores them in the **termios** structure referenced by *termios\_p* as follows:

1. If *optional\_actions* is **TCSANOW**, the change occurs immediately.
2. If *optional\_actions* is **TCSADRAIN**, the change occurs after all output written to *fildes* has been transmitted. Use **tcsetattr** when changing parameters that affect output.
3. If *optional\_actions* is **TCSAFLUSH**, the change occurs after all output written to *fildes* has been transmitted, and all input so far received but not read is discarded before the change is made.

The symbolic constants for the values of the *optional\_actions* are defined in *<termios.h>*.

If the output baud rate stored in the **termios** structure pointed to by *termios\_p* is the zero baud rate, B0, the modem control lines are no longer asserted. Normally this disconnects the line.

If the input baud rate stored in the `termios` structure pointed to by `termios_p` is zero, the input baud rate given to the hardware is the same as the output baud rate stored in the `termios` structure.

If `POSIX_JOB_CONTROL` is defined, attempts to use the `tcsetattr` routine from a process that is a member of a background process group on a *fildev* associated with its controlling terminal, cause the process group to be sent a `SIGTTOU` signal. If the calling process is blocking or ignoring `SIGTTOU` signals, the process is allowed to perform the operation, and no signal is sent.

## Note

---

If trying to change baud rates, applications should first call `tcsetattr` to determine what baud rates were actually selected, and then call `tcgetattr`.

## Return value

---

Upon successful completion, these routines return a value of 0. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

## Errors

---

If any of the following conditions occur, `tcgetattr` returns -1 and sets `errno` to the corresponding value:

[EBADF] The *fildev* argument is not a valid file descriptor.

[ENOTTY] The file associated with *fildev* is not a terminal.

If any of the following conditions occur, `tcsetattr` returns -1 and sets `errno` to the corresponding value:

[EBADF] The *fildev* argument is not a valid file descriptor.

[EINVAL] The *optional\_actions* argument is not a proper value, or an attempt was made to change an attribute represented in the `termios` structure to an unsupported value.

[ENOTTY] The file associated with *fildev* is not a terminal.

## See also

---

`tcflow(S)`

## Standards conformance

---

`tcgetattr` and `tcsetattr` are conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

and IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1).

## tcdrain, tcflow, tcflush, tcsendbreak

---

line control functions

### Syntax

---

```
cc ... -lc

#include <termios.h>

int tcdrain (fildes)
int fildes;

int tcflow (fildes, action)
int fildes, action;

int tcflush (fildes, queue_selector)
int fildes, queue_selector;

int tcsendbreak (fildes, duration)
int fildes, duration;
```

### Description

---

**tcdrain** - wait until all output has been transmitted

**tcflow** - suspend transmission or reception of data

**tcflush** - discard all data not written or read

**tcsendbreak** - transmit zero-valued bits for specified duration

If the terminal is using asynchronous serial data transmission, the **tcsendbreak** function causes transmission of a continuous stream of zero-valued bits for a specific duration. If *duration* is zero, **tcsendbreak** transmits zero-valued bits for at least .25 seconds and not more than .5 seconds. If *duration* is not zero, **tcsendbreak** sends zero-valued bits for an amount of time dependent on the baud rate and the line discipline. Refer to **ttiocom(K)** for more information on the **TCSBRK** value that **tcsendbreak** calls.

If the terminal is not using asynchronous serial data transmission, it is implementation defined whether the **tcsendbreak** function sends data to generate a break condition (as defined by the implementation) or returns without taking any action.

The **tcdrain** function waits until all output written to *fildes* has been transmitted.

The **tcflush** function discards all data written to *fildev* but not transmitted, or data received but not read, depending on the value of *queue\_selector*:

- If *queue\_selector* is **TCIFLUSH**, it flushes data received but not read.
- If *queue\_selector* is **TCOFLUSH**, it flushes data written but not transmitted.
- If *queue\_selector* is **TCIOFLUSH**, it performs both actions.

The **tcflow** function suspends transmission or reception of data on the object referred to by *fildev*, depending on the value of *action*. Possible values for *action* include:

- If *action* is **TCOOFF**, output is suspended.
- If *action* is **TCOON**, output is restarted.
- If *action* is **TCIOFF**, a **STOP** character is transmitted, which is intended to cause the terminal to stop transmitting data to the system.
- If *action* is **TCION**, a **START** character is transmitted, which is intended to cause the terminal to start transmitting data to the system.

The symbolic constants for the values of *queue\_selector* and *action* are defined in *<termios.h>*. The default on open of a terminal file is that neither its input nor its output is suspended.

## Note

If **POSIX\_JOB\_CONTROL** is defined, attempts to use **tcdrain**, **tcflow**, **tcflush**, or **tcsendbreak** from a process that is a member of a background process group on a *fildev* associated with its controlling terminal, cause the process group to be sent a **SIGTTOU** signal. If the calling process is blocking or ignoring **SIGTTOU** signals, the process is allowed to perform the operation, and no signal is sent.

## Return value

Upon successful completion, these routines return a value of zero. Otherwise, a value of -1 is returned and **errno** is set to indicate the error. Possible error conditions include:

- tcdrain:**
- [EBADF] The *fildev* argument is not a valid file descriptor.
  - [EINTR] A signal interrupted the function.
  - [ENOTTY] The file associated with *fildev* is not a terminal.

**tcflow:**    [EBADF]   The *fildes* argument is not a valid file descriptor.  
             [EINVAL]   The *action* argument is not a proper value.  
             [ENOTTY]   The file associated with *fildes* is not a terminal.

**tcflush:**   [EBADF]   The *fildes* argument is not a valid file descriptor.  
             [EINVAL]   The *queue\_selector* argument is not a proper value.  
             [ENOTTY]   The file associated with *fildes* is not a terminal.

**tcsendbreak:**  
             [EBADF]   The *fildes* argument is not a valid file descriptor.  
             [ENOTTY]   The file associated with *fildes* is not a terminal.

## *See also*

---

**tcattr(S), tticom(K)**

## *Standards conformance*

---

**tcdrain, tcflow, tcflush and tcsendbreak** are conformant with:

AT&T SVID Issue 2;  
X/Open Portability Guide, Issue 3, 1989;  
and IEEE POSIX Std 1003.1-1990 System Application Program Interface (API)  
[C Language] (ISO/IEC 9945-1).

# tcgetpgrp, tcsetpgrp

process group id functions

## Syntax

```
cc ... -lc

#include <sys/types.h>

pid_t tcgetpgrp (fildes);
int fildes;

int tcsetpgrp (fildes, pgrp_id);
int fildes;
pid_t pgrp_id;
```

## Description

**tcgetpgrp** - gets process group ID for tty foreground process

**tcsetpgrp** - sets the foreground process ID group

These routines identify and set the parent process group ID when job control is defined. The **tcgetpgrp** function returns the value of the process group ID of the foreground process group associated with the terminal.

**tcgetpgrp** is allowed only from a process that is part of a background process group. However, this information can be changed by a process that is part of the foreground process group by means of the **tcsetpgrp** call.

**tcsetpgrp** sets the foreground process ID group associated with the terminal to the value of *pgrp\_id*. *fildes* must be the file associated with the controlling terminal of the calling process. The controlling terminal must also be currently associated with the session of the calling process. The value of *pgrp\_id* must match a process group ID of a process in the same session as the calling process. Any other value of *pgrp\_id* causes an error.

## Return value

If successful, **tcgetpgrp** returns the process ID of the foreground process group associated with the calling terminal. Otherwise, -1 is returned and **errno** is set to indicate the error.

**tcsetpgrp** returns a value of zero upon success. Otherwise, -1 is returned and **errno** is set to indicate the error.



## ***Errors***

---

If any of the following conditions occur, **tcgetpgrp** returns -1 and sets **errno** to the corresponding value:

- [EBADF]**     The *fil-des* argument is not a valid file descriptor.
- [ENOSYS]**     The **tcgetpgrp** function is not supported.
- [ENOTTY]**     The calling process does not have a controlling terminal or the file described in *fil-des* is not the controlling terminal.

If any of the following conditions occur, **tcsetpgrp** returns -1 and sets **errno** to the corresponding value:

- [EBADF]**     The *fil-des* argument is not a valid file descriptor.
- [EINVAL]**     The value of *pgrp-id* is not supported.
- [ENOSYS]**     The **tcsetpgrp** function is not supported.
- [ENOTTY]**     The calling process does not have a controlling terminal or the file described in *fil-des* is not the controlling terminal.
- [EPERM]**     The value of *pgrp-id* is a value supported by the implementation but does not match the process group ID of a process in the same session as the calling process.

## ***See also***

---

**tcflow(S), tcattr(S)**

## ***Standards conformance***

---

**tcgetpgrp** and **tcsetpgrp** are conformant with:

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1) and X/Open Portability Guide, Issue 3, 1989.

# termcap: tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs

performs terminal functions

## Syntax

```
extern char PC;
extern char *BC;
extern char *UP;
extern short ospeed;

int tgetent(bp, name)
char *bp, *name;

int tgetnum(id)
char *id;

int tgetflag(id)
char *id;

char * tgetstr(id, area)
char *id, **area;

char * tgoto(cm, destcol, destline)
char *cm;
int destcol, destline;

void tputs(cp, affcnt, outc)
char *cp;
int affcnt;
int (*outc)();
```

## Description

These functions extract and use capabilities from the terminal capability data base *termcap*(F). These are low level routines; see *curses*(S) for a higher level package.

**tgetent** extracts the entry for terminal *name* into the buffer at *bp*; *bp* should be a character buffer of size 1024 and must be retained through all subsequent calls to **tgetnum**, **tgetflag**, and **tgetstr**. **tgetent** returns -1 if it cannot open the *termcap* file, 0 if the terminal name given does not have an entry, and 1 if all goes well. It looks in the environment for a TERMCAP variable. If found, and the value does not begin with a slash, and the terminal type *name* is the same as the environment string TERM, the TERMCAP string is used instead of reading the *termcap* file. If it does begin with a slash, the string is used as a pathname rather than */etc/termcap*. This can speed up entry into programs that call **tgetent**, as well as to help debug new terminal descriptions or to make one for your terminal if you can't write the file */etc/termcap*.

**tgetnum** gets the numeric value of capability *id*, returning -1 if it is not given for the terminal. **tgetflag** returns 1 if the specified capability is present in the terminal's entry, 0 if it is not. **tgetstr** returns the string value of capability *id*, advancing the *area* pointer. It decodes the abbreviations for this field described in *termcap*(F), except for cursor addressing and padding information.

**tgoto** returns a cursor addressing string decoded from *cm* to go to column *destcol* in line *destline*. It uses the external variables UP (from the up capability) and BC (if bc is given rather than bs) if necessary to avoid placing \n, (Ctrl-D) or NULL in the returned string. Programs which call **tgoto** should be sure to turn off the TAB3 bit (see **tty**(M)), since **tgoto** may now output a tab. Note that programs using *termcap* should turn off TAB3 anyway since some terminals use (Ctrl-I) for other functions, such as nondestructive space.) If a % sequence is given which is not understood, then **tgoto** returns OOPS.

**tputs** decodes the leading padding information of the string *cp*; *affcnt* gives the number of lines affected by the operation, or 1 if this is not applicable, *outc* is a routine which is called with each character in turn. The external variable *ospeed* should contain the output speed of the terminal as encoded by **stty**(C). The external variable PC should contain a pad character to be used (from the pc capability) if a NULL is inappropriate.

## Files

---

/usr/lib/libtermcap.a -ltermcap library  
/etc/termcap data base

## See also

---

**curses**(S), **termcap**(F), **tty**(M)

## Notes

---

These routines must be linked by using the **-ltermcap** linker option.

## Standards conformance

---

**termcap** is not part of any currently supported standard; it was developed at the University of California at Berkeley and is used by permission.

# terminfo

terminal description database

## Syntax

```
cc -DM_TERMINFO ... -ltnfo [-lx]
```

```
#include <curses.h>
#include <term.h>
```

## Description

These routines give the user a method of updating screens with reasonable optimization. In order to initialize the routines, the routine `initscr` must be called before any of the other routines that deal with windows and screens are used. The routine `endwin` should be called before exiting. To get character-at-a-time input without echoing, (most interactive, screen oriented-programs want this) after calling `initscr` you should call `"nonl(); cbreak(); noecho();"`

The `curses` interface permits manipulation of data structures called **windows** which can be thought of as two dimensional arrays of characters representing all or part of a CRT screen. A default window called `stdscr` is supplied, and others can be created with `newwin`. Windows are referred to by variables declared `"WINDOW *"`, the type `WINDOW` is defined in `curses.h` to be a C structure. These data structures are manipulated with functions described below, among which the most basic are `move`, and `addch`. (More general versions of these functions are included with names beginning with 'w', allowing you to specify a window. The routines not beginning with 'w' affect `stdscr`.) Then `refresh()` is called, telling the routines to make the users CRT screen look like `stdscr`.

If the environment variable `TERMINFO` is defined, any program using `curses` checks for a local terminal definition before checking in the standard place. For example, if the standard place is `/usr/lib/terminfo`, and `TERM` is set to `"vt100"`, then normally the compiled file is found in `/usr/lib/terminfo/v/vt100`. (The "v" is copied from the first letter of "vt100" to avoid creation of huge directories.) However, if `TERMINFO` is set to `/usr/mark/myterms`, `curses` first checks `/usr/mark/myterms/v/vt100`, and if that fails, then checks `/usr/lib/terminfo/v/vt100`. This is useful for developing experimental definitions or when write permission in `/usr/lib/terminfo` is not available.

## Functions

---

Routines listed here may be called when using curses.

|                            |                                                                                                                                        |
|----------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| <b>addch(ch)</b>           | add a character to <b>stdscr</b> (like putchar)<br>(wraps to next line at end of line)                                                 |
| <b>addstr(str)</b>         | calls addch with each character in <b>str</b>                                                                                          |
| <b>attroff(attrs)</b>      | turn off attributes named                                                                                                              |
| <b>attron(attrs)</b>       | turn on attributes named                                                                                                               |
| <b>attrset(attrs)</b>      | set current attributes to <b>attrs</b>                                                                                                 |
| <b>baudrate()</b>          | current terminal speed                                                                                                                 |
| <b>beep()</b>              | sound beep on terminal                                                                                                                 |
| <b>box(win, vert, hor)</b> | draw a box around edges of <b>win</b><br><b>vert</b> and <b>hor</b> are chars to use for <b>vert.</b><br>and horizontal edges of a box |
| <b>clear()</b>             | clear <b>stdscr</b>                                                                                                                    |
| <b>clearok(win, bf)</b>    | clear screen before next redraw of <b>win</b>                                                                                          |
| <b>clrtobot()</b>          | clear to bottom of <b>stdscr</b>                                                                                                       |
| <b>clrtoeol()</b>          | clear to end of line on <b>stdscr</b>                                                                                                  |
| <b>cbreak()</b>            | set cbreak mode                                                                                                                        |
| <b>delay_output(ms)</b>    | insert <b>ms</b> millisecond pause in output                                                                                           |
| <b>delch()</b>             | delete a character                                                                                                                     |
| <b>deleteln()</b>          | delete a line                                                                                                                          |
| <b>delwin(win)</b>         | delete <b>win</b>                                                                                                                      |
| <b>doupdate()</b>          | update screen from all <b>wnoutrefresh</b>                                                                                             |
| <b>echo()</b>              | set echo mode                                                                                                                          |
| <b>endwin()</b>            | end window modes                                                                                                                       |
| <b>erase()</b>             | erase <b>stdscr</b>                                                                                                                    |
| <b>erasechar()</b>         | return user's erase character                                                                                                          |
| <b>fixterm()</b>           | restore tty to "in curses" state                                                                                                       |
| <b>flash()</b>             | flash screen or beep                                                                                                                   |
| <b>flushinp()</b>          | throw away any typeahead                                                                                                               |
| <b>getch()</b>             | get a char from tty                                                                                                                    |
| <b>getstr(str)</b>         | get a string through <b>stdscr</b>                                                                                                     |
| <b>gettmode()</b>          | establish current tty modes                                                                                                            |
| <b>getyx(win, y, x)</b>    | get (y, x) co-ordinates                                                                                                                |
| <b>has_ic()</b>            | true if terminal can do insert character                                                                                               |
| <b>has_il()</b>            | true if terminal can do insert line                                                                                                    |
| <b>idlok(win, bf)</b>      | use terminal's insert/delete line if <b>bf</b> != 0                                                                                    |
| <b>inch()</b>              | get char at current (y, x) co-ordinates                                                                                                |
| <b>initscr()</b>           | initialize screens                                                                                                                     |
| <b>insch(c)</b>            | insert a char                                                                                                                          |
| <b>insertln()</b>          | insert a line                                                                                                                          |
| <b>intrflush(win, bf)</b>  | interrupts flush output if <b>bf</b> is TRUE                                                                                           |
| <b>keypad(win, bf)</b>     | enable keypad input                                                                                                                    |
| <b>killchar()</b>          | return current user's kill character                                                                                                   |
| <b>leaveok(win, flag)</b>  | OK to leave cursor anywhere after<br>refresh if <b>flag</b> !=0 for <b>win</b> , otherwise cursor<br>must be left at current position. |
| <b>longname()</b>          | return verbose name of terminal                                                                                                        |

|                                                                                |                                                                                                      |
|--------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|
| <b>meta(win, flag)</b>                                                         | allow meta characters on input if flag != 0                                                          |
| <b>move(y, x)</b>                                                              | move to (y, x) on <b>stdscr</b>                                                                      |
| <b>mvaddch(y, x, ch)</b>                                                       | move(y, x) then addch(ch)                                                                            |
| <b>mvaddstr(y, x, str)</b>                                                     | similar...                                                                                           |
| <b>mvcur(oldrow, oldcol, newrow, newcol)</b>                                   | low level cursor motion                                                                              |
| <b>mvdelch(y, x)</b>                                                           | like delch, but move(y, x) first                                                                     |
| <b>mvgetch(y, x)</b>                                                           | etc.                                                                                                 |
| <b>mvgetstr(y, x)</b>                                                          |                                                                                                      |
| <b>mvinch(y, x)</b>                                                            |                                                                                                      |
| <b>mvinsch(y, x, c)</b>                                                        |                                                                                                      |
| <b>mvprintw(y, x, fmt, args)</b>                                               |                                                                                                      |
| <b>mvscanw(y, x, fmt, args)</b>                                                |                                                                                                      |
| <b>mvwaddch(win, y, x, ch)</b>                                                 |                                                                                                      |
| <b>mvwaddstr(win, y, x, str)</b>                                               |                                                                                                      |
| <b>mvwdelch(win, y, x)</b>                                                     |                                                                                                      |
| <b>mvwgetch(win, y, x)</b>                                                     |                                                                                                      |
| <b>mvwgetstr(win, y, x, str)</b>                                               |                                                                                                      |
| <b>mvwin(win, by, bx)</b>                                                      |                                                                                                      |
| <b>mvwinch(win, y, x)</b>                                                      |                                                                                                      |
| <b>mvwinsch(win, y, x, c)</b>                                                  |                                                                                                      |
| <b>mvwprintw(win, y, x, fmt, args)</b>                                         |                                                                                                      |
| <b>mvwscanw(win, y, x, fmt, args)</b>                                          |                                                                                                      |
| <b>newpad(nlines, ncols)</b>                                                   | create a new pad with given dimensions                                                               |
| <b>newterm(type, fd)</b>                                                       | set up new terminal of given type to output on fd                                                    |
| <b>newwin(lines, cols, begin_y, begin_x)</b>                                   | create a new window                                                                                  |
| <b>nl()</b>                                                                    | set newline mapping                                                                                  |
| <b>nocbreak()</b>                                                              | unset cbreak mode                                                                                    |
| <b>nodelay(win, bf)</b>                                                        | enable nodelay input mode through getch                                                              |
| <b>noecho()</b>                                                                | unset echo mode                                                                                      |
| <b>nonl()</b>                                                                  | unset newline mapping                                                                                |
| <b>noraw()</b>                                                                 | unset raw mode                                                                                       |
| <b>overlay(win1, win2)</b>                                                     | overlay win1 on win2                                                                                 |
| <b>overwrite(win1, win2)</b>                                                   | overwrite win1 on top of win2                                                                        |
| <b>pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)</b> | like prefresh but with no output until doupdate called                                               |
| <b>preresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)</b>      | refresh from pad starting with given upper left corner of pad with output to given portion of screen |
| <b>printw(fmt, arg1, arg2, ...)</b>                                            | printf on <b>stdscr</b>                                                                              |
| <b>raw()</b>                                                                   | set raw mode                                                                                         |
| <b>refresh()</b>                                                               | make current screen look like <b>stdscr</b>                                                          |
| <b>resetterm()</b>                                                             | set tty modes to "out of curses" state                                                               |

|                                                   |                                                |
|---------------------------------------------------|------------------------------------------------|
| <b>resetty()</b>                                  | reset tty flags to stored value                |
| <b>saveterm()</b>                                 | save current modes as "in curses" state        |
| <b>savetty()</b>                                  | store current tty flags                        |
| <b>scanw(fmt, arg1, arg2, ...)</b>                | scanf through stdscr                           |
| <b>scroll(win)</b>                                | scroll <b>win</b> one line                     |
| <b>scrollok(win, flag)</b>                        | allow terminal to scroll if flag !=0           |
| <b>set_term(new)</b>                              | now talk to terminal new                       |
| <b>setscrreg(t, b)</b>                            | set user scrolling region to lines t through b |
| <b>setterm(type)</b>                              | establish terminal with given type             |
| <b>setupterm(term, filenum, errret)</b>           | reads in <b>terminfo(F)</b> database           |
| <b>standend()</b>                                 | clear standout mode attribute                  |
| <b>standout()</b>                                 | set standout mode attribute                    |
| <b>subwin(win, lines, cols, begin_y, begin_x)</b> | create a subwindow                             |
| <b>touchwin(win)</b>                              | "change" all of <b>win</b>                     |
| <b>traceoff()</b>                                 | turn off debugging trace output                |
| <b>traceon()</b>                                  | turn on debugging trace output                 |
| <b>typeahead(fd)</b>                              | use file descriptor fd to check typeahead      |
| <b>unctrl(ch)</b>                                 | printable version of <b>ch</b>                 |
| <b>waddch(win, ch)</b>                            | add char to <b>win</b>                         |
| <b>waddstr(win, str)</b>                          | add string to <b>win</b>                       |
| <b>wattroff(win, attrs)</b>                       | turn off <b>attrs</b> in <b>win</b>            |
| <b>wattron(win, attrs)</b>                        | turn on <b>attrs</b> in <b>win</b>             |
| <b>wattrset(win, attrs)</b>                       | set <b>attrs</b> in <b>win</b> to <b>attrs</b> |
| <b>wclear(win)</b>                                | clear <b>win</b>                               |
| <b>wclrtoBot(win)</b>                             | clear to bottom of <b>win</b>                  |
| <b>wclrtoeol(win)</b>                             | clear to end of line on <b>win</b>             |
| <b>wdelch(win, c)</b>                             | delete char from <b>win</b>                    |
| <b>wdeleteln(win)</b>                             | delete line from <b>win</b>                    |
| <b>werase(win)</b>                                | erase <b>win</b>                               |
| <b>wgetch(win)</b>                                | get a char through <b>win</b>                  |
| <b>wgetstr(win, str)</b>                          | get a string through <b>win</b>                |
| <b>winch(win)</b>                                 | get char at current (y, x) in <b>win</b>       |
| <b>winsch(win, c)</b>                             | insert char into <b>win</b>                    |
| <b>winsertln(win)</b>                             | insert line into <b>win</b>                    |
| <b>wmove(win, y, x)</b>                           | set current (y, x) coordinates on <b>win</b>   |
| <b>wnoutrefresh(win)</b>                          | refresh but no screen output                   |
| <b>wprintw(win, fmt, arg1, arg2, ...)</b>         | printf on <b>win</b>                           |
| <b>wrefresh(win)</b>                              | make screen look like <b>win</b>               |
| <b>wscanw(win, fmt, arg1, arg2, ...)</b>          | scanf through <b>win</b>                       |
| <b>wsetscrreg(win, t, b)</b>                      | set scrolling region of <b>win</b>             |
| <b>wstandend(win)</b>                             | clear standout attribute in <b>win</b>         |
| <b>wstandout(win)</b>                             | set standout attribute in <b>win</b>           |

## Terminfo level routines

These routines should be called by programs wishing to deal directly with the terminfo database. Due to the low level of this interface, it is discouraged. Initially, **setupterm** should be called. This defines the set of terminal dependent variables defined in *terminfo(F)*. The include files *curses.h* and *term.h* should be included to get the definitions for these strings, numbers, and flags. Parameterized strings should be passed through **tparm** to instantiate them. All terminfo strings (including the output of **tparm**) should be printed with **tputs** or **putp**. Before exiting, **resetterm** should be called to restore the tty modes. (Programs desiring shell escapes can call **resetterm** before the shell is called and **fixterm** after returning from the shell.)

|                                        |                                                                                                                                                                                                                                                                                                                       |
|----------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>fixterm()</b>                       | restore tty modes for terminfo use (called by <b>setupterm</b> )                                                                                                                                                                                                                                                      |
| <b>resetterm()</b>                     | reset tty modes to state before program entry                                                                                                                                                                                                                                                                         |
| <b>setupterm(term,<br/>fd, rc)</b>     | read in database. Terminal type is the character string <b>term</b> , all output is to the system file descriptor <b>fd</b> . A status value is returned in the integer pointed to by <b>rc</b> : 1 is normal. The simplest call would be <b>setupterm(0, 1, 0)</b> which uses all defaults.                          |
| <b>tparm(str, p1, p2,<br/>..., p9)</b> | instantiate string <b>str</b> with parms <b>p<sub>i</sub></b> .                                                                                                                                                                                                                                                       |
| <b>tputs(str, affcnt,<br/>putc)</b>    | apply padding info to string <b>str</b> . <b>affcnt</b> is the number of lines affected, or 1 if not applicable. <b>putc</b> is a putchar-like function to which the characters are passed, one at a time. Note that the user must supply their own <b>putc</b> function.                                             |
| <b>putp(str)</b>                       | handy function that calls <b>tputs</b>                                                                                                                                                                                                                                                                                |
| <b>vidputs(attrs, putc)</b>            | <b>(str, 1, putchar)</b><br>output the string to put terminal in video attribute mode <b>attrs</b> , which is any combination of the attributes listed below. Chars are passed to putchar-like function <b>putc</b> . Standard <b>putchar</b> can be used here or the user can supply their own <b>putc</b> function. |
| <b>vidattr(attrs)</b>                  | Like <b>vidputs</b> but outputs through <b>putchar</b>                                                                                                                                                                                                                                                                |



## Termcap compatibility routines

---

These routines were included as a conversion aid for programs that use **termcap**(S). Their parameters are the same as used in **termcap**. They are emulated using the **terminfo**(F) database. They may be removed at a later date.

|                                |                                            |
|--------------------------------|--------------------------------------------|
| <b>tgetent</b> (bp, name)      | look up termcap entry for name             |
| <b>tgetflag</b> (id)           | get boolean entry for id                   |
| <b>tgetnum</b> (id)            | get numeric entry for id                   |
| <b>tgetstr</b> (id, area)      | get string entry for id                    |
| <b>tgoto</b> (cap, col, row)   | apply parms to given cap                   |
| <b>tputs</b> (cap, affcnt, fn) | apply padding to cap calling fn as putchar |

## Attributes

---

The following video attributes can be passed to the functions **attron**, **attroff**, **attrset**.

|                     |                                   |
|---------------------|-----------------------------------|
| <b>A_STANDOUT</b>   | Terminal's best highlighting mode |
| <b>A_UNDERLINE</b>  | Underlining                       |
| <b>A_REVERSE</b>    | Reverse video                     |
| <b>A_BLINK</b>      | Blinking                          |
| <b>A_DIM</b>        | Half bright                       |
| <b>A_BOLD</b>       | Extra bright or bold              |
| <b>A_BLANK</b>      | Blanking (invisible)              |
| <b>A_PROTECT</b>    | Protected                         |
| <b>A_ALTCHARSET</b> | Alternate character set           |

## Function keys

---

The following function keys might be returned by **getch** if **keypad** has been enabled. Note that not all of these are currently supported, due to lack of definitions in **terminfo** or the terminal not transmitting a unique code when the key is pressed.

| Name                 | Value        | Key name                                 |
|----------------------|--------------|------------------------------------------|
| <b>KEY_BREAK</b>     | 0401         | break key (unreliable)                   |
| <b>KEY_DOWN</b>      | 0402         | The four arrow keys ...                  |
| <b>KEY_UP</b>        | 0403         |                                          |
| <b>KEY_LEFT</b>      | 0404         |                                          |
| <b>KEY_RIGHT</b>     | 0405         | ...                                      |
| <b>KEY_HOME</b>      | 0406         | Home key (upward+left arrow)             |
| <b>KEY_BACKSPACE</b> | 0407         | backspace (unreliable)                   |
| <b>KEY_F0</b>        | 0410         | Function keys. Space for 64 is reserved. |
| <b>KEY_F(n)</b>      | (KEY_F0+(n)) | Formula for fn.                          |
| <b>KEY_DL</b>        | 0510         | Delete line                              |
| <b>KEY_IL</b>        | 0511         | Insert line                              |
| <b>KEY_DC</b>        | 0512         | Delete character                         |
| <b>KEY_IC</b>        | 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 (unreliable)        |
| KEY_SRESET | 0530 | soft (partial) reset (unreliable) |
| KEY_RESET  | 0531 | reset or hard reset (unreliable)  |
| KEY_PRINT  | 0532 | print or copy                     |
| KEY_LL     | 0533 | home down or bottom (lower left)  |

## See also

---

**terminfo(F)**

## Standards conformance

---

**terminfo** is not part of any currently supported standard; it is an extension of AT&T System V provided by the Santa Cruz Operation.

# time, ftime

---

return time

## Syntax

---

```
cc ... -lc

#include <sys/types.h>
#include <time.h>

time_t time (tloc)
time_t *tloc;

#include <sys/types.h>
#include <sys/timeb.h>
#include <time.h>

void ftime (tp)
struct timeb *tp;
```

## Description

---

The **time** system call returns the value of time in seconds since 00:00:00 Greenwich Mean Time (GMT), January 1, 1970.

If *tloc* is nonzero, the return value is also stored in the location to which *tloc* points.

**ftime** returns the time in the **timeb** structure as follows:

```
struct timeb {
 time_t time; /* time, seconds since the epoch */
 unsigned short millitm; /* 1000 msec of additional accuracy */
 short timezone; /* timezone, minutes west of GMT */
 short dstflag; /* daylight savings */
};
```

The **timezone** member is a system default timezone and not the value of the **TZ** environment value. This structure contains the time since the epoch in seconds, up to 1000 milliseconds of a more-precise interval, the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Savings Time applies locally during the appropriate part of the year.

## Diagnostics

---

Upon successful completion, **time** returns the value of time. Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

The **time** system call fails and its actions are undefined if *tloc* points to an illegal address. **ftime** fails if *tp* points to an illegal address. For either failure, **errno** is set to **EFAULT**.

## See Also

---

**ctime(S)**, **difftime(S)**, **mktime(S)**, **nl\_cxtime(S)**, **stime(S)**, **times(S)**

## Standards conformance

---

**time** is conformant with:

AT&T SVID Issue 2;  
 X/Open Portability Guide, Issue 3, 1989;  
 ANSI X3.159-1989 Programming Language — C;  
 Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);  
 IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);  
 and NIST FIPS 151-1.

**ftime** is not part of any currently supported standard; it is an extension of AT&T System V provided by The Santa Cruz Operation, Inc.

# times

get process and child process times

## Syntax

---

```
cc ... -lc

#include <sys/types.h>
#include <sys/times.h>

clock_t times (buffer)
struct tms *buffer;
```

## Description

---

The **times** system call fills the structure pointed to by *buffer* with time-accounting information. The following are the contents of this structure:

```
struct tms {
 clock_t tms_utime;
 clock_t tms_stime;
 clock_t tms_cutime;
 clock_t tms_cstime;
};
```

This information comes from the calling process and each of its terminated child processes for which it has executed a **wait**. All times are reported in clock ticks. The number of clock ticks per second is a system-dependent parameter, which is returned by **sysconf(\_SC\_CLK\_TCK)**.

**tms\_utime** is the CPU time used while executing instructions in the user space of the calling process.

**tms\_stime** is the CPU time used by the system on behalf of the calling process.

**tms\_cutime** is the sum of the **tms\_utimes** and **tms\_cutimes** of the child processes.

**tms\_cstime** is the sum of the **tms\_stimes** and **tms\_cstimes** of the child processes.

## Diagnostics

---

[EFAULT] The **times** system call fails if *buffer* points to an illegal address.

## *Return value*

---

Upon successful completion, **times** returns the elapsed real time, in clock ticks, from an arbitrary point in the past (for example, system start-up time). This point does not change from one invocation of **times** to another. If **times** fails, a -1 is returned and **errno** is set to indicate the error. The return value may overflow the possible range of type **clock\_t**.

## *See also*

---

**exec(S)**, **fork(S)**, **gethz(S)**, **sysconf(S)**, **time(S)**, **wait(S)**

## *Standards conformance*

---

**times** is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

# tmpfile

---

create a temporary file

## Syntax

---

```
cc ... -lc

#include <stdio.h>

FILE *tmpfile ()
```

## Description

---

The **tmpfile** function creates a temporary file and returns the corresponding **FILE** pointer. If the file cannot be opened, a **NULL** pointer is returned. The file is automatically deleted when all references to the file have been closed. The file is opened for update (**w+**).

## Diagnostics

---

The **tmpfile** routine fails if:

- |          |                                                                                   |
|----------|-----------------------------------------------------------------------------------|
| [EMFILE] | [FOPEN_MAX] streams are currently open in the calling process.                    |
| [ENFILE] | The system file table is full.                                                    |
| [ENOSPC] | The directory of file system which would contain the new file cannot be expanded. |

The **tmpfile** routine may fail if:

- |           |                                                                                                                                                                            |
|-----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [EACCES]  | Search permission is denied on a component of the path prefix of the file to be created, or write permission is denied for the parent directory of the file to be created. |
| [EINTR]   | A signal was caught during the <b>tmpfile</b> function.                                                                                                                    |
| [ENOMEM]  | Insufficient storage space is available.                                                                                                                                   |
| [ENOTDIR] | A component of the path prefix of the file to be created is not a directory.                                                                                               |
| [EROFS]   | The file to be created would reside on a read-only file system.                                                                                                            |

## Note

---

The stream refers to a file that has been unlinked. If the process is killed in the period between file creation and unlinking, a permanent file may be left behind.

## See also

---

creat(S), fopen(S), mktemp(S), stdio(S), tmpnam(S) unlink(S)

## Standards conformance

---

tmpfile is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

ANSI X3.159-1989 Programming Language — C;

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.



# tmpnam, tempnam

---

create a name for a temporary file

## Syntax

---

```
cc ... -lc

#include <stdio.h>

char *tmpnam (s)
char *s;

char *tempnam (dir, pfx)
char *dir, *pfx;
```

## Description

---

These functions generate filenames that are not the same as name of an existing file.

The **tmpnam** function always generates a filename using the path-prefix defined as *P\_tmpdir* in the *<stdio.h>* header file. The generated filename is different each time that **tmpnam** is called from the same process, up to **TMP\_MAX** times. If *s* is **NULL**, **tmpnam** leaves its result in an internal static area and returns a pointer to that area. The next call to **tmpnam** destroys the contents of the area. If *s* is not **NULL**, it is assumed to be the address of an array of at least **L\_tmpnam** bytes, where **L\_tmpnam** is a constant defined in *<stdio.h>*; **tmpnam** places its result in that array and returns *s*.

**tempnam** allows choosing a directory. The argument *dir* points to the name of the directory in which the file is to be created. If *dir* is **NULL** or points to a string that is not a name for an appropriate directory, the path-prefix defined as *P\_tmpdir* in the *<stdio.h>* header file is used. If that directory is not accessible, */tmp* is used as a last resort. This entire sequence can be up-staged by providing an environment variable **TMPDIR** in the user's environment, whose value is the name of the desired temporary-file directory.

Many applications prefer their temporary files to have certain favorite initial letter sequences in their names. Use the *pfx* argument for this. This argument may be **NULL** or point to a string of up to five characters to be used as the first few characters of the temporary-filename.

## Return value

---

**tempnam** uses **malloc(S)** to get space for the constructed filename and returns a pointer to this area. Thus, any pointer value returned from **tempnam** may serve as an argument to **free** (see **malloc(S)**).

If **tmpnam** cannot return the expected result for any reason, i.e., **malloc(S)** failed, or none of the above mentioned attempts to find an appropriate directory was successful, a **NULL** pointer is returned.

## Diagnostics

---

The **tmpnam** function will fail if:

[ENOMEM]      Insufficient storage space is available.

## Notes

---

These functions generate a different filename each time they are called.

Files created using these functions and either **fopen(S)** or **creat(S)** are temporary only in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to use **unlink(S)** to remove the file when its use is ended.

If called more than 17,576 times in a single process, these functions recycle previously used names.

Between the time a filename is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using these functions or **mktemp**, and the filenames are chosen to render duplication by other means unlikely.

## See also

---

**creat(S)**, **fopen(S)**, **malloc(S)**, **mktemp(S)**, **tmpfile(S)**, **unlink(S)**

## Standards conformance

---

**tmpnam** is conformant with:

AT&T SVID Issue 2;  
 X/Open Portability Guide, Issue 3, 1989;  
 ANSI X3.159-1989 Programming Language — C;  
 IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);  
 and NIST FIPS 151-1.

# **toascii, todigit, toint, tolower, \_tolower, toupper, \_toupper**

---

routines used to translate characters

## ***Syntax***

---

```
cc ... -lc

#include <ctype.h>

int toascii(c)
int c;

int todigit(i)
int i;

int toint(c)
int c;

int tolower(c)
int c;

int _tolower(c)
int c;

int toupper(c)
int c;

int _toupper(c)
int c;
```

## ***Description***

---

The functions **toupper** and **tolower** convert the argument *c* to a letter of the opposite case. Arguments may be the integers  $-1$  to  $255$  (the same values returned by **get(S)**). If the argument of **toupper** represents a lowercase letter, the result is the corresponding uppercase letter. If the argument of **tolower** represents an uppercase letter, the result is the corresponding lowercase letter. All other arguments are returned unchanged.

The effect of the macros `_toupper` and `_tolower` is the same as `toupper` and `tolower`. The advantage of `_toupper` and `_tolower` is that they have restricted argument values and are faster. The `_toupper` macro takes a lowercase letter as its argument. The result is the corresponding uppercase letter. The `_tolower` macro takes an uppercase letter as its argument. The result is the corresponding lowercase letter. All other arguments cause unpredictable results.

The macro `toascii` yields its argument with all bits cleared that are not part of a standard ASCII character; it is intended for compatibility with other systems.

The macro `todigit` returns the digit character corresponding to its integer argument. The argument must be in the range 0-9, otherwise the behavior is undefined.

The macro `toint` returns the integer corresponding to the digit character supplied as its argument. The argument must be one of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, otherwise the behavior is undefined.

## Notes

---

The routines `_toupper`, `_tolower` and `toascii` are implemented as macros, and on other systems, `toupper` and `tolower` may be implemented as macros. On this system, all `conv(S)` routines evaluate their arguments once and once only. It is, however, good practice to ensure that macros are not called with arguments that have side effects which differ if they are evaluated more than once. An example of such a macro is:

```
toupper (*p++)
```

Avoiding such argument calls helps to keep code portable.

The functions `toupper` and `tolower` are library functions. Their arguments are converted to integers. For this reason, care should be taken that character arguments are supplied as unsigned characters to avoid problems with sign-extension of 8-bit character values.

The `todigit` and `toint` macros are extensions, and may not be present on other systems.

## See also

---

`ctype(S)`

## ***Standards conformance***

---

**toascii** is conformant with: AT&T SVID Issue 2;  
and X/Open Portability Guide, Issue 3, 1989. **tolower** and **toupper** are conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

ANSI X3.159-1989 Programming Language — C;

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

# trig: acos, asin, atan, atan2, cos, sin, tan

## trigonometric functions

### Syntax

```
cc ... -lm

#include <math.h>

double acos (x)
double x;

double asin (x)
double x;

double atan (x)
double x;

double atan2 (y, x)
double y, x;

double cos (x)
double x;

double sin (x)
double x;

double tan (x)
double x;
```

### Description

The **acos** routine computes the arc cosine of  $x$  in the range of  $[0, \pi]$  radians. Specify  $x$  in the range of -1 to 1. Zero is returned on unsuccessful completion and **errno** indicates the error, or NaN (not a number) is returned and **errno** may be set to indicate the error. (NaN is defined in `/usr/include/math.h` and is detected with the **isnan(S)** routine.)

The **asin** routine returns the arc sine of  $x$  in the range of  $[-\pi/2, \pi/2]$  radians. Specify  $x$  in range of -1 to 1. Zero is returned on unsuccessful completion and **errno** indicates the error, or NaN is returned and **errno** may indicate the error.

The **atan** routine returns the arc tangent of  $x$  in the range  $[-\pi/2, \pi/2]$  radians. Otherwise **errno** indicates the error or NaN is returned.

The **atan2** routine returns the arc tangent of  $y/x$  in the range of  $(-\pi, \pi]$ . Zero is returned on unsuccessful completion with **errno** indicating the error, or NaN is returned and **errno** may indicate the error.

The **cos** routine returns the cosine of  $x$  measured in radians on successful completion. Otherwise, **errno** is set to indicate the error or NaN is returned. The **cos** routine may lose accuracy when its argument is far from zero.

The **sin** routine returns the sine of its argument  $x$  measured in radians upon successful completion. Otherwise, **errno** is set to indicate the error or NaN is returned. The **sin** routine may lose accuracy when its argument is far from zero. When the **sin** routine fails, **errno** is set to one of:

[EDOM]      The value of  $x$  is NaN, or  $x$  is  $\pm$ HUGE\_VAL.

[ERANGE]    The magnitude of  $x$  is such that total or partial loss of significance resulted.

The **tan** routine returns the tangent of  $x$  measured in radians on successful completion. Otherwise, **errno** indicates the error or NaN is returned. The **tan** routine may lose accuracy when its argument is far from zero.

The **tan** routine may fail if:

[EDOM]      The value of  $x$  is NaN.

[ERANGE]    The value to be returned would have caused overflow or underflow, or the magnitude of  $x$  is such that total or partial loss of signal would result.

## *Note*

---

Before checking for errors, set **errno** to zero. If a value appears in **errno** or if the routine returns NaN, an error occurred.

## *Diagnostics*

---

**sin**, **cos**, and **tan** lose accuracy when their argument is far from zero. For arguments sufficiently large, these functions return zero when there would otherwise be a complete loss of significance. In this case a message indicating TLOSS error is printed on the standard error output. For less extreme arguments causing partial loss of significance, a PLOSS error is generated but no message is printed. In both cases, **errno** is set to ERANGE.

If the magnitude of the argument of **asin** or **acos** is greater than one, or if both arguments of **atan2** are zero, zero is returned and **errno** is set to EDOM. In addition, a message indicating DOMAIN error is printed on the standard error output.

These error-handling procedures may be changed with the function **matherr(S)**.

## See also

---

**matherr(S)**

## Standards conformance

---

**acos, asin, atan, atan2, cos, sin, and tan**

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

ANSI X3.159-1989 Programming Language — C;

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

(S)



# tsearch, tfind, tdelete, twalk

manage binary search trees

## Syntax

```
cc ... -lc

#include <search.h>

void *tsearch (key, rootp, compar)
void *key, **rootp;
int (*compar)();

void *tfind (key, rootp, compar)
void *key, **rootp;
int (*compar)();

void *tdelete (key, rootp, compar)
void *key, **rootp;
int (*compar)();

void twalk (root, action)
void *root;
void (*action)();
```

## Description

**tsearch** - builds and accesses search tree

**tfind** - searches for a datum in the tree and returns a pointer

**tdelete** - deletes a node from a binary search tree

**twalk** - traverses a binary search tree

The **tsearch**, **tfind**, **tdelete**, and **twalk** routines manipulate binary search trees. They are generalized from Knuth (6.2.2) Algorithms T and D. All comparisons are done with a user-supplied routine. This routine is called with two arguments, the pointers to the elements being compared. The user-supplied function must return an integer less than, equal to, or greater than 0, according to whether the first argument is to be considered less than, equal to, or greater than the second argument. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

The **tsearch** function is used to build and access the tree. *key* is a pointer to a datum to be accessed or stored. If there is a datum in the tree equal to *\*key* (the value pointed to by *key*), a pointer to this found datum is returned. Otherwise, *\*key* is inserted, and a pointer to it returned. Only pointers are copied, so the calling routine must store the data. *rootp* points to a variable that

points to the root of the tree. A null value for the variable pointed to by *rootp* denotes an empty tree; in this case, the variable is set to point to the datum that is at the root of the new tree.

Like *tsearch*, *tfind* searches for a datum in the tree, returning a pointer to it if found. However, if it is not found, *tfind* returns a null pointer. The arguments for *tfind* are the same as for *tsearch*.

*tdelete* deletes a node from a binary search tree. The arguments are the same as for *tsearch*. The variable pointed to by *rootp* is changed if the deleted node was the root of the tree. *tdelete* returns a pointer to the parent of the deleted node, or a NULL pointer if the node is not found.

*twalk* traverses a binary search tree. *root* is the root of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) *action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The second argument is a value from an enumeration data type `typedef enum { preorder, postorder, endorder, leaf } VISIT;` (defined in the `<search.h>` header file), depending on whether this is the first, second, or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level zero.

The pointers to the key and the root of the tree should be of type pointer-to-element, and cast to type pointer-to-void. Similarly, although declared as type pointer-to-void, the value returned should be cast into type pointer-to-element.

## Example

---

The following code reads in strings and stores structures containing a pointer to each string and a count of its length. It then walks the tree, printing out the stored strings and their lengths in alphabetical order.

```
#include <search.h>
#include <stdio.h>

#define STRSZ 10000
#define NODSZ 500

struct node { /* pointers to these are stored in the tree */
 char *string;
 int length;
};
char string_space[STRSZ]; /* space to store strings */
struct node nodes[NODSZ]; /* nodes to store */
struct node *root = NULL; /* this points to the root */

main()
{
 char *strptr = string_space;
 struct node *nodeptr = nodes;
 void print_node(), twalk();
 int i = 0, node_compare();

 while (gets(strptr) != NULL && i++ < NODSZ) {
 /* set node */
 nodeptr->string = strptr;
 nodeptr->length = strlen(strptr);
 /* put node into the tree */
 (void) tsearch((void *)nodeptr, (void **) &root,
 node_compare);
 /* adjust pointers, so we don't overwrite tree */
 strptr += nodeptr->length + 1;
 nodeptr++;
 }
 twalk((void *)root, print_node);
}
/*
 This routine compares two nodes, based on an
 alphabetical ordering of the string field.
*/
int
node_compare(node1, node2)
char *node1, *node2;
{
 return strcmp(((struct node *)node1)->string,
 ((struct node *) node2)->string);
}
/*
 This routine prints out a node, the second time
 twalk encounters it or if it is a leaf.
*/
```

```

void
print_node(node, order, level)
struct node **node;
VISIT order;
int level;
{
 if (order == postorder || order == leaf) {
 (void)printf("string = %s, length = %d\n",
 STRSZ/NODSZ,
 (*node)->string, (*node)->length);
 }
}

```

## Diagnostics

---

A NULL pointer is returned by **tsearch** if there is not enough space available to create a new node.

A NULL pointer is returned by **tfind** and **tdelete** if *rootp* is NULL on entry.

If the datum is found, both **tsearch** and **tfind** return a pointer to it. If not, **tfind** returns NULL, and **tsearch** returns a pointer to the inserted item.

## Warnings

---

The *root* argument to the **twalk** routine is one level of indirection less than the *rootp* arguments to the **tsearch** and **tdelete** routines.

There are two nomenclatures used to refer to the order in which tree nodes are visited. The **tsearch** function uses **preorder**, **postorder**, and **endorder** to respectively refer to visiting a node before any of its children, after its left child and before its right, and after both its children. The alternate nomenclature uses **preorder**, **inorder**, and **postorder** to refer to the same visits, which could result in some confusion over the meaning of **postorder**.

## Note

---

If the calling function alters the pointer to the root, results are unpredictable.

## See also

---

**bsearch(S)**, **hsearch(S)**, **lsearch(S)**

## Standards conformance

---

**tdelete**, **tfind**, **tsearch** and **twalk** are conformant with:  
X/Open Portability Guide, Issue 3, 1989.

# ttyname, isatty

---

get name of a terminal

## Syntax

---

```
cc ... -lc

#include <stdlib.h>
#include <unistd.h>

char *ttyname (fildes)
int fildes;

int isatty (fildes)
int fildes;
```

## Description

---

**ttyname** - get terminal device pathname

**isatty** - test for a terminal device

The **ttyname** routine returns a pointer to a string containing the null-terminated pathname of the terminal device associated with file descriptor *fildes*. The return value may point to static data whose content is overwritten by each call.

The **isatty** routine tests whether *fildes*, an open file descriptor, is associated with a terminal device. **isatty** returns 1 if *fildes* is associated with a terminal device, 0 otherwise with **errno** set.

## Files

---

/dev/\*

## Diagnostics

---

On unsuccessful completion **ttyname**, returns a null pointer if *fildes* does not describe a terminal device in directory */dev* and **errno** is set.

If either **isatty** or **ttyname** fail, then **errno** contains one of:

- |          |                                                                 |
|----------|-----------------------------------------------------------------|
| [EBADF]  | The <i>fildes</i> argument is not a valid file descriptor.      |
| [ENOTTY] | The <i>fildes</i> argument does not refer to a terminal device. |

## **Note**

---

The return value points to static data whose content is overwritten by each call.

## **Standards conformance**

---

**isatty** and **ttyname** are conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

(S)

# ttyslot

---

find the slot in the utmp file of the current user

## Syntax

---

```
cc ... -lc

#include <stdlib.h>

int ttyslot ()
```

## Description

---

The **ttyslot** function returns the index of the current user's entry in the */etc/utmp* file. This is accomplished by actually scanning the file */etc/inittab* for the name of the terminal associated with the standard input, the standard output, or the error output (0, 1 or 2).

## Files

---

*/etc/inittab*  
*/etc/utmp*

## Diagnostics

---

A value of -1 is returned if an error was encountered while searching for the terminal name or if none of the above file descriptors is associated with a terminal device.

## See also

---

**getut(S), ttyname(S)**

## Standards conformance

---

The **ttyslot** routine has been withdrawn from XPG3.

# uadmin

administrative control

## Syntax

```
cc ... -lc

#include <sys/uadmin.h>
#include <prototype.h>

int uadmin (cmd, fcn, mdep)
int cmd, fcn;
char *mdep;
```

## Description

The **uadmin** system call provides control for basic administrative functions. This system call is tightly coupled to the system administrative procedures and is not intended for general use. The argument *mdep* is provided for machine-dependent use and is not defined except for the **A\_GETDEV** command.

As specified by *cmd*, the following commands are available:

**A\_GETDEV** Gets the major number of a block or character device named in *mdep*. Set *fcn* to **b** for block devices or to **c** for character devices. *mdep* is the two- to four-character device driver prefix name. These names are listed in the fourth column of the */etc/conf/cf.d/mdevice* file. If *mdep* is not specified, this message displays: **A\_GETDEV** requires the *mdep* argument.

An example call is: **uadmin(A\_GETDEV, b, hd)**. This call gets the block major device number for the hard disk device driver.

**A\_SHUTDOWN** The system is shutdown. All user processes are killed, the buffer cache is flushed, and the root file system is unmounted. The action to be taken after the system has been shut down is specified by *fcn*. The functions are generic; the hardware capabilities vary on specific machines.

**AD\_HALT** Halt the processor until a key is entered to reboot the system.

**AD\_BOOT** Interactive reboot; user is prompted for system name.



|                    |                     |                                                                                                                                                                                                                                               |
|--------------------|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                    | <b>AD_IBOOT</b>     | Interactive reboot; user is prompted for system name.                                                                                                                                                                                         |
|                    | <b>AD_PWRDOWN</b>   | Halt the processor; system remains down with no reboot option given.                                                                                                                                                                          |
| <b>A_REBOOT</b>    |                     | The system stops immediately without any further processing. The action to be taken next is specified by <i>fcn</i> as above.                                                                                                                 |
| <b>A_REMOUNT</b>   |                     | The root file system is mounted again after having been fixed. This should be used only during the startup process.                                                                                                                           |
| <b>A_SETCONFIG</b> |                     | Sets the system configuration. Currently, only the following function is available:                                                                                                                                                           |
|                    | <b>AD_PANICBOOT</b> | If <i>mdep</i> is 1, this function causes the machine to reboot automatically after a kernel panic. If <i>mdep</i> is 0, this function causes the machine to wait for user intervention at the console before rebooting after a kernel panic. |

The **uadmin** system call fails if any of the following is true:

|                |                                          |
|----------------|------------------------------------------|
| <b>[EPERM]</b> | The effective user ID is not super user. |
|----------------|------------------------------------------|

## *Diagnostics*

---

Upon successful completion, the value returned depends on *cmd* as follows:

|                    |                                  |
|--------------------|----------------------------------|
| <b>A_GETDEV</b>    | Major number of specified device |
| <b>A_SHUTDOWN</b>  | No return value                  |
| <b>A_REBOOT</b>    | No return value                  |
| <b>A_REMOUNT</b>   | 0                                |
| <b>A_SETCONFIG</b> | 0                                |

Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

## *Standards conformance*

---

The **uadmin** routine is an extension of AT&T System V provided by The Santa Cruz Operation.

# ulimit

get and set user limits

## Syntax

```
cc ... -lc

#include <ulimit.h>

long ulimit (cmd, ...)
int cmd;
```

## Description

The **ulimit** routine examines or changes process' file size limits. The *cmd* values available are:

- |                    |                                                                                                                                                                                                                                                                                                                                    |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>UL_GETFSIZE</b> | Return the process's regular file size limit in units of 512-byte blocks. The regular file size limit is inherited by child processes. Files of any size may be read.                                                                                                                                                              |
| <b>UL_SETFSIZE</b> | Set the regular file size limit for output operations of the process to the value of the optional second argument (must be of type long). Any process may decrease its own file size limit, but only a process with an effective user ID of super user may increase the limit. The new file size limit is returned by the routine. |

## Return value

Upon successful completion, the **ulimit** routine returns the value of the limit requested. If the routine fails, **ulimit** returns a value of -1 and **errno** is set to indicate the appropriate error.

## Diagnostics

If one of the following conditions occurs, the **ulimit** routine fails (no values will be changed) and **errno** is set to the corresponding value:

- |                 |                                                                                                              |
|-----------------|--------------------------------------------------------------------------------------------------------------|
| <b>[EINVAL]</b> | The <i>cmd</i> argument is invalid.                                                                          |
| <b>[EPERM]</b>  | A process with an effective user ID other than super user attempted to increase its regular file size limit. |

*ulimit(S)*

## ***See also***

---

**write(S)**

## ***Standards conformance***

---

**ulimit** conforms with: X/Open Portability Guide, Issue 3, 1989.

# umask

set and get file creation mask

## Syntax

```
cc ... -lc

#include <sys/types.h>
#include <sys/stat.h>

mode_t umask (cmask)
mode_t cmask;
```

## Description

The **umask** routine sets the process's file mode creation mask to the argument *cmask* and returns the previous value of the mask. Only the file permission mode bits of *cmask* (see *stat.h*) and of the process's file mode creation mask are used.

The process's file mode creation mask is used by **creat(S)**, **open(S)**, **mkdir(S)**, and **mkfifo(S)** to turn off permission bits in each routine's specified *mode* argument. Bit positions that are set in *cmask* are cleared in the mode of the created file.

## Return value

The previous value of the file mode creation mask is returned. No errors are defined for the **umask** routine.

## See also

**creat(S)**, **mkdir(C)**, **mknod(S)**, **open(S)**

## Standards conformance

**umask** conforms with:

AT&T SVID Issue 2;  
X/Open Portability Guide, Issue 3, 1989;  
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);  
and NIST FIPS 151-1.

# umount

umount a file system

## Syntax

---

```
cc ... -lc

#include <prototypes.h>

int umount (file)
char *file;
```

## Description

---

The **umount** system call requests that a previously mounted file system contained on the block special device or directory identified by *file* be unmounted. *file* is a pointer to a path name. After unmounting the file system, the directory upon which the file system was mounted reverts to its ordinary interpretation.

The **umount** system call may be invoked only by the super user.

The **umount** system call fails if one or more of the following is true:

|             |                                                                                               |
|-------------|-----------------------------------------------------------------------------------------------|
| [EBUSY]     | A file on <i>file</i> is busy.                                                                |
| [EFAULT]    | <i>file</i> points to an illegal address.                                                     |
| [EINVAL]    | <i>file</i> does not exist, or is not mounted.                                                |
| [EMULTIHOP] | Components of the path pointed to by <i>file</i> require hopping to multiple remote machines. |
| [ENOENT]    | The named file does not exist.                                                                |
| [ENOLINK]   | <i>file</i> is on a remote machine, and the link to that machine is no longer active.         |
| [ENOTBLK]   | <i>file</i> is not a block special device.                                                    |
| [ENOTDIR]   | A component of the path-prefix is not a directory.                                            |
| [EPERM]     | The process's effective user ID is not super user.                                            |
| [EREMOTE]   | <i>file</i> is remote.                                                                        |

## Diagnostics

---

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

## ***See also***

---

**mount(S)**

## ***Standards conformance***

---

**umount** is conformant with:

AT&T SVID Issue 2; however this routine has been withdrawn from XPG3.

(S)

# uname

get name of current system

---

## Syntax

---

```
cc ... -lc

#include <sys/utsname.h>

int uname (name)
struct utsname *name;
```

## Description

---

The **uname** system call stores information identifying the current system in the structure pointed to by *name*.

The **uname** system call uses the **utsname** structure defined in *<sys/utsname.h>* whose members are:

```
struct utsname {
 char sysname[9];
 char nodename[9];
 char release[9];
 char version[9];
 char machine[9];
};
```

The **uname** system call returns a null-terminated character string naming the current system in the character array *sysname*. *nodename* contains the name that the system is known by on a communications network; however the inclusion of this member does not imply that it is sufficient information for interfacing to communications networks. *release* and *version* further identify the operating system. *machine* contains a standard name that identifies the hardware that the system is running on.

## See also

---

**uname(C)**, **\_\_scoinfo(S)**

## Return value

---

Upon successful completion, a non-negative value is returned.

## ***Standards conformance***

---

**uname** is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C  
Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

(S)



# ungetc

---

push character back into input stream

## Syntax

---

```
cc ... -lc

#include <stdio.h>

int ungetc (c, stream)
int c;
FILE *stream;
```

## Description

---

The **ungetc** function pushes the character specified by *c* (converted to an unsigned char) back onto the input stream pointed to by *stream*. The pushed-back characters are returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by *stream*) to a file-positioning function ( **fseek(S)** or **rewind(S)**) discards any pushed-back characters for the stream. The external storage corresponding to the stream is unchanged.

One character of pushback is guaranteed. If the **ungetc** function is called too many times on the same stream without an intervening read or file-positioning operation on that stream, the operation may fail.

If the value of *c* equals that of the macro **EOF**, the operation fails and the input stream is unchanged.

A successful call to the **ungetc** function clears the end-of-file indicator for the stream. The value of the file-position indicator for the stream after reading or discarding all pushed-back characters is the same as it was before the characters were pushed back. One character of pushback is guaranteed, provided something has already been read from the stream and the stream is actually buffered. In the case that *stream* is *stdin*, and the stream is buffered, one character may be pushed back onto the buffer without a previous read statement. For a stream, the value of its file-position indicator after a successful call to the **ungetc** function is unspecified until all pushed-back characters are read or discarded.

## Diagnostics

---

**ungetc** returns **EOF** if it cannot insert the character.

## Notes

---

When **stream** is *stdin*, one character may be pushed back onto the buffer without a previous read statement.

## See also

---

**fseek(S), getc(S), setbuf(S), stdio(S)**

## Standards conformance

---

**ungetc** is conformant with:

AT&T SVID Issue 2;  
X/Open Portability Guide, Issue 3, 1989;  
ANSI X3.159-1989 Programming Language — C;  
Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);  
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);  
and NIST FIPS 151-1.

(S)

# unlink

---

remove directory entry

## Syntax

---

```
cc ... -lc

int unlink (path)
char *path;
```

## Description

---

The **unlink** function removes the link named by the pathname pointed to by *path* and decrements the link count of the file referenced by the link.

When the file's link count becomes zero and no process has the file open, the space occupied by the file is freed and the file is no longer accessible. If one or more processes have the file open when the last link is removed, the link is removed before **unlink** returns, but the removal of the file contents is postponed until all references to the file have been closed.

Upon successful completion, **unlink** marks for update the "st\_ctime" and "st\_mtime" fields of the parent directory. Also, if the file's link count is not zero, the "st\_ctime" and "st\_mtime" fields of the file are marked for update.

If all of the following conditions are true, a file is not unlinked:

1. The parent directory has the sticky bit set
2. The file is not writable by the user
3. The user does not own the parent directory
4. The user does not own the file
5. The user is not root

When all links to a file have been removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, the removal is postponed until all references to the file have been closed.

## Return value

---

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

## Diagnostics

---

The **unlink** function will fail and not unlink the file if:

- |                       |                                                                                                                                                                 |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>[EACCES]</b>       | Search permission is denied for a component of the path prefix, or write permission is denied on the directory containing the link to be removed.               |
| <b>[EBUSY]</b>        | The entry to be unlinked is the mount point for a mounted file system.                                                                                          |
| <b>[EINTR]</b>        | A signal was caught during the <b>unlink</b> system call.                                                                                                       |
| <b>[EMULTIHOP]</b>    | Components of <i>path</i> require hopping to multiple remote machines.                                                                                          |
| <b>[ENAMETOOLONG]</b> | The length of the <i>path</i> string exceeds <b>PATH_MAX</b> or a path-name component is longer than <b>NAME_MAX</b> while <b>_POSIX_NO_TRUNC</b> is in effect. |
| <b>[ENOENT]</b>       | The named file does not exist, or the <i>path</i> argument points to an empty string.                                                                           |
| <b>[ENOLINK]</b>      | <i>path</i> points to a remote machine and the link to that machine is no longer active.                                                                        |
| <b>[ENOTDIR]</b>      | A component of the path prefix is not a directory.                                                                                                              |
| <b>[EPERM]</b>        | The named file is a directory and the effective user ID of the process is not super user.                                                                       |
| <b>[EROFS]</b>        | The directory entry to be unlinked is part of a read-only file system.                                                                                          |
| <b>[ETXTBSY]</b>      | The entry to be unlinked is the last link to a pure procedure (shared text) file that is being executed.                                                        |

## See also

---

**close(S), link(S), open(S), rm(C), stat(S)**

## Standards conformance

---

**unlink** is conformant with:

AT&T SVID Issue 2;  
X/Open Portability Guide, Issue 3, 1989;  
Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);  
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);  
and NIST FIPS 151-1.

# ustat

get file system statistics

## Syntax

```
cc ... -lc

#include <sys/types.h>
#include <ustat.h>

int ustat (dev, buf)
dev_t dev;
struct ustat *buf;
```

## Description

The **ustat** system call returns information about a mounted file system. *dev* is a device number identifying a device containing a mounted file system. *buf* is a pointer to a **ustat** structure that includes the following elements:

```
daddr_t f_tfree; /* Total free blocks */
ino_t f_tinode; /* Number of free inodes */
char f_fname[6]; /* Filsys name */
char f_fpack[6]; /* Filsys pack name */
```

The last two fields, *f\_fname* and *f\_fpack* may not have significant information on all systems, and, in that case, contain the null character.

The **ustat** system call fails if one or more of the following is true:

- [ECOMM]      *dev* is on a remote machine and the link to that machine is no longer active.
- [EFAULT]     *buf* points outside the process's allocated address space.
- [EINTR]      A signal was caught during a **ustat** system call.
- [EINVAL]     *dev* is not the device number of a device containing a mounted file system.
- [ENOLINK]    *dev* is on a remote machine and the link to that machine is no longer active.

## Diagnostics

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

## *See also*

---

fs(F), stat(S), statfs(S)

## *Standards conformance*

---

ustat is conformant with:

AT&T SVID Issue 2; however, this routine has been withdrawn from XPG3.

(S)

# utime

set file access and modification times

## Syntax

```
cc ... -lc

#include <sys/types.h>
#include <utime.h>

int utime (path, times)
char *path;
struct utimbuf *times;
```

## Description

The **utime** function sets the access and modification times of the file named by the *path* argument.

If *times* is **NULL**, the access and modification times of the file are set to the current time. The effective user ID of the process must match the owner of the file, or the process must have write permission to the file or have super user privileges to use **utime** in this manner.

If *times* is not **NULL**, *times* is interpreted as a pointer to a **utimbuf** structure and the access and modification times are set to the values contained in the designated structure. Only a process with effective user ID equal to the user ID of the file or a process with super user privileges may use **utime** this way.

The **utimbuf** structure is defined by the header *<utime.h>*. The times in the structure **utimbuf** are measured in seconds since the "Epoch", 00:00:00 Greenwich Mean Time (GMT), January 1, 1970.

```
struct utimbuf {
 time_t actime; /* access time */
 time_t modtime; /* modification time */
};
```

The **utime** system call fails if one or more of the following is true:

- [EACCES]** The effective user ID is not super user and not the owner of the file, and *times* is null and write access is denied, or search permission is denied by a component of the path prefix.
- [EINTR]** A signal was caught during the **utime** system call.
- [EMULTIHOP]** Components of *path* require hopping to multiple remote machines.

**[ENAMETOOLONG]**

The length of the *path* argument exceeds `PATH_MAX` or a pathname component is longer than `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.

**[ENOENT]**

The named file does not exist or the *path* argument points to an empty string.

**[ENOLINK]**

*path* points to a remote machine, and the link to that machine is no longer active.

**[ENOTDIR]**

A component of the path prefix is not a directory.

**[EPERM]**

The effective user ID has write access to the file, but is not super user and not the owner of the file, and *times* is not `NULL`.

**[EROFS]**

The file system containing the file is mounted read-only.

## Diagnostics

---

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned, and `errno` is set to indicate the error, and the file times are not affected.

## See also

---

`stat(S)`

## Standards conformance

---

**utime** is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.



# varargs, stdarg

---

variable argument list

## *Syntax*

---

### *varargs.h Syntax*

```
#include <varargs.h>

function (va_alist)
va_dcl
va_list pvar;

va_start (pvar);
f = va_arg (pvar, type);
va_end (pvar);
```

### *stdarg.h Syntax*

```
#include <stdarg.h>

function (parmN, ...)
va_list pvar;

va_start (pvar, parmN);
f = va_arg (pvar, type);
va_end (pvar);
```

## *Description*

---

This man page describes two header files that allow the manipulation of variable argument lists. *varargs.h* is provided for backward compatibility while *stdarg.h* is provided for greater portability.

### *varargs.h Description*

These macros provide a means of writing portable procedures that accept variable argument lists. Routines having variable argument lists (such as `printf(S)`) that do not use either `stdarg` or `varargs` are inherently nonportable, since different machines use different argument passing conventions.

`va_alist` is used in a function header to denote a variable argument list.

`va_dcl` is a declaration for `va_alist`. Note that there is no semicolon after `va_dcl`.

**va\_list** is a type which can be used for the variable **pvar**, which is used to traverse the list. One such variable must always be declared.

**va\_start** (*pvar*) is called to initialize **pvar** to the beginning of the list.

**va\_arg** (*pvar, type*) returns the next argument in the list pointed to by **pvar**. *type* is the type the argument is expected to be. Different types can be mixed but it is up to the routine to know what type of argument is expected since it cannot be determined at runtime.

**va\_end** (*pvar*) is used to finish up.

Multiple traversals, each bracketed by **va\_start ... va\_end**, are possible.

### ***stdarg.h Description***

These macros provide a means of writing portable procedures that accept variable argument lists. Routines having variable argument lists (such as **printf**(S)) that do not use either **stdarg** or **varargs** are inherently nonportable, since different machines use different argument passing conventions.

**va\_list** is a type which can be used for the variable **pvar**, which is used to traverse the list. One such variable must always be declared.

**va\_start** (*pvar, parmN*) is called to initialize **pvar** to the beginning of the list. **parmN** is the identifier of the rightmost parameter in the variable parameter list in the function definition.

**va\_arg** (*pvar, type*) returns the next argument in the list pointed to by **pvar**. *type* is the type the argument is expected to be. Different types can be mixed but it is up to the routine to know what type of argument is expected since it cannot be determined at runtime.

**va\_end** (*pvar*) is used to finish up.

Multiple traversals, each bracketed by **va\_start ... va\_end**, are possible.

## Examples

---

The following examples illustrate the use of the macros specified in each header file.

### *varargs.h* example

The following example illustrates the use of the macros specified in *varargs.h*.

```
#include <stdio.h>
#include <varargs.h>

/*
 * the first argument is an int which tells how many pairs follow.
 * the pairs are doubles and character pointers
 *
 * remember that when variables are passed to functions
 * floats are promoted to doubles and chars to ints.
 */

void show(n, va_alist)
int n;
va_dcl
{
 va_list ap;
 int i;
 double f;
 char *p;

 va_start(ap);
 for (i = 0; i < n; ++i) {
 f = va_arg(ap, double);
 p = va_arg(ap, char *);
 printf("%4.1f %s\n", f, p);
 }
 va_end(ap);
}

main()
{
 show(2, 3.1, "but", 4.1, "end");
 show(1, 5.9, "hello");
 show(4, 6.2, "oops", 5.3, "blah", 5.1, "lovely",
 2.3, "madrigal");
}
```

## *stdarg.h* example

The following example illustrates the use of the macros specified in *stdarg.h*.

```

#include <stdio.h>
#include <stdarg.h>

/*
 * the first argument is an int which tells how many pairs follow.
 * the pairs are doubles and character pointers
 *
 * remember that when variables are passed to functions
 * floats are promoted to doubles and chars to ints.
 */

void show(int n, ...)
{
 va_list ap;
 int i;
 double f;
 char *p;

 va_start(ap, n);
 for (i = 0; i < n; ++i) {
 f = va_arg(ap, double);
 p = va_arg(ap, char *);
 printf("%4.1f %s\n", f, p);
 }
 va_end(ap);
}

main()
{
 show(2, 3.1, "but", 4.1, "end");
 show(1, 5.9, "hello");
 show(4, 6.2, "oops", 5.3, "blah", 5.1, "lovely",
 2.3, "madrigal");
}

```

## **Notes**

---

It is up to the called routine to determine how many arguments there are, since it is not possible to determine this from the stack frame. For example, **excel** passes a 0 to signal the end of the list. **printf** can tell how many arguments are supposed to be there by the format of the list.

## **Standards conformance**

---

**stdarg**, **va\_arg**, **va\_end**, **va\_list**, and **va\_start** are conformant with:

ANSI X3.159-1989 Programming Language — C;

X/Open Portability Guide, Issue 3, 1989;

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1).

**varargs** is conformant with:

AT&T SVID Issue 2.

# vprintf, vfprintf, vsprintf

---

print formatted output of a varargs argument list

## Syntax

---

```
cc ... -lc

#include <stdio.h>
#include <varargs.h>

int vprintf (format, ap)
const char *format;
va_list ap;

int vfprintf (stream, format, ap)
FILE *stream;
const char *format;
va_list ap;

int vsprintf (s, format, ap)
char *s;
const char *format;
va_list ap;
```

## Description

---

The **vprintf**, **vfprintf**, and **vsprintf** functions are the same as **printf**, **fprintf**, and **sprintf** respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by **varargs(S)**.

## **Example**

---

The following demonstrates the use of **vfprintf** to write an error routine.

```

#include <stdio.h>
#include <varargs.h>

...

/*
 * error should be called by:
 * error(function_name, format, arg1, arg2...); */

/*VARARGS*/
void
error(va_alist)
/* Note that the function_name and format arguments cannot be
 * separately declared because of the definition of varargs. */
va_dcl
{
 va_list args;
 char *fmt;

 va_start(args);
 /* print out name of function causing error */
 (void)fprintf(stderr, "ERROR in %s: ", va_arg(args, char *));
 fmt = va_arg(args, char *);
 /* print out remainder of message */
 (void)vfprintf(stderr, fmt, args);
 va_end(args);
 (void)abort();
}

```

## **See also**

---

**printf(S), varargs(S)**

## **Standards conformance**

---

**vfprintf, vprintf** and **vsprintf** are conformant with:

AT&T SVID Issue 2;  
 X/Open Portability Guide, Issue 3, 1989;  
 and ANSI X3.159-1989 Programming Language — C.

# wait, waitpid

wait for child process to stop or terminate

## Syntax

```
cc ... -lc

#include <sys/types.h>
#include <sys/wait.h>

pid_t wait (stat_loc)
int *stat_loc;

pid_t waitpid (pid, stat_loc, options)
pid_t pid;
int *stat_loc, options;
```

## Description

**wait** - suspends calling process

**waitpid** - suspends calling process of *pid*

The **wait** and **waitpid** system calls suspend the calling process until one of the immediate children terminates, or until a process that is being traced stops. The **wait** system call returns prematurely if a signal is received. If a child process stops or terminates prior to the call on **wait**; return is immediate. If status information is available for two or more child processes, the order in which their status is reported is unspecified.

The **waitpid** function behaves identically to the **wait** function, if the *pid* argument has a value of -1 and the *options* argument has a value of zero. Otherwise, its behavior is modified by the values of the *pid* and *options* arguments.

The *pid* argument specifies a set of child processes for which status is requested. **waitpid** only returns the status of a child process from this set.

1. If *pid* is equal to -1, status is requested for any child process. In this respect, **waitpid** is then equivalent to **wait**.
2. If *pid* is greater than zero, it specifies the process ID of a single child process for which status is requested.
3. If *pid* is equal to zero, status is requested for any child process whose process group ID is equal to that of the calling process.
4. If *pid* is less than -1, status is requested for any child process whose process group ID is equal to the absolute value of *pid*.



The *options* argument is constructed from the bitwise inclusive or of zero or more of the following flags, defined in the header `<sys/wait.h>`:

- WNOHANG**      The `waitpid` function does not suspend execution of the calling process if status is not immediately available for one of the child processes specified by *pid*.
- WUNTRACED**    If the implementation supports job control, the status of any child processes specified by *pid* that are stopped, and whose status has not yet been reported since they stopped, are also reported to the requesting process.

If `wait` or `waitpid` return because the status of a child process is available, these functions return a value equal to the process ID of the child process.

In this case, if the value of the argument *stat\_loc* is not NULL, information is stored in the location pointed to by *stat\_loc*. If and only if the status returned is from a terminated child process that returned a value of zero from `main` or passed a value of zero as the *status* argument to `_exit` or `exit`, the value stored at the location pointed to by *stat\_loc* is zero. Regardless of its value, this information may be interpreted using the following macros, which are defined in `<sys/wait.h>` and evaluate to integral expressions; the *stat\_val* argument is the integer value pointed to by *stat\_loc*.

- WIFEXITED(*stat\_val*)**      Evaluates to a non-zero value if status was returned for a child process that terminated normally.
- WEXITSTATUS(*stat\_val*)**    If the value of **WIFEXITED(*stat\_val*)** is non-zero, this macro evaluates to the low-order 8 bits of the *status* argument that the child process passed to `_exit` or `exit`, or the value the child process returned from `main`.
- WIFSIGNALED(*stat\_val*)**    Evaluates to a non-zero value if status was returned for a child process that terminated due to the receipt of a signal that was not caught (see `signal(S)`).
- WTERMSIG(*stat\_val*)**      If the value of **WIFSIGNALED(*stat\_val*)** is non-zero, this macro evaluates to the number of the signal that caused the termination of the child process.
- WIFSTOPPED(*stat\_val*)**    Evaluates to a non-zero value if status was returned for a child process that is currently stopped.
- WSTOPSIG(*stat\_val*)**      If the value of **WIFSTOPPED(*stat\_val*)** is non-zero, this macro evaluates to the number of the signal that caused the child process to stop.

If the information stored at the location pointed to by *stat\_loc* was stored there by a call to the `waitpid` function that specified the **WUNTRACED** flag, exactly one of the macros **WIFEXITED(*stat\_loc*)**, **WIFSIGNALED(*stat\_loc*)**,

and `WIFSTOPPED(stat_loc)` evaluates to a non-zero value. If the information stored at the location pointed to by `stat_loc` was stored there by a call to the `waitpid` function that did not specify the `WUNTRACED` flag or by a call to the `wait` function, exactly one of the macros `WIFEXITED(*stat_loc)` and `WIFSIGNALED(*stat_loc)` evaluates to a non-zero number.

If `stat_loc` is non-zero, 16 bits of information called status are stored in the low order 16 bits of the location pointed to by `stat_loc`. `status` can be used to differentiate between stopped and terminated child processes and if the child process terminated, status identifies the cause of termination and passes useful information to the parent. This is accomplished in the following manner:

If the child process stopped, the high order 8 bits of status contain the number of the signal that caused the process to stop, and the low order 8 bits are set equal to 0177.

If the child process terminated due to an `exit` call, the low order 8 bits of status are zero, and the high order 8 bits contain the low order 8 bits of the argument that the child process passed to `exit` (see `exit(S)`).

If the child process terminated due to a signal, the high order 8 bits of status are zero, and the low order 8 bits contain the number of the signal that caused the termination. In addition, if the low order seventh bit (that is, bit 200) is set, a "core image" has been produced (see `signal(S)`).

If a parent process terminates without waiting for its child processes to terminate, the parent process ID of each child process is set to 1. This means the initialization process inherits the child processes.

## Return value

---

If the `wait` or `waitpid` functions return because the status of a child process is available, these functions return a value equal to the process ID of the child process for which status is reported. If the `wait` or `waitpid` functions return due to the delivery of a signal to the calling process, a value of -1 is returned and `errno` is set to `[EINTR]`. If the `waitpid` function was invoked with `WNOHANG` set in `options`, it has at least one child process specified by `pid` for which status is not available, and status is not available for any process specified by `pid`, a value of zero is returned. Otherwise, a value of -1 is returned, and `errno` is set to indicate the error.

The `wait` function fails and returns -1 if one or more of the following is true:

- [ECHILD] The calling process has no existing unwaited-for child processes.
- [EINTR] The function was interrupted by a signal. The value of the location pointed to by `stat_loc` is undefined.

If any of the following conditions occur, the **waitpid** function returns -1 and sets **errno** to the corresponding value:

- [ECHILD]** The process or process group defined by *pid* does not exist or is not a child of the calling process.
- [EINTR]** The function was interrupted by a signal. The value of the location pointed to by *stat\_loc* is undefined.
- [EINVAL]** The value of the **options** argument is not valid.

## ***Warning***

---

The **wait** system call fails and its actions are undefined if *stat\_loc* points to an invalid address.

## ***See also***

---

**exec(S), exit(S), fork(S), pause(S), ptrace(S), signal(S), times(S)**

## ***Standards conformance***

---

**wait** is conformant with:

AT&T SVID Issue 2;  
X/Open Portability Guide, Issue 3, 1989;  
IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1); and NIST FIPS 151-1.

**waitpid** is conformant with:

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);  
X/Open Portability Guide, Issue 3, 1989;  
and NIST FIPS 151-1.

# waitsem, nbwaitsem

awaits and checks access to a semaphore-governed resource

## Syntax

```
cc ... -lx

int waitsem (sem_num)
int sem_num;

int nbwaitsem (sem_num)
int sem_num;
```

## Description

**waitsem** - gives access to a semaphore-governed resource

**nbwaitsem** - waits for access to a semaphore-governed resource

**waitsem** gives the calling process access to the resource governed by the semaphore **sem\_num**. If the resource is in use by another process, **waitsem** puts the process to sleep until the resource becomes available; **nbwaitsem** returns the error ENAVAIL. **waitsem** and **nbwaitsem** are used in conjunction with **sigsem** to allow synchronization of processes wishing to access a resource. One or more processes may wait on the given semaphore (using **waitsem**) and are put to sleep until the process that currently has access to the resource issues **sigsem**. **sigsem** causes the next process in line on the semaphore's queue to be rescheduled for execution. The semaphore's queue is organized in first in first out FIFO order.

## System compatibility

**waitsem** can only be used to synchronize semaphores created under UNIX Version 3, not for UNIX System V semaphores.

## Diagnostics

**waitsem** returns the value (int) -1 if an error occurs. If **sem\_num** has not been previously opened by a call to **opensem** or **creatsem**, **errno** is set to EBADF. If **sem\_num** does not refer to a semaphore type file, **errno** is set to ENOTNAM. All processes waiting (or attempting to wait) on the semaphore return with **errno** set to ENAVAIL when the process controlling the semaphore exits without relinquishing control (thereby leaving the resource in an undeterminate state). If a process does two calls to **waitsem** in a row without doing an intervening **sigsem**, **errno** is set to EINVAL.

*waitsem(S)*

## ***Notes***

---

This feature is a XENIX specific enhancement and may not be present in all UNIX implementations. This routine must be linked with the linker option **-lx**.

## ***See also***

---

**creatsem(S), opensem(S), sigsem(S)**

## ***Standards conformance***

---

**waitsem** and **nbwaitsem** are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.

# write

write on a file

## Syntax

```
cc ... -lc

int write (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

## Description

The **write** function attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the *fildes* descriptor.

*fildes* is a file descriptor obtained from a **creat**(S), **open**(S), **dup**(S), **fcntl**(S), or **pipe**(S) system call.

If *nbyte* is zero, **write** returns zero and has no other results if the file is a regular file.

On a regular file or other file capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file offset. If this incremented file offset is greater than the length of the file, the length of the file is set to this file offset.

If the **O\_SYNC** flag of the file status flags is set and *fildes* refers to a regular file, a successful **write** does not return until the data is delivered to the underlying hardware.

Upon successful return from **write**, the file offset is incremented by the number of bytes actually written.

On a file not capable of seeking, writing always takes place starting at the current position. This value of a file offset associated with such a device is undefined.

If the **O\_APPEND** flag of the file status flags is set, the file offset is set to the end of the file prior to each write.

If a **write** requests that more bytes be written than there is room for (for example, the *ulimit* or physical end of a medium), only as many bytes as there is room for are written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes returns 20. The next write of a nonzero number of bytes gives a failure return (except as noted later).

On successful completion, the **write** function returns the actual number of bytes written to the file associated with *fil-des*. This number is never greater than *nbyte*.

If a call to **write** is interrupted by a signal before it writes any data, **write** returns -1 with **errno** set to **EINTR**.

If **write** is interrupted by a signal after it successfully writes some data, either **write** returns -1 with **errno** set to **EINTR**, or **write** returns the number of bytes written. A write to a pipe or FIFO (first in first out) never returns with **errno** set to **EINTR** if it has transferred any data and *nbyte* is less than or equal to **PIPE\_BUF**.

If the value of *nbyte* is greater than **INT\_MAX**, the result is implementation defined.

Write requests to a pipe or FIFO are handled the same as a regular file with the following exceptions:

- There is no file offset associated with a pipe, hence each write request is appended to the end of the pipe.
- Write requests of **PIPE\_BUF** bytes or less are not interleaved with data from other processes doing writes on the same pipe. Writes of greater than **PIPE\_BUF** bytes may have data interleaved on arbitrary boundaries with writes by other processes. This can occur whether or not the **O\_NONBLOCK** flag of the file status flags is set.
- If the **O\_NONBLOCK** flag is clear, a write request may cause the process to block, but on normal completion it returns *nbyte*.
- If the **O\_NONBLOCK** flag is set, **write** requests are handled differently, in the following ways: the **write** function does not block the process; write requests for **PIPE\_BUF** or fewer bytes either succeeds completely and returns *nbyte*, or returns -1 with **errno** set to **EAGAIN**. A write request for greater than **PIPE\_BUF** bytes, either transfers what it can and returns the number of bytes written, or transfers no data and returns -1 with **errno** set to **EAGAIN**. Also, if a request is greater than **PIPE\_BUF** bytes and all data previously written to the pipe has been read, **write** transfers at least **PIPE\_BUF** bytes.

When attempting to write to a file descriptor (other than a pipe or FIFO) that supports nonblocking writes and cannot accept the data immediately:

- If the **O\_NONBLOCK** flag is clear, **write** blocks until the data is accepted.
- If the **O\_NONBLOCK** flag is set, **write** does not block the process. If some data can be written without blocking the process, **write** writes what it can and returns the number of bytes written. Otherwise, **write** returns -1 with **errno** set to **EAGAIN**.

Upon successful completion, the **write** function marks for update the **st\_ctime** and **st\_mtime** fields of the file, and if the file is a regular file, the **S\_ISUID** and **S\_ISGID** bits of the file mode may be cleared.

A write to a **STREAMS** file can fail if an error message has been received at the stream head. In this case, **errno** is set to the value included in the error message.

## Errors

---

The write system call fails and the file pointer remains unchanged if one or more of the following is true:

- [EAGAIN]**      The **O\_NONBLOCK** flag is set for the file descriptor and the process would be delayed in the write operation, or an attempt was made to write to a stream that cannot accept data with the **O\_NONBLOCK** flag set.
- [EBADF]**      *fildes* is not a valid file descriptor open for writing.
- [EDEADLK]**    The write was going to go to sleep and cause a deadlock situation to occur.
- [EFBIG]**      An attempt was made to write a file that exceeds the process's file size limit or the maximum file size (see **ulimit(S)**).
- [EINTR]**      The write operation was terminated due to the receipt of a signal and either no data was transferred or the implementation does not report partial transfers for this file.
- [EINVAL]**    Attempt to write to a stream linked below a multiplexer.
- [EIO]**        The implementation supports job control, the process is a member of a background process group attempting to write to its controlling terminal, **TOSTOP** is set, the process is neither ignoring not blocking **SIGTTOU** and the process group of the process is orphaned.
- [ENOLCK]**    The system record lock table was full, so the write could not go to sleep until the blocking record lock was removed.
- [ENOLINK]**    *fildes* is on a remote machine and the link to that machine is no longer active.
- [ENOSPC]**    During a write to an ordinary file, there is no free space left on the device.
- [EPIPE]**      (With a **SIGPIPE** signal.) An attempt is made to write to a pipe or FIFO not open for reading by any process.
- [ERANGE]**    Attempt to write to a stream with *nbyte* outside specified minimum and maximum write range, and the minimum value is non-zero.
- [ENXIO]**      A hangup occurred on the stream being written to, a request was made to a nonexistent device, or the request was outside the capabilities of the device.



*write(S)*

## ***Note***

---

**O\_NONBLOCK** previously described on this manual page has been replaced in XPG3 and POSIX by **O\_NONBLOCK**. **O\_NONBLOCK** caused 0 (zero) to be returned if a **write** request would block; **O\_NONBLOCK** causes -1 to be returned with **errno** set to **EINTR**.

## ***Diagnostics***

---

Upon successful completion the number of bytes actually written is returned. Otherwise, -1 is returned, and **errno** is set to indicate the error.

## ***See also***

---

**creat(S), dup(S), fcntl(S), lseek(S), open(S), pipe(S), ulimit(S)**

## ***Standards conformance***

---

**write** is conformant with:

AT&T SVID Issue 2;

X/Open Portability Guide, Issue 3, 1989;

Intel386 Binary Compatibility Specification, Edition 2 (iBCSe2);

IEEE POSIX Std 1003.1-1990 System Application Program Interface (API) [C Language] (ISO/IEC 9945-1);

and NIST FIPS 151-1.

# xlist, fxlist

gets name list entries from files

## Syntax

```
#include <x.out.h>

int xlist (filename, xl)
char *filename;
struct xlist xl[];

#include <x.out.h>
#include <stdio.h>

int fxlist (fp, xl)
FILE *fp;
struct xlist xl [];
```

## Description

**xlist** - gets name list entries from file

**fxlist** - gets name list entries from file pointer

The **fxlist** and **xlist** macros examine the name list in an OMF or COFF executable and selectively extract values.

**fxlist** performs the same function as **xlist**, except that **fxlist** accepts a pointer to a previously opened file instead of a filename.

**xlist** examines the name list in the given executable output file and selectively extracts a list of values. The given executable file can be either OMF format or COFF. The name list structure **xl** consists of an array of **xlist** structures containing names, types, values, and segment values (if applicable). The list is terminated by either a pointer to a null name or a null pointer. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted into the next two fields. The segment value (if it exists) is inserted in the third field. If the name is not found, both entries are set to zero. See **a.out(FP)** for a discussion of the **xlist** structure.

**x.out** and **a.out** formats are understood, as well as 8086 relocatable and **x.out** segmented formats.

If the symbol table is in **a.out** format, and if the symbol name given to **xlist** is longer than eight characters, only the first eight characters are used for comparison. In all other cases, the name given to **xlist** must be the same length as a name list entry in order to match.

If two or more symbols happen to match the name given to **xlist**, then the type and value used are those of the last symbol found.

*xlist(S)*

## ***Diagnostics***

---

**xlist** returns -1 and sets all type entries to zero if the file cannot be read, is not an object file, or contains an invalid name list. Otherwise, **xlist** returns zero. A return value of zero does not indicate that any or all of the given symbols were found.

## ***See also***

---

**a.out(FP)**

## ***Standards conformance***

---

**xlist** and **fxlist** are not part of any currently supported standard; they are an extension of AT&T System V provided by the Santa Cruz Operation.