CTIX™ OPERATING SYSTEM MANUAL

Version C
Volume 3

This manual was prepared on a Convergent Technologies S/320 Computer System and was printed on an Apple LaserWriter II Laser Printer.

Second Edition (November 1989) 09-02264-01

TABLE OF CONTENTS: VOLUME 3

abs	return integer absolute value
assert	verify program assertion
bessel	Bessel functions
bsearch	binary search a sorted table
bstring	bit and byte string operations
byteorder	convert values between host and network byte order
clock	report CPU time used
conv	translate characters
crypt	generate hashing encryption
crypt	password and file encryption functions
ctermid	generate file name for terminal
ctime	convert date and time to string
ctype	character handling
curses	terminal screen handling and optimization package
cuserid	get character login name of the user
dbm	database subroutines
dial	establish an out-going terminal line connection
directory	directory operations
drand48	generate uniformly distributed pseudo-random numbers
dup2	duplicate an open file descriptor
ecvt	convert floating-point number to string
end	last locations in program
erf	error function and complementary error function
exp	exponential, logarithm, power, square root functions
fclose	close or flush a stream
ferror	stream status inquiries
floor	floor, ceiling, remainder, absolute value functions
fopen	open a stream
fpgetround	IEEE floating point environment control
fread	binary input/output
frexp	manipulate parts of floating-point numbers
fseek	reposition a file pointer in a stream
ftw	walk a file tree
gamma	log gamma function
getc	get character or word from a stream
getcwd	get path-name of current working directory
getenv	return value for environment name
getgrent	get group file entry
gethostbyname	get network host entry
getlogin	get login name
getnetent	get network entry
getopt	get option letter from argument vector
getpass	read a password
getprotoent	get protocol entry
getpw	get name from UID
getpwent	get password file entry
getrpcent	get rpc entry
getrpcport	get RPC port number
gets	get a string from a stream
getservent	get service entry
getspent	get shadow
getut	access utmp file entry

HOW TO USE THIS MANUAL

This second edition of the *CTIX Operating System Manual, Version C,* describes the commands, system calls, libraries, data files, and device interfaces that make up the CTIX Operating System for S/Series Computer Systems. This manual should always be your starting point when you need to find the documentation for a CTIX feature with which you are unfamiliar.

The manual consists of a large number of short entries, sometimes called ''the *man* pages,'' after the command that accesses the entries when they are kept online. Each entry briefly documents some feature of CTIX. Some features require longer documentation than an entry in this manual; such features have an entry that outlines the feature and cross-references the manual that documents the feature fully. Entries that do not refer to other manuals are self-contained and are the final word on the features they describe.

Organization of the manual. The entries are organized into seven sections in four volumes:

Volumes 1 and 2:
1. Commands and Application Programs.

Volume 3:
2. System Calls.
3. Subroutines and Libraries.

Volume 4:
4. File Formats.
5. Miscellaneous Facilities.
6. Games.
7. Special Files.

Within each section, entries are alphabetical by title, except for an *intro* entry at the beginning of each section.

Entry Title Conventions. An entry title looks like this example:

```
erf(3M)
|   ||
|   ||
|   |Entry Type
|   |
|   Section Number
|
Name
```

Name is the name of the entry. *Section Number* indicates the section that contains the entry. In this case, the entry is in Section 3, which is in Volume 2. *Entry Type* appears only on entries that belong to special categories; refer to the section's *intro* entry for an explanation. In this case, a reference to *intro*(3) would tell you that *erf*(3M) describes functions from the Math Library, which the C compiler does not load by default.

Finding the entry you need. To find out which entry you need, refer to the following guides:

- The Permuted Index. This indexes each significant word in each entry's description. It is useful when you have only a general notion what you're looking for. It is also useful when you know the name of the command or function you are interested in, but there is no entry by that name.
- The Table of Contents. This is a simple list of entries, by section, together with the entry descriptions. Volumes 1 and 2 have Tables of Contents for Section 1. Volume 3 has a Table of Contents for Sections 2 and 3. Volume 4 has a Table of Contents for Sections 4 through 7.
- The Table of Related Entries. For Volume 1 only. A table of entries organized so that related entries are grouped together.

Section organization. Each section begins with an *intro* entry, which provides important general information for that section.

Section 1, Commands and Application Programs, describes programs intended to be invoked directly by the user or by command language procedures, as opposed to subroutines, which are intended to be called by the user's programs. Commands generally reside in the directory **/bin** (for **bin**ary programs). Some programs also reside in **/usr/bin**, to save space in **/bin**. These directories are searched automatically by the command interpreter called the *shell*. Commands that were not transported from UNIX System V reside in **/usr/local/bin**; this directory is recommended for locally implemented programs. Some administrative commands reside in **/etc** and various other places. The **/etc** directory is searched automatically if you are logged in as root; otherwise use the full path name given under **SYNOPSIS** or change the **PATH** environment variable to include the command's directory.

Section 2, System Calls, describes the entries into the CTIX kernel, including the C language interfaces.

Section 3, Subroutines and Libraries, describes the available library functions or subroutines. Their binary versions reside in various system libraries in the directories **/lib** and **/usr/lib**. See *intro*(3) for descriptions of these libraries and the files in which they are stored.

Section 4, File Formats, documents the structure of particular kinds of files; for example, the format of the output of the link editor is given in *a.out*(4). Excluded are files used by only one command (for example, the assembler's intermediate files). In general, the C language **struct** declarations corresponding to these formats can be found in the directories **/usr/include** and **/usr/include/sys**.

Section 5, Miscellaneous Facilities, contains descriptions of character sets, macro packages, and other such information.

Section 6, Games, describes the games and educational programs that reside in the directory **/usr/games**.

Section 7, Special Files, discusses the characteristics of files that actually refer to input/output devices.

Entry organization. All entries are based on a common format, in which some parts are optional:

NAME	The **NAME** part gives the name(s) of the entry and briefly states its purpose.
SYNOPSIS	The **SYNOPSIS** part summarizes the use of the program being described. A few conventions are used, particularly in Section 1 (Commands and Application Programs):

Bold	**Boldface** strings are literals, and are to be typed just as they appear.
Regular	*Regular face* strings usually represent substitutable argument prototypes and program names found elsewhere in the manual.
[ ]	Square brackets around an argument prototype indicate that the argument is optional. When an argument prototype is given as "name" or "file," it always refers to a *file* name.
...	Ellipses are used to show that the previous argument prototype can be repeated.
– + =	A final convention is used by the commands themselves. An argument beginning with a minus (–), plus (+), or equal sign (=) is often taken to be some sort of flag argument, even if it appears in a position where a file name could appear. Therefore, it is unwise to have files whose names begin with –, +, or =.

DESCRIPTION	The **DESCRIPTION** part discusses the subject at hand.
EXAMPLE(S)	The **EXAMPLE(S)** part gives example(s) of usage, where appropriate.
FILES	The **FILES** part gives the file names that are built into the program.
SEEALSO	The **SEE ALSO** part gives pointers to related information.
DIAGNOSTICS	The **DIAGNOSTICS** part discusses the diagnostic indications that may be produced. Messages that are intended to be self-explanatory are not listed.
NOTES	The **NOTES** part gives information that might be helpful under the particular circumstance described.
WARNINGS	The **WARNINGS** part points out potential pitfalls.
BUGS	The **BUGS** part gives known bugs and sometimes deficiencies. Occasionally, the suggested fix is also described.

A table of contents is provided at the front of each of the four volumes, along with a complete permuted index derived from the tables. On each *index* line, the title of the entry to which that line refers is followed by the appropriate section number in parentheses. This is important because there is considerable duplication of names among the sections, arising principally from commands that exist only to exercise a particular system call.

PERMUTED INDEX

This index includes entries for all pages of Volumes 1 through 4. The entries themselves are based on the one-line descriptions or titles found in the NAME portion of each manual page; the significant words (keywords) of these descriptions are listed alphabetically down the center of the index.

The index is actually a keyword-in-context (KWIC) index that has three columns. To use the index, read the center column to look up specific commands by name or by subject topics. Note that the entry may begin in the left column or wrap around and continue into the left column. A period (.) marks the end of the entry, and a slash (/) indicates where the entry has been continued or truncated. The right column gives the manual page where the command or subject is described.

hpio: Hewlett-Packard	2645A terminal tape file/	hpio(1)
/special functions of DASI	300 and 300s terminals.	300(1)
for Interphase V/TAPE	3200 half-inch tape/ /interface	ipt(7)
l3tol, ltol3: convert between	3-byte integers and long/	l3tol(3C)
comparison. diff3:	3-way differential file	diff3(1)
paginator for the Tektronix	4014 terminal. 4014:	4014(1)
special functions of the DASI	450 terminal. 450: handle	450(1)
long integer and base-64/	a64l, l64a: convert between	a64l(3C)
	abort: generate a SIGABRT.	abort(3C)
value.	abs: return integer absolute	abs(3C)
adb:	absolute debugger.	adb(1)
abs: return integer	absolute value.	abs(3C)
/floor, ceiling, remainder,	absolute value functions.	floor(3M)
tiop: terminal	accelerator interface.	tiop(7)
t_accept:	accept a connect request.	t_accept(3n)
prevent LP requests.	accept, reject: allow or	accept(1M)
a directory for remote	access. adv: advertise	adv(1M)
of a file. touch: update	access and modification times	touch(1)
utime: set file	access and modification times.	utime(2)
accessibility of a file.	access: determine	access(2)
commands. graphics:	access graphical and numerical	graphics(1G)
sputl, sgetl:	access long integer data in a/	sputl(3X)
fusage: disk	access profiler.	fusage(1M)
sadp: disk	access profiler.	sadp(1M)
ldfcn: common object file	access routines.	ldfcn(4)
copy file systems for optimal	access time. dcopy:	dcopy(1M)
locking: exclusive	access to regions of a file.	locking(2)
/setutent, endutent, utmpname:	access utmp file entry.	getut(3C)
access: determine	accessibility of a file.	access(2)
enable or disable process	accounting. acct:	acct(2)
acctcon2: connect-time	accounting. acctcon1,	acctcon(1M)
acctprc1, acctprc2: process	accounting.	acctprc(1M)
turnacct: shell procedures for	accounting. /startup,	acctsh(1M)
/accton, acctwtmp: overview of	accounting and miscellaneous/	acct(1M)
accounting and miscellaneous	accounting commands. /of	acct(1M)
diskusg: generate disk	accounting data by user ID.	diskusg(1M)
acct: per-process	accounting file format.	acct(4)

search and print process	accounting file(s). acctcom:	acctcom(1)
acctmerg: merge or add total	accounting files.	acctmerg(1M)
summary from per-process	accounting records. /command	acctcms(1M)
wtmpfix: manipulate connect	accounting records. fwtmp,	fwtmp(1M)
runacct: run daily	accounting.	runacct(1M)
process accounting.	acct: enable or disable	acct(2)
file format.	acct: per-process accounting	acct(4)
per-process accounting/	acctcms: command summary from	acctcms(1M)
process accounting file(s).	acctcom: search and print	acctcom(1)
connect-time accounting.	acctcon1, acctcon2:	acctcon(1M)
acctwtmp: overview of/	acctdisk, acctdusg, accton,	acct(1M)
accounting files.	acctmerg: merge or add total	acctmerg(1M)
accounting.	acctprc1, acctprc2: process	acctprc(1M)
orderly release/ t_rcvrel:	acknowledge receipt of an	t_rcvrel(3n)
trig: sin, cos, tan, asin,	acos, atan, atan2:/	trig(3M)
killall: kill all	active processes.	killall(1M)
sag: system	activity graph.	sag(1G)
sar: sa1, sa2, sadc: system	activity report package.	sar(1M)
sar: system	activity reporter.	sar(1)
current SCCS file editing	activity. sact: print	sact(1)
report process data and system	activity. /time a command;	timex(1)
Dialers:	ACU/modem calling protocols.	Dialers(5)
random, hopefully interesting,	adage. fortune: print a	fortune(6)
	adb: absolute debugger.	adb(1)
acctmerg: merge or	add total accounting files.	acctmerg(1M)
putenv: change or	add value to environment.	putenv(3C)
/inet_netof: Internet	address manipulation routines.	inet(3)
getservaddr: get network	address of service host.	getservad(1M)
control. arp:	address resolution display and	arp(1M)
arp:	Address Resolution Protocol.	arp(7)
endpoint. t_bind: bind an	address to a transport	t_bind(3n)
allow synchronization of the/	adjtime: correct the time to	adjtime(2)
system.	adman: administer a CTIX	adman(1)
SCCS files.	admin: create and administer	admin(1)
network listener service	administration. nlsadmin:	nlsadmin(1M)
rfadmin: Remote File Sharing	administration.	rfadmin(1M)
uadmin:	administrative control.	uadmin(1M)
uadmin:	administrative control.	uadmin(2)
swap: swap	administrative interface.	swap(1M)
remote access.	adv: advertise a directory for	adv(1M)
	advent: explore Colossal Cave.	advent(6)
remote access. adv:	advertise a directory for	adv(1M)
fumount: forced unmount of an	advertised resource.	fumount(1M)
alarm: set a process	alarm clock.	alarm(2)
clock.	alarm: set a process alarm	alarm(2)
sendmail.	aliases: aliases file for	aliases(4)
aliases:	aliases file for sendmail.	aliases(4)
the data base for the mail	aliases file. /rebuild	newaliases(1)
t_alloc:	allocate a library structure.	t_alloc(3n)
change data segment space	allocation. brk, sbrk:	brk(2)
realloc, calloc: main memory	allocator. malloc, free,	malloc(3C)
mallinfo: fast main memory	allocator. /calloc, mallopt,	malloc(3X)
accept, reject:	allow or prevent LP requests.	accept(1M)
adjtime: correct the time to	allow synchronization of the/	adjtime(2)
process by changing/ renice:	alter priority of running	renice(1)
sort: sort	and/or merge files.	sort(1)
link editor output.	a.out: common assembler and	a.out(4)
introduction to commands and	application programs. intro:	intro(1)

maintainer for portable/	ar: archive and library	ar(1)
format.	ar: common archive file	ar(4)
number: convert	Arabic numerals to English.	number(6)
language. bc:	arbitrary-precision arithmetic	bc(1)
for portable archives. ar:	archive and library maintainer	ar(1)
cpio: format of cpio	archive.	cpio(4)
ar: common	archive file format.	ar(4)
header of a member of an	archive file. /the archive	ldahread(3X)
formats. convert: convert	archive files to common	convert(1)
an archive/ ldahread: read the	archive header of a member of	ldahread(3X)
2645A terminal tape file	archiver. /Hewlett-Packard	hpio(1)
tar: tape file	archiver.	tar(1)
maintainer for portable	archives. /archive and library	ar(1)
cpio: copy file	archives in and out.	cpio(1)
varargs: handle variable	argument list.	varargs(5)
formatted output of a varargs	argument list. /print	vprintf(3S)
command. xargs: construct	argument list(s) and execute	xargs(1)
getopt: get option letter from	argument vector.	getopt(3C)
expr: evaluate	arguments as an expression.	expr(1)
echo: echo	arguments.	echo(1)
bc: arbitrary-precision	arithmetic language.	bc(1)
number facts.	arithmetic: provide drill in	arithmetic(6)
display and control.	arp: address resolution	arp(1M)
Protocol.	arp: Address Resolution	arp(7)
ftp:	ARPANET file transfer program.	ftp(1)
expr: evaluate arguments	as an expression.	expr(1)
	as: common assembler.	as(1)
/attach and detach serial lines	as network interfaces.	slattach(1M)
/locate a terminal to use	as the virtual system console.	conlocate(1M)
characters. asa: interpret	ASA carriage control	asa(1)
and/ /gmtime, asctime, cftime,	ascftime, tzset: convert date	ctime(3C)
ascii: map of	ASCII character set.	ascii(5)
hd: hexadecimal and	ascii file dump.	hd(1)
set.	ascii: map of ASCII character	ascii(5)
long integer and base-64	ASCII string. /convert between	a64l(3C)
strings: extract the	ASCII text strings in a file.	strings(1)
ctime, localtime, gmtime,	asctime, cftime, ascftime,/	ctime(3C)
trig: sin, cos, tan,	asin, acos, atan, atan2:/	trig(3M)
output. a.out: common	assembler and link editor	a.out(4)
as: common	assembler.	as(1)
assertion.	assert: verify program	assert(3X)
setbuf, setvbuf:	assign buffering to a stream.	setbuf(3S)
system commands.	assist: assistance using CTIX	assist(1)
astgen: generate/modify	ASSIST menus and command/	astgen(1)
commands. assist:	assistance using CTIX system	assist(1)
print the list of blocks	associated with an. bcheck:	bcheck(1M)
/create device nodes for	assorted device types.	createdev(1M)
menus and command forms.	astgen: generate/modify ASSIST	astgen(1)
a later time.	at, batch: execute commands at	at(1)
/sin, cos, tan, asin, acos,	atan, atan2: trigonometric/	trig(3M)
cos, tan, asin, acos, atan,	atan2: trigonometric/ /sin,	trig(3M)
description file. queuedefs:	at/batch/cron queue	queuedefs(4)
double-precision/ strtod,	atof: convert string to	strtod(3C)
integer. strtol, atol,	atoi: convert string to	strtol(3C)
integer. strtol,	atol, atoi: convert string to	strtol(3C)
as/ slattach, sldetach:	attach and detach serial lines	slattach(1M)
resources. rmnttry:	attempt to mount remote	rmnttry(1M)
log of failed login	attempts. /usr/adm/loginlog:	loginlog(4)

wait:	await completion of process.	wait(1)
processing language.	awk: pattern scanning and	awk(1)
ungetc: push character	back into input stream.	ungetc(3S)
	back: the game of backgammon.	back(6)
back: the game of	backgammon.	back(6)
finc: fast incremental	backup.	finc(1M)
ckbupscd: check file system	backup schedule.	ckbupscd(1M)
frec: recover files from a	backup tape.	frec(1M)
	banner: make posters.	banner(1)
newaliases: rebuild the data	base for the mail aliases/	newaliases(1)
Sun rpc program number data	base. rpc:	rpc(4)
terminal capability data	base. termcap:	termcap(4)
terminal capability data	base. terminfo:	terminfo(4)
between long integer and	base-64 ASCII string. /convert	a64l(3C)
(visual) display editor	based on ex. /screen-oriented	vi(1)
from proto file; set links	based on. /out file lists	qlist(1)
portions of path names.	basename, dirname: deliver	basename(1)
later time. at,	batch: execute commands at a	at(1)
arithmetic language.	bc: arbitrary-precision	bc(1)
blocks associated with an.	bcheck: print the list of	bcheck(1M)
system initialization/ brc,	bcheckrc, drvload, powerfail:	brc(1M)
string operations. bcopy,	bcmp, bzero: bit and byte	bstring(3)
byte string operations.	bcopy, bcmp, bzero: bit and	bstring(3)
	bcopy: interactive block copy.	bcopy(1M)
	bdiff: big diff.	bdiff(1)
cb: C program	beautifier.	cb(1)
about the operating system for	beginning users. /information	starter(1)
j0, j1, jn, y0, y1, yn:	Bessel functions. bessel:	bessel(3M)
yn: Bessel functions.	bessel: j0, j1, jn, y0, y1,	bessel(3M)
	bfs: big file scanner.	bfs(1)
cpset: install object files in	binary directories.	cpset(1M)
fread, fwrite:	binary input/output.	fread(3S)
bsearch:	binary search a sorted table.	bsearch(3C)
tfind, tdelete, twalk: manage	binary search trees. tsearch,	tsearch(3C)
bind:	bind a name to a socket.	bind(2)
endpoint. t_bind:	bind an address to a transport	t_bind(3n)
	bind: bind a name to a socket.	bind(2)
nfsd,	biod: NFS daemons.	nfsd(1M)
bcopy, bcmp, bzero:	bit and byte string/	bstring(3)
	bj: the game of black jack.	bj(6)
bj: the game of	black jack.	bj(6)
bcopy: interactive	block copy.	bcopy(1M)
sum: print checksum and	block count of a file.	sum(1)
sync: update the super	block.	sync(1M)
sync: update super	block.	sync(2)
df: report number of free disk	blocks and i-nodes.	df(1M)
bcheck: print the list of	blocks associated with an.	bcheck(1M)
libdev: manipulate Volume Home	Blocks (VHB).	libdev(3X)
powerfail: system/	brc, bcheckrc, drvload,	brc(1M)
space allocation.	brk, sbrk: change data segment	brk(2)
modest-sized programs.	bs: a compiler/interpreter for	bs(1)
sorted table.	bsearch: binary search a	bsearch(3C)
stdio: standard	buffered input/output package.	stdio(3S)
setbuf, setvbuf: assign	buffering to a stream.	setbuf(3S)
mknod:	build special file.	mknod(1M)
vme: VME	bus interface.	vme(7)
between host and network	byte order. /convert values	byteorder(3)
bcopy, bcmp, bzero: bit and	byte string operations.	bstring(3)

size: print section sizes in	bytes of common object files.	size(1)
swab: swap	bytes.	swab(3C)
operations. bcopy, bcmp,	bzero: bit and byte string	bstring(3)
cc:	C compiler.	cc(1)
cflow: generate	C flowgraph.	cflow(1)
cpp: the	C language preprocessor.	cpp(1)
include/ includes: determine	C language preprocessor	includes(1)
cb:	C program beautifier.	cb(1)
lint: a	C program checker.	lint(1)
cxref: generate	C program cross-reference.	cxref(1)
ctrace:	C program debugger.	ctrace(1)
extract and share strings in	C programs. xstr:	xstr(1)
time. cprofile: setting up a	C shell environment at login	cprofile(4)
object file. list: produce	C source listing from a common	list(1)
	cal: print calendar.	cal(1)
dc: desk	calculator.	dc(1)
cal: print	calendar.	cal(1)
	calendar: reminder service.	calendar(1)
cu:	call another UNIX system.	cu(1C)
data returned by stat system	call. stat:	stat(5)
Dialers: ACU/modem	calling protocols.	Dialers(5)
malloc, free, realloc,	calloc: main memory allocator.	malloc(3C)
fast/ malloc, free, realloc,	calloc, mallopt, mallinfo:	malloc(3X)
intro: introduction to system	calls and error numbers.	intro(2)
common shared NFS system	calls. nfssys:	nfssys(2)
request. rumount:	cancel queued remote resource	rumount(1M)
to an LP line printer. lp,	cancel: send/cancel requests	lp(1)
termcap: terminal	capability data base.	termcap(4)
terminfo: terminal	capability data base.	terminfo(4)
description into a terminfo/	captoinfo: convert a termcap	captoinfo(1M)
asa: interpret ASA	carriage control characters.	asa(1)
text editor (variant of ex for	casual users). edit:	edit(1)
files.	cat: concatenate and print	cat(1)
advent: explore Colossal	Cave.	advent(6)
	cb: C program beautifier.	cb(1)
	cc: C compiler.	cc(1)
cc2sw, cc2fp: front-end to the	cc command. cc1sw,	cc1sw(1)
create a front-end to the	cc command. gencc:	gencc(1M)
to the cc command.	cc1sw, cc2sw, cc2fp: front-end	cc1sw(1)
command. cc1sw, cc2sw,	cc2fp: front-end to the cc	cc1sw(1)
cc command. cc1sw,	cc2sw, cc2fp: front-end to the	cc1sw(1)
	cd: change working directory.	cd(1)
commentary of an SCCS delta.	cdc: change the delta	cdc(1)
/ceil, fmod, fabs: floor,	ceiling, remainder, absolute/	floor(3M)
	cflow: generate C flowgraph.	cflow(1)
/localtime, gmtime, asctime,	cftime, ascftime, tzset:/	ctime(3C)
strings.	cftime: language specific	cftime(4)
delta: make a delta	(change) to an SCCS file.	delta(1)
priority of running process by	changing nice. renice: alter	renice(1)
pipe: create an interprocess	channel.	pipe(2)
terminal's local RS-232	channels. tp: controlling	tp(7)
stream. ungetc: push	character back into input	ungetc(3S)
conversion/ chrtbl: generate	character classification and	chrtbl(1M)
and neqn. eqnchar: special	character definitions for eqn	eqnchar(5)
_toupper, setchrclass:	character handling. /_tolower,	ctype(3C)
user. cuserid: get	character login name of the	cuserid(3S)
/getchar, fgetc, getw: get	character or word from a/	getc(3S)
/putchar, fputc, putw: put	character or word on a stream.	putc(3S)

ascii: map of ASCII	character set.	ascii(5)
fgrep: search a file for a	character string.	fgrep(1)
interpret ASA carriage control	characters. asa:	asa(1)
_tolower, toascii: translate	characters. /_toupper,	conv(3C)
tr: translate	characters.	tr(1)
lastlogin, monacct, nulladm,/	chargefee, ckpacct, dodisk,	acctsh(1M)
directory.	chdir: change working	chdir(2)
fsck, dfsck:	check and repair file systems.	fsck(1M)
schedule. ckbupscd:	check file system backup	ckbupscd(1M)
permissions file. uucheck:	check the uucp directories and	uucheck(1M)
constant-width text for/ cw,	checkcw: prepare	cw(1)
text for nroff or/ eqn, neqn,	checkeq: format mathematical	eqn(1)
lint: a C program	checker.	lint(1)
grpck: password/group file	checkers. pwck,	pwck(1M)
systems processed by fsck and/	checklist: list of file	checklist(4)
formatted with the MM/ mm,	checkmm: print/check documents	mm(1)
file. sum: print	checksum and block count of a	sum(1)
chown,	chgrp: change owner or group.	chown(1)
times: get process and	child process times.	times(2)
terminate. wait: wait for	child process to stop or	wait(2)
libraries tool.	chkshlib: compare shared	chkshlib(1)
	chmod: change mode.	chmod(1)
	chmod: change mode of file.	chmod(2)
of a file.	chown: change owner and group	chown(2)
group.	chown, chgrp: change owner or	chown(1)
	chroot: change root directory.	chroot(2)
for a command.	chroot: change root directory	chroot(1M)
classification and conversion/	chrtbl: generate character	chrtbl(1M)
backup schedule.	ckbupscd: check file system	ckbupscd(1M)
monacct, nulladm,/ chargefee,	ckpacct, dodisk, lastlogin,	acctsh(1M)
chrtbl: generate character	classification and conversion/	chrtbl(1M)
strclean: STREAMS error logger	cleanup program.	strclean(1M)
uucp spool directory	clean-up. uucleanup:	uucleanup(1M)
	clear: clear terminal screen.	clear(1)
clri:	clear i-node.	clri(1M)
clear:	clear terminal screen.	clear(1)
status/ ferror, feof,	clearerr, fileno: stream	ferror(3S)
the listener. nlsgetcall: get	client's data passed through	nlsgetcall(3n)
(command interpreter) with	C-like syntax. csh: a shell	csh(1)
synchronization of the system	clock. /the time to allow	adjtime(2)
alarm: set a process alarm	clock.	alarm(2)
cron:	clock daemon.	cron(1M)
	clock: report CPU time used.	clock(3C)
on a STREAMS driver.	clone: open any minor device	clone(7)
ldclose, ldaclose:	close a common object file.	ldclose(3X)
close:	close a file descriptor.	close(2)
t_close:	close a transport endpoint.	t_close(3n)
fclose, fflush:	close or flush a stream.	fclose(3S)
telldir, seekdir, rewinddir,	closedir: directory/ /readdir,	directory(3X)
	clri: clear i-node.	clri(1M)
	cmp: compare two files.	cmp(1)
dis: object	code disassembler.	dis(1)
line-feeds.	col: filter reverse	col(1)
advent: explore	Colossal Cave.	advent(6)
comb:	combine SCCS deltas.	comb(1)
common to two sorted files.	comm: select or reject lines	comm(1)
nice: run a	command at low priority.	nice(1)
cc2fp: front-end to the cc	command. cc1sw, cc2sw,	cc1sw(1)

change root directory for a	command. chroot:	chroot(1M)
examples. usage: retrieve a	command description and usage	usage(1)
env: set environment for	command execution.	env(1)
rcmd: remote shell	command execution.	rcmd(1)
uux: UNIX-to-UNIX system	command execution.	uux(1C)
/ASSIST menus and	command forms.	astgen(1)
create a front-end to the cc	command. gencc:	gencc(1M)
quits. nohup: run a	command immune to hangups and	nohup(1)
C-like syntax. csh: a shell	(command interpreter) with	csh(1)
getopt: parse	command options.	getopt(1)
getopts, getoptcvt: parse	command options.	getopts(1)
locate executable file for	command. path:	path(1)
/shell, the standard/restricted	command programming language.	sh(1)
returning a stream to a remote	command. /routines for	rcmd(3)
and system/ timex: time a	command; report process data	timex(1)
uuxqt: execute remote	command requests.	uuxqt(1M)
return stream to a remote	command. rexec:	rexec(3)
per-process/ acctcms:	command summary from	acctcms(1M)
system: issue a shell	command.	system(3S)
used by the /etc/tapeset	command.. /information	tapedrives(4)
test: condition evaluation	command.	test(1)
time: time a	command.	time(1)
locate: identify a CTIX system	command using keywords.	locate(1)
argument list(s) and execute	command. xargs: construct	xargs(1)
and miscellaneous accounting	commands. /of accounting	acct(1M)
intro: introduction to	commands and application/	intro(1)
assistance using CTIX system	commands. assist:	assist(1)
at, batch: execute	commands at a later time.	at(1)
access graphical and numerical	commands. graphics:	graphics(1G)
install: install	commands.	install(1M)
mkhosts: make node name	commands.	mkhosts(1M)
multi-user/ rc2, rc3: run	commands performed for	rc2(1M)
operating system. rc0: run	commands performed to stop the	rc0(1M)
network useful with graphical	commands. stat: statistical	stat(1G)
streamio: STREAMS ioctl	commands.	streamio(7)
manipulate the object file	comment section. mcs:	mcs(1)
cdc: change the delta	commentary of an SCCS delta.	cdc(1)
ar:	common archive file format.	ar(4)
editor output. a.out:	common assembler and link	a.out(4)
as:	common assembler.	as(1)
glossary: definitions of	common CTIX system terms and/	glossary(1)
convert archive files to	common formats. convert:	convert(1)
routines. ldfcn:	common object file access	ldfcn(4)
conv:	common object file converter.	conv(1)
cprs: compress a	common object file.	cprs(1)
ldopen, ldaopen: open a	common object file for/	ldopen(3X)
/line number entries of a	common object file function.	ldlread(3X)
ldclose, ldaclose: close a	common object file.	ldclose(3X)
read the file header of a	common object file. ldfhread:	ldfhread(3X)
entries of a section of a	common object file. /number	ldlseek(3X)
the optional file header of a	common object file. /seek to	ldohseek(3X)
/entries of a section of a	common object file.	ldrseek(3X)
/section header of a	common object file.	ldshread(3X)
an indexed/named section of a	common object file. /seek to	ldsseek(3X)
of a symbol table entry of a	common object file. /the index	ldtbindex(3X)
symbol table entry of a	common object file. /indexed	ldtbread(3X)
seek to the symbol table of a	common object file. ldtbseek:	ldtbseek(3X)
line number entries in a	common object file. linenum:	linenum(4)

C source listing from a	common object file. /produce	list(1)
nm: print name list of	common object file.	nm(1)
relocation information for a	common object file. reloc:	reloc(4)
scnhdr: section header for a	common object file.	scnhdr(4)
line number information from a	common object file. /and	strip(1)
/retrieve symbol name for	common object file symbol/	ldgetname(3X)
table format. syms:	common object file symbol	syms(4)
filehdr: file header for	common object files.	filehdr(4)
ld: link editor for	common object files.	ld(1)
section sizes in bytes of	common object files. /print	size(1)
calls. nfssys:	common shared NFS system	nfssys(2)
comm: select or reject lines	common to two sorted files.	comm(1)
ipcs: report inter-process	communication facilities/	ipcs(1)
/ftok: standard interprocess	communication package.	stdipc(3C)
talkd: remote user	communication server.	talkd(1M)
socket: create an endpoint for	communication.	socket(2)
/configuration file for uucp	communications lines.	Devices(5)
diff: differential file	comparator.	diff(1)
descriptions. infocmp:	compare or print out terminfo	infocmp(1M)
chkshlib:	compare shared libraries tool.	chkshlib(1)
cmp:	compare two files.	cmp(1)
SCCS file. sccsdiff:	compare two versions of an	sccsdiff(1)
diff3: 3-way differential file	comparison.	diff3(1)
dircmp: directory	comparison.	dircmp(1)
expression. regcmp, regex:	compile and execute regular	regcmp(3X)
regexp: regular expression	compile and match routines.	regexp(5)
regcmp: regular expression	compile.	regcmp(1)
term: format of	compiled term file..	term(4)
cc: C	compiler.	cc(1)
tic: terminfo	compiler.	tic(1M)
yacc: yet another	compiler-compiler.	yacc(1)
modest-sized programs. bs: a	compiler/interpreter for	bs(1)
erf, erfc: error function and	complementary error function.	erf(3M)
wait: await	completion of process.	wait(1)
cprs:	compress a common object file.	cprs(1)
pack, pcat, unpack:	compress and expand files.	pack(1)
table entry of a/ ldtbindex:	compute the index of a symbol	ldtbindex(3X)
cat:	concatenate and print files.	cat(1)
test:	condition evaluation command.	test(1)
system.	config: configure a CTIX	config(1M)
NFS file systems export	configuration file. exports:	exports(4)
(internet/ inetd.conf:	configuration file for inetd	inetd.conf(4)
communications/ Devices:	configuration file for uucp	Devices(5)
gateways: routed	configuration file.	gateways(4)
netcf: Network	Configuration File.	netcf(4)
resolv.conf: resolver	configuration file.	resolver(4)
STREAMS linker, load socket	configuration. /ldsocket:	slink(1)
rtab: Remote I/O Processor	configuration table.	rtab(4)
config:	configure a CTIX system.	config(1M)
enpstart:	configure Ethernet processor.	enpstart(1M)
parameters. ifconfig:	configure network interface	ifconfig(1M)
I/O Processor. riopcfg:	configure system for Remote	riopcfg(1M)
system. lpadmin:	configure the LP spooling	lpadmin(1M)
system. uconf:	configure the operating	uconf(1M)
t_rcvconnect: receive the	confirmation from a connect/	t_rcvconnect(3)
to use as the virtual system/	conlocate: locate a terminal	conlocate(1M)
fwtmp, wtmpfix: manipulate	connect accounting records.	fwtmp(1M)
on a socket.	connect: initiate a connection	connect(2)

t_accept: accept a	connect request.	t_accept(3n)
t_listen: listen for a	connect request.	t_listen(3n)
the confirmation from a	connect request. /receive	t_rcvconnect(3)
getpeername: get name of	connected peer.	getpeername(2)
an out-going terminal line	connection. dial: establish	dial(3C)
connect: initiate a	connection on a socket.	connect(2)
down part of a full-duplex	connection. shutdown: shut	shutdown(2)
or expedited data sent over a	connection. /receive data	t_rcv(3n)
data or expedited data over a	connection. t_snd: send	t_snd(3n)
t_connect: establish a	connection with another/	t_connect(3n)
listen: listen for	connections on a socket.	listen(2)
acctcon1, acctcon2:	connect-time accounting.	acctcon(1M)
to use as the virtual system	console. /locate a terminal	conlocate(1M)
the kernel debugger system	console port. /change	dbconsole(1M)
console:	console terminal.	console(7)
for implementation-speci fic	constants. /file header	limits(4)
math: math functions and	constants.	math(5)
file header for symbolic	constants. unistd:	unistd(4)
cw, checkcw: prepare	constant-width text for troff.	cw(1)
mkfs:	construct a file system.	mkfs(1M)
execute command. xargs:	construct argument list(s) and	xargs(1)
nroff/troff, tbl, and eqn	constructs. deroff: remove	deroff(1)
debugging on. Uutry: try to	contact a remote system with	Uutry(1M)
ls: list	contents of directory.	ls(1)
ttoc, vtoc: graphical table of	contents routines. toc: dtoc,	toc(1G)
csplit:	context split.	csplit(1)
address resolution display and	control. arp:	arp(1M)
asa: interpret ASA carriage	control characters.	asa(1)
ioctl:	control device.	ioctl(2)
scsi: scsi	control device.	scsi(7)
Serial Line Internet Protocol	control facility. /switched	slipd(1M)
fcntl: file	control.	fcntl(2)
floating point environment	control. /fpsetsticky: IEEE	fpgetround(3)
init, telinit: process	control initialization.	init(1M)
icmp: Internet	Control Message Protocol.	icmp(7)
msgctl: message	control operations.	msgctl(2)
semctl: semaphore	control operations.	semctl(2)
shmctl: shared memory	control operations.	shmctl(2)
fcntl: file	control options.	fcntl(5)
tcp: Internet Transmission	Control Protocol.	tcp(7)
uadmin: administrative	control.	uadmin(1M)
uadmin: administrative	control.	uadmin(2)
uucp status inquiry and job	control. uustat:	uustat(1C)
vc: version	control.	vc(1)
V/TAPE 3200 half-inch tape	controller. /for Interphase	ipt(7)
set drive parameters for tape	controllers. tapeset:	tapeset(1M)
interface. tty:	controlling terminal	tty(7)
RS-232 channels. tp:	controlling terminal's local	tp(7)
converter.	conv: common object file	conv(1)
_toupper, _tolower, toascii:/	conv: toupper, tolower,	conv(3C)
terminals. term:	conventional names for	term(5)
units:	conversion program.	units(1)
character classification and	conversion tables. /generate	chrtbl(1M)
into a terminfo/ captoinfo:	convert a termcap description	captoinfo(1M)
dd:	convert and copy a file.	dd(1M)
English. number:	convert Arabic numerals to	number(6)
common formats. convert:	convert archive files to	convert(1)
integers and/ l3tol, ltol3:	convert between 3-byte	l3tol(3C)

and base-64 ASCII/ a64l, l64a:	convert between long integer	a64l(3C)
to common formats.	convert: convert archive files	convert(1)
/cftime, ascftime, tzset:	convert date and time to/	ctime(3C)
to string. ecvt, fcvt, gcvt:	convert floating-point number	ecvt(3C)
scanf, fscanf, sscanf:	convert formatted input.	scanf(3S)
strtod, atof:	convert string to/	strtod(3C)
strtol, atol, atoi:	convert string to integer.	strtol(3C)
htonl, htons, ntohl, ntohs:	convert values between host/	byteorder(3)
conv: common object file	converter.	conv(1)
timod: Transport Interface	cooperating STREAMS module.	timod(7)
dd: convert and	copy a file.	dd(1M)
bcopy: interactive block	copy.	bcopy(1M)
cpio:	copy file archives in and out.	cpio(1)
access time. dcopy:	copy file systems for optimal	dcopy(1M)
cp, ln, mv:	copy, link, or move files.	cp(1)
volcopy: make literal	copy of file system.	volcopy(1M)
rcp: remote file	copy.	rcp(1)
uuname: UNIX-to-UNIX system	copy. uucp, uulog,	uucp(1C)
UNIX-to-UNIX system file	copy. uuto, uupick: public	uuto(1C)
core: format of	core image file.	core(4)
synchronization of/ adjtime:	correct the time to allow	adjtime(2)
atan2:/ trig: sin,	cos, tan, asin, acos, atan,	trig(3M)
functions. sinh,	cosh, tanh: hyperbolic	sinh(3M)
sum: print checksum and block	count of a file.	sum(1)
wc: word	count.	wc(1)
move files.	cp, ln, mv: copy, link, or	cp(1)
cpio: format of	cpio archive.	cpio(4)
and out.	cpio: copy file archives in	cpio(1)
preprocessor.	cpp: the C language	cpp(1)
environment at login time.	cprofile: setting up a C shell	cprofile(4)
file.	cprs: compress a common object	cprs(1)
binary directories.	cpset: install object files in	cpset(1M)
clock: report	CPU time used.	clock(3C)
craps: the game of	craps.	craps(6)
	crash: examine system images.	crash(1M)
rewrite an existing one.	creat: create a new file or	creat(2)
command. gencc:	create a front-end to the cc	gencc(1M)
file. tmpnam, tempnam:	create a name for a temporary	tmpnam(3S)
an existing one. creat:	create a new file or rewrite	creat(2)
fork:	create a new process.	fork(2)
mkshlib:	create a shared library.	mkshlib(1)
ctags:	create a tags file.	ctags(1)
tmpfile:	create a temporary file.	tmpfile(3S)
communication. socket:	create an endpoint for	socket(2)
channel. pipe:	create an interprocess	pipe(2)
files. admin:	create and administer SCCS	admin(1)
assorted device/ createdev:	create device nodes for	createdev(1M)
umask: set and get file	creation mask.	umask(2)
	cron: clock daemon.	cron(1M)
crontab: user	crontab file.	crontab(1)
cxref: generate C program	cross-reference.	cxref(1)
pg: file perusal filter for	CRTs.	pg(1)
	crypt: encode/decode.	crypt(1)
encryption functions.	crypt: password and file	crypt(3X)
generate hashing encryption.	crypt, setkey, encrypt:	crypt(3C)
interpreter) with C-like/	csh: a shell (command	csh(1)
	csplit: context split.	csplit(1)
terminal.	ct: spawn getty to a remote	ct(1C)

	ctags: create a tags file.	ctags(1)
for terminal.	ctermid: generate file name	ctermid(3S)
asctime, cftime, ascftime,/	ctime, localtime, gmtime,	ctime(3C)
	ctinstall: install software.	ctinstall(1)
adman: administer a	CTIX system.	adman(1)
config: configure a	CTIX system.	config(1M)
uname: get name of current	CTIX system.	uname(2)
/definitions of common	CTIX system terms and/	glossary(1)
	ctrace: C program debugger.	ctrace(1)
	cu: call another UNIX system.	cu(1C)
ttt,	cubic: tic-tac-toe.	ttt(6)
uname: get name of	current CTIX system.	uname(2)
endpoint. t_look: look at the	current event on a transport	t_look(3n)
get/set unique identifier of	current host. /sethostid:	gethostid(2)
sethostname: get/set name of	current host. gethostname,	gethostname(2)
set or print identifier of	current host system. hostid:	hostid(1)
uname: print name of	current CTIX system.	uname(1)
activity. sact: print	current SCCS file editing	sact(1)
t_getstate: get the	current state.	t_getstate(3)
the Internet host name of the	current system. /set or print	hostname(1)
slot in the utmp file of the	current user. /find the	ttyslot(3C)
getcwd: get path-name of	current working directory.	getcwd(3C)
scr_dump: format of	curses screen image file..	scr_dump(4)
handling and optimization/	curses: terminal screen	curses(3X)
spline: interpolate smooth	curve.	spline(1G)
name of the user.	cuserid: get character login	cuserid(3S)
each line of a file. cut:	cut out selected fields of	cut(1)
constant-width text for/	cw, checkcw: prepare	cw(1)
cross-reference.	cxref: generate C program	cxref(1)
cron: clock	daemon.	cron(1M)
rfudaemon: Remote File Sharing	daemon process.	rfudaemon(1M)
routed: network routing	daemon.	routed(1M)
strerr: STREAMS error logger	daemon.	strerr(1M)
nfsd, biod: NFS	daemons.	nfsd(1M)
runacct: run	daily accounting.	runacct(1M)
Protocol server. ftpd:	DARPA Internet File Transfer	ftpd(1M)
number mapper. portmap:	DARPA port to RPC program	portmap(1M)
telnetd:	DARPA TELNET protocol server.	telnetd(1M)
tftp: user interface to the	DARPA TFTP protocol.	tftp(1)
Protocol server. tftpd:	DARPA Trivial File Transfer	tftpd(1M)
/handle special functions of	DASI 300 and 300s terminals.	300(1)
special functions of the	DASI 450 terminal. /handle	450(1)
/time a command; report process	data and system activity.	timex(1)
file. newaliases: rebuild the	data base for the mail aliases	newaliases(1)
rpc: Sun rpc program number	data base.	rpc(4)
termcap: terminal capability	data base.	termcap(4)
terminfo: terminal capability	data base.	terminfo(4)
generate disk accounting	data by user ID. diskusg:	diskusg(1M)
t_rcvuderr: receive a unit	data error indication.	t_rcvuderr(3)
/sgetl: access long integer	data in a machine-independent/	sputl(3X)
plock: lock process, text, or	data in memory.	plock(2)
connection. t_snd: send	data or expedited data over a	t_snd(3n)
over a/ t_rcv: receive	data or expedited data sent	t_rcv(3n)
nlsgetcall: get client's	data passed through the/	nlsgetcall(3n)
prof: display profile	data.	prof(1)
call. stat:	data returned by stat system	stat(5)
I/O Processor for online	data. riopqry: query Remote	riopqry(1M)
brk, sbrk: change	data segment space allocation.	brk(2)

/receive data or expedited	data sent over a connection.	t_rcv(3n)
types: primitive system	data types.	types(5)
t_rcvudata: receive a	data unit.	t_rcvudata(3)
t_sndudata: send a	data unit.	t_sndudata(3)
changes to the Help Facility	database. helpadm: make	helpadm(1M)
join: relational	database operator.	join(1)
using the mkfs(1) proto file	database. /and verify software	qinstall(1)
delete, firstkey, nextkey:	database subroutines. /store,	dbm(3X)
/dbm_error, dbm_clearerr:	database subroutines.	ndbm(3X)
a terminal or query terminfo	database. tput: initialize	tput(1)
udp: Internet User	Datagram Protocol.	udp(7)
settimeofday: get/set	date and time. gettimeofday,	gettimeofday(2)
/ascftime, tzset: convert	date and time to string.	ctime(3C)
date: print and set the	date.	date(1)
	date: print and set the date.	date(1)
debugger system console port.	dbconsole: change the kernel	dbconsole(1M)
/dbm_nextkey, dbm_error,	dbm_clearerr: database/	ndbm(3X)
dbm_store,/ dbm_open,	dbm_close, dbm_fetch,	ndbm(3X)
/dbm_fetch, dbm_store,	dbm_delete, dbm_firstkey,/	ndbm(3X)
/dbm_firstkey, dbm_nextkey,	dbm_error, dbm_clearerr:/	ndbm(3X)
dbm_open, dbm_close,	dbm_fetch, dbm_store,/	ndbm(3X)
/dbm_store, dbm_delete,	dbm_firstkey, dbm_nextkey,/	ndbm(3X)
firstkey, nextkey: database/	dbminit, fetch, store, delete,	dbm(3X)
/dbm_delete, dbm_firstkey,	dbm_nextkey, dbm_error,/	ndbm(3X)
dbm_fetch, dbm_store,/	dbm_open, dbm_close,	ndbm(3X)
/dbm_close, dbm_fetch,	dbm_store, dbm_delete,/	ndbm(3X)
	dc: desk calculator.	dc(1)
optimal access time.	dcopy: copy file systems for	dcopy(1M)
	dd: convert and copy a file.	dd(1M)
adb: absolute	debugger.	adb(1)
ctrace: C program	debugger.	ctrace(1)
fsdb: file system	debugger.	fsdb(1M)
load symbols in kernel	debugger. mkdbsym:	mkdbsym(1M)
sdb: symbolic	debugger.	sdb(1)
dbconsole: change the kernel	debugger system console port.	dbconsole(1M)
contact a remote system with	debugging on. Uutry: try to	Uutry(1M)
timezone: set	default system time zone.	timezone(4)
sysdef: output system	definition.	sysdef(1M)
eqnchar: special character	definitions for eqn and neqn.	eqnchar(5)
system terms and/ glossary:	definitions of common CTIX	glossary(1)
dbminit, fetch, store,	delete, firstkey, nextkey:/	dbm(3X)
names. basename, dirname:	deliver portions of path	basename(1)
file. tail:	deliver the last part of a	tail(1)
delta commentary of an SCCS	delta. cdc: change the	cdc(1)
file. delta: make a	delta (change) to an SCCS	delta(1)
delta. cdc: change the	delta commentary of an SCCS	cdc(1)
rmdel: remove a	delta from an SCCS file.	rmdel(1)
to an SCCS file.	delta: make a delta (change)	delta(1)
comb: combine SCCS	deltas.	comb(1)
errdemon: error-logging	demon.	errdemon(1M)
terminate the error-logging	demon. errstop:	errstop(1M)
mesg: permit or	deny messages.	mesg(1)
tbl, and eqn constructs.	deroff: remove nroff/troff,	deroff(1)
usage: retrieve a command	description and usage/	usage(1)
description into a terminfo	description. /a termcap	captoinfo(1M)
queuedefs: at/batch/cron queue	description file.	queuedefs(4)
system: system	description file.	system(4)
captoinfo: convert a termcap	description into a terminfo/	captoinfo(1M)

compare or print out terminfo	descriptions. infocmp:	infocmp(1M)
close: close a file	descriptor.	close(2)
dup: duplicate an open file	descriptor.	dup(2)
dup2: duplicate an open file	descriptor.	dup2(3C)
getdtablesize: get	descriptor table size.	getdtablesize(2)
dc:	desk calculator.	dc(1)
slattach, sldetach: attach and	detach serial lines as network/	slattach(1M)
file. access:	determine accessibility of a	access(2)
preprocessor/ includes:	determine C language	includes(1)
identifier. fstyp:	determine file system	fstyp(1M)
file:	determine file type.	file(1)
drivers: loadable	device drivers.	drivers(7)
lines for finite width output	device. fold: fold long	fold(1)
master: master	device information table.	master(4)
ioctl: control	device.	ioctl(2)
devnm:	device name.	devnm(1M)
device/ createdev: create	device nodes for assorted	createdev(1M)
clone: open any minor	device on a STREAMS driver.	clone(7)
/tekset, td: graphical	device routines and filters.	gdev(1G)
scsi: scsi control	device.	scsi(7)
device nodes for assorted	device types. /create	createdev(1M)
for uucp communications/	Devices: configuration file	Devices(5)
scsimap: set mappings for SCSI	devices.	scsimap(1M)
	devnm: device name.	devnm(1M)
blocks and i-nodes.	df: report number of free disk	df(1M)
systems. fsck,	dfsck: check and repair file	fsck(1M)
terminal line connection.	dial: establish an out-going	dial(3C)
ratfor: rational FORTRAN	dialect.	ratfor(1)
protocols.	Dialers: ACU/modem calling	Dialers(5)
bdiff: big	diff.	bdiff(1)
comparison.	diff3: 3-way differential file	diff3(1)
sdiff: side-by-side	difference program.	sdiff(1)
diffmk: mark	differences between files.	diffmk(1)
diff:	differential file comparator.	diff(1)
diff3: 3-way	differential file comparison.	diff3(1)
	dir: format of directories.	dir(4)
	dircmp: directory comparison.	dircmp(1)
file. uucheck: check the uucp	directories and permissions	uucheck(1M)
install object files in binary	directories. cpset:	cpset(1M)
dir: format of	directories.	dir(4)
link and unlink files and	directories. link, unlink:	link(1M)
mkdir, mkdirs: make	directories.	mkdir(1)
rm, rmdir: remove files or	directories.	rm(1)
cd: change working	directory.	cd(1)
chdir: change working	directory.	chdir(2)
chroot: change root	directory.	chroot(2)
uucleanup: uucp spool	directory clean-up.	uucleanup(1M)
dircmp:	directory comparison.	dircmp(1)
file. getdents: read	directory entries and put in a	getdents(2)
file system independent	directory entry. dirent:	dirent(4)
unlink: remove	directory entry.	unlink(2)
chroot: change root	directory for a command.	chroot(1M)
/make a lost+found	directory for fsck.	mklostfnd(1M)
adv: advertise a	directory for remote access.	adv(1M)
path-name of current working	directory. getcwd: get	getcwd(3C)
ls: list contents of	directory.	ls(1)
mkdir: make a	directory.	mkdir(2)
mvdir: move a	directory.	mvdir(1M)

pwd: working	directory name.	pwd(1)
/seekdir, rewinddir, closedir:	directory operations.	directory(3X)
ordinary file. mknod: make a	directory, or a special or	mknod(2)
rmdir: remove a	directory.	rmdir(2)
independent directory entry.	dirent: file system	dirent(4)
path names. basename,	dirname: deliver portions of	basename(1)
	dis: object code disassembler.	dis(1)
t_unbind:	disable a transport endpoint.	t_unbind(3n)
printers. enable,	disable: enable/disable LP	enable(1)
acct: enable or	disable process accounting.	acct(2)
dis: object code	disassembler.	dis(1)
type, modes, speed, and line	discipline. /set terminal	getty(1M)
type, modes, speed, and line	discipline. /set terminal	uugetty(1M)
t_snddis: send user-initiated	disconnect request.	t_snddis(3n)
retrieve information from	disconnect. t_rcvdis:	t_rcvdis(3n)
fusage:	disk access profiler.	fusage(1M)
sadp:	disk access profiler.	sadp(1M)
ID. diskusg: generate	disk accounting data by user	diskusg(1M)
df: report number of free	disk blocks and i-nodes.	df(1M)
disk: general	disk driver.	disk(7)
update: provide	disk synchronization.	update(1M)
du: summarize	disk usage.	du(1M)
accounting data by user ID.	diskusg: generate disk	diskusg(1M)
arp: address resolution	display and control.	arp(1M)
vi: screen-oriented (visual)	display editor based on ex.	vi(1)
information. rmntstat:	display mounted resource	rmntstat(1M)
prof:	display profile data.	prof(1)
statistics. serstat:	display serial port error	serstat(1M)
local network. ruptime:	display status of nodes on	ruptime(1)
hypot: Euclidean	distance function.	hypot(3M)
/lcong48: generate uniformly	distributed pseudo-random/	drand48(3C)
Sharing domain and network/	dname: print Remote File	dname(1M)
routines. /res_send, res_init,	dn_comp, dn_expand: resolver	resolver(3)
/res_send, res_init, dn_comp,	dn_expand: resolver routines.	resolver(3)
MM/ mm, checkmm: print/check	documents formatted with the	mm(1)
macro package for formatting	documents. mm: the MM	mm(5)
slides. mmt, mvt: typeset	documents, view graphs, and	mmt(1)
nulladm,/ chargefee, ckpacct,	dodisk, lastlogin, monacct,	acctsh(1M)
whodo: who is	doing what.	whodo(1M)
/print Remote File Sharing	domain and network names.	dname(1M)
named: Internet	domain name server.	named(1M)
/atof: convert string to	double-precision number.	strtod(3C)
gtdl, ptdl: RS-232 terminal	download. tdl,	tdl(1)
nrand48, mrand48, jrand48,/	drand48, erand48, lrand48,	drand48(3C)
graph:	draw a graph.	graph(1G)
arithmetic: provide	drill in number facts.	arithmetic(6)
controllers. tapeset: set	drive parameters for tape	tapeset(1M)
used by the/ tapedrives: tape	drive specific information	tapedrives(4)
facilitate usage of a tape	drive. tsioctl:	tsioctl(1)
any minor device on a STREAMS	driver. clone: open	clone(7)
disk: general disk	driver.	disk(7)
lddrv: manage loadable	drivers.	lddrv(1M)
drivers.	drivers: loadable device	drivers(7)
initialization/ brc, bcheckrc,	drvload, powerfail: system	brc(1M)
table of contents/ toc:	dtoc, ttoc, vtoc: graphical	toc(1G)
	du: summarize disk usage.	du(1M)
and status information from	dump. /extract error records	errdead(1M)
hd: hexadecimal and ascii file	dump.	hd(1)

od: octal	dump.	od(1)
object file. dump:	dump selected parts of an	dump(1)
descriptor.	dup: duplicate an open file	dup(2)
descriptor.	dup2: duplicate an open file	dup2(3C)
descriptor. dup:	duplicate an open file	dup(2)
descriptor. dup2:	duplicate an open file	dup2(3C)
echo:	echo arguments.	echo(1)
network/ ping: send ICMP	ECHO_REQUEST packets to	ping(1M)
floating-point number to/	ecvt, fcvt, gcvt: convert	ecvt(3C)
	ed, red: text editor.	ed(1)
program. end, etext,	edata: last locations in	end(3C)
ex for casual users).	edit: text editor (variant of	edit(1)
sact: print current SCCS file	editing activity.	sact(1)
/(visual) display	editor based on ex.	vi(1)
ed, red: text	editor.	ed(1)
ex: text	editor.	ex(1)
files. ld: link	editor for common object	ld(1)
ged: graphical	editor.	ged(1G)
common assembler and link	editor output. a.out:	a.out(4)
sed: stream	editor.	sed(1)
casual users). edit: text	editor (variant of ex for	edit(1)
ldeeprom: load	EEPROM.	ldeeprom(1M)
/user, real group, and	effective group IDs.	getuid(2)
and/ /getegid: get real user,	effective user, real group,	getuid(2)
language.	efl: extended FORTRAN	efl(1)
split FORTRAN, ratfor, or	efl files. fsplit:	fsplit(1)
pattern using full regular/	egrep: search a file for a	egrep(1)
	en: Ethernet Processor.	en(7)
enable/disable LP printers.	enable, disable:	enable(1)
accounting. acct:	enable or disable process	acct(2)
real-time priorities	enabled/disabled. rtpenable:	rtpenable(1M)
enable, disable:	enable/disable LP printers.	enable(1)
crypt:	encode/decode.	crypt(1)
encrypt: generate hashing	encryption. crypt, setkey,	crypt(3C)
crypt: password and file	encryption functions.	crypt(3X)
makekey: generate	encryption key.	makekey(1)
locations in program.	end, etext, edata: last	end(3C)
/getgrgid, getgrnam, setgrent,	endgrent, fgetgrent: get group/	getgrent(3C)
/gethostent, sethostent,	endhostent: get network host/	gethostbyname(3)
/getnetbyname, setnetent,	endnetent: get network entry.	getnetent(3)
socket: create an	endpoint for communication.	socket(2)
bind an address to a transport	endpoint. t_bind:	t_bind(3n)
t_close: close a transport	endpoint.	t_close(3n)
current event on a transport	endpoint. t_look: look at the	t_look(3n)
t_open: establish a transport	endpoint.	t_open(3n)
manage options for a transport	endpoint. t_optmgmt:	t_optmgmt(3n)
t_unbind: disable a transport	endpoint.	t_unbind(3n)
/getprotobyname, setprotoent,	endprotoent: get protocol/	getprotoent(3)
/getpwuid, getpwnam, setpwent,	endpwent, fgetpwent: get/	getpwent(3C)
/getservbyname, setservent,	endservent: get service entry.	getservent(3)
getspent, getspnam, setspent,	endspent, fgetspent, lckpwdf,/	getspent(3X)
utmp/ /pututline, setutent,	endutent, utmpname: access	getut(3C)
convert Arabic numerals to	English. number:	number(6)
processor.	enpstart: configure Ethernet	enpstart(1M)
getdents: read directory	entries and put in a file.	getdents(2)
nlist: get	entries from name list.	nlist(3C)
file. linenum: line number	entries in a common object	linenum(4)
file/ /manipulate line number	entries of a common object	ldlread(3X)

/ldnlseek: seek to line number	entries of a section of a/	ldlseek(3X)
/ldnrseek: seek to relocation	entries of a section of a/	ldrseek(3X)
system independent directory	entry. dirent: file	dirent(4)
utmp, wtmp: utmp and wtmp	entry formats.	utmp(4)
fgetgrent: get group file	entry. /setgrent, endgrent,	getgrent(3C)
endhostent: get network host	entry. /sethostent,	gethostbyname(3)
endnetent: get network	entry. /setnetent,	getnetent(3)
endprotoent: get protocol	entry. /setprotoent,	getprotoent(3)
fgetpwent: get password file	entry. /setpwent, endpwent,	getpwent(3C)
getrpcbynumber: get rpc	entry. /getrpcbyname,	getrpcent(3)
endservent: get service	entry. /setservent,	getservent(3)
utmpname: access utmp file	entry. /setutent, endutent,	getut(3C)
object file symbol table	entry. /symbol name for common	ldgetname(3X)
/the index of a symbol table	entry of a common object file.	ldtbindex(3X)
/read an indexed symbol table	entry of a common object file.	ldtbread(3X)
putpwent: write password file	entry.	putpwent(3C)
write shadow password file	entry. putspent:	putspent(3X)
unlink: remove directory	entry.	unlink(2)
command execution.	env: set environment for	env(1)
	environ: user environment.	environ(5)
cprofile: setting up a C shell	environment at login time.	cprofile(4)
profile: setting up an	environment at login time.	profile(4)
/IEEE floating point	environment control.	fpgetround(3)
environ: user	environment.	environ(5)
execution. env: set	environment for command	env(1)
getenv: return value for	environment name.	getenv(3C)
putenv: change or add value to	environment.	putenv(3C)
performed for multi-user	environment. /run commands	rc2(1M)
stop the Remote File Sharing	environment. rfstop:	rfstop(1M)
interface, and terminal	environment. /terminal	tset(1)
character definitions for	eqn and neqn. /special	eqnchar(5)
remove nroff/troff, tbl, and	eqn constructs. deroff:	deroff(1)
mathematical text for nroff/	eqn, neqn, checkeq: format	eqn(1)
definitions for eqn and neqn.	eqnchar: special character	eqnchar(5)
rhosts: remote	equivalent users.	rhosts(4)
mrand48, jrand48,/ drand48,	erand48, lrand48, nrand48,	drand48(3C)
graphical device/ gdev: hpd,	erase, hardcopy, tekset, td:	gdev(1G)
complementary error function.	erf, erfc: error function and	erf(3M)
	err: error-logging interface.	err(7)
and status information from/	errdead: extract error records	errdead(1M)
	errdemon: error-logging demon.	errdemon(1M)
format.	errfile: error-log file	errfile(4)
system error/ perror,	errno, sys_errlist, sys_nerr:	perror(3C)
function and complementary	error function. /erfc: error	erf(3M)
receive a unit data	error indication. t_rcvuderr:	t_rcvuderr(3)
strclean: STREAMS	error logger cleanup program.	strclean(1M)
strerr: STREAMS	error logger daemon.	strerr(1M)
log: interface to STREAMS	error logging and event/	log(7)
t_error: produce	error message.	t_error(3n)
sys_errlist, sys_nerr: system	error messages. /errno,	perror(3C)
to system calls and	error numbers. /introduction	intro(2)
information/ errdead: extract	error records and status	errdead(1M)
serstat: display serial port	error statistics.	serstat(1M)
matherr:	error-handling function.	matherr(3M)
errfile:	error-log file format.	errfile(4)
errdemon:	error-logging demon.	errdemon(1M)
errstop: terminate the	error-logging demon.	errstop(1M)
err:	error-logging interface.	err(7)

process a report of logged	errors. errpt:	errpt(1M)
hashcheck: find spelling	errors. /hashmake, spellin,	spell(1)
error-logging demon.	errstop: terminate the	errstop(1M)
another transport/ t_connect:	establish a connection with	t_connect(3n)
endpoint. t_open:	establish a transport	t_open(3n)
terminal line/ dial:	establish an out-going	dial(3C)
setmnt:	establish mount table.	setmnt(1M)
with information from	/etc/passwd. //etc/shadow	pwconv(1M)
with information from	/etc/passwd. //etc/shadow	pwunconv(1M)
pwconv: install and update	/etc/shadow with information/	pwconv(1M)
pwunconv: install and update	/etc/shadow with information/	pwunconv(1M)
/information used by the	/etc/tapeset command..	tapedrives(4)
in program. end,	etext, edata: last locations	end(3C)
en:	Ethernet Processor.	en(7)
enpstart: configure	Ethernet processor.	enpstart(1M)
hypot:	Euclidean distance function.	hypot(3M)
expression. expr:	evaluate arguments as an	expr(1)
test: condition	evaluation command.	test(1)
t_look: look at the current	event on a transport endpoint.	t_look(3n)
to STREAMS error logging and	event tracing. log: interface	log(7)
notify, unnotify, evwait,	evnowait: manage/	notify(2)
notify, unnotify,	evwait, evnowait: manage/	notify(2)
edit: text editor (variant of	ex for casual users).	edit(1)
	ex: text editor.	ex(1)
display editor based on	ex. /screen-oriented (visual)	vi(1)
crash:	examine system images.	crash(1M)
a file. locking:	exclusive access to regions of	locking(2)
execve, execlp, execvp:/	exec: execl, execv, execle,	exec(2)
execlp, execvp: execute/ exec:	execl, execv, execle, execve,	exec(2)
execvp:/ exec: execl, execv,	execle, execve, execlp,	exec(2)
/execl, execv, execle, execve,	execlp, execvp: execute a/	exec(2)
path: locate	executable file for command.	path(1)
execve, execlp, execvp:	execute a file. /execle,	exec(2)
construct argument list(s) and	execute command. xargs:	xargs(1)
time. at, batch:	execute commands at a later	at(1)
regcmp, regex: compile and	execute regular expression.	regcmp(3X)
requests. uuxqt:	execute remote command	uuxqt(1M)
set environment for command	execution. env:	env(1)
sleep: suspend	execution for an interval.	sleep(1)
sleep: suspend	execution for interval.	sleep(3C)
monitor: prepare	execution profile.	monitor(3C)
rcmd: remote shell command	execution.	rcmd(1)
rexecd: remote	execution server.	rexecd(1M)
profil:	execution time profile.	profil(2)
UNIX-to-UNIX system command	execution. uux:	uux(1C)
execvp: execute/ exec: execl,	execv, execle, execve, execlp,	exec(2)
exec: execl, execv, execle,	execve, execlp, execvp:/	exec(2)
/execv, execle, execve, execlp,	execvp: execute a file.	exec(2)
a new file or rewrite an	existing one. creat: create	creat(2)
exit,	_exit: terminate process.	exit(2)
exponential, logarithm,/	exp, log, log10, pow, sqrt:	exp(3M)
pcat, unpack: compress and	expand files. pack,	pack(1)
to spaces, and vice versa.	expand, unexpand: expand tabs	expand(1)
t_snd: send data or	expedited data over a/	t_snd(3n)
t_rcv: receive data or	expedited data sent over a/	t_rcv(3n)
advent:	explore Colossal Cave.	advent(6)
exp, log, log10, pow, sqrt:	exponential, logarithm, power,/	exp(3M)
exports: NFS file systems	export configuration file.	exports(4)

export configuration file.	exports: NFS file systems	exports(4)
expression.	expr: evaluate arguments as an	expr(1)
routines. regexp: regular	expression compile and match	regexp(5)
regcmp: regular	expression compile.	regcmp(1)
expr: evaluate arguments as an	expression.	expr(1)
compile and execute regular	expression. regcmp, regex:	regcmp(3X)
a pattern using full regular	expressions. /a file for	egrep(1)
efl:	extended FORTRAN language.	efl(1)
extproc: turn	external processing on or off.	extproc(1M)
programs. xstr:	extract and share strings in C	xstr(1)
status information/ errdead:	extract error records and	errdead(1M)
in a file. strings:	extract the ASCII text strings	strings(1)
remainder,/ floor, ceil, fmod,	fabs: floor, ceiling,	floor(3M)
drive. tsioctl:	facilitate usage of a tape	tsioctl(1)
factors of a number.	factor: obtain the prime	factor(1)
factor: obtain the prime	factors of a number.	factor(1)
/usr/adm/loginlog: log of	failed login attempts.	loginlog(4)
true,	false: provide truth values.	true(1)
data in a machine-independent	fashion. /access long integer	sputl(3X)
finc:	fast incremental backup.	finc(1M)
/calloc, mallopt, mallinfo:	fast main memory allocator.	malloc(3X)
a stream.	fclose, fflush: close or flush	fclose(3S)
	fcntl: file control.	fcntl(2)
	fcntl: file control options.	fcntl(5)
floating-point number/ ecvt,	fcvt, gcvt: convert	ecvt(3C)
fopen, freopen,	fdopen: open a stream.	fopen(3S)
status inquiries. ferror,	feof, clearerr, fileno: stream	ferror(3S)
fileno: stream status/	ferror, feof, clearerr,	ferror(3S)
firstkey, nextkey:/ dbminit,	fetch, store, delete,	dbm(3X)
for a file system.	ff: file names and statistics	ff(1M)
stream. fclose,	fflush: close or flush a	fclose(3S)
word from a/ getc, getchar,	fgetc, getw: get character or	getc(3S)
/getgrnam, setgrent, endgrent,	fgetgrent: get group file/	getgrent(3C)
/getpwnam, setpwent, endpwent,	fgetpwent: get password file/	getpwent(3C)
stream. gets,	fgets: get a string from a	gets(3S)
/getspnam, setspent, endspent,	fgetspent, lckpwdf, ulckpwdf:/	getspent(3X)
character string.	fgrep: search a file for a	fgrep(1)
times. utime: set	file access and modification	utime(2)
ldfcn: common object	file access routines.	ldfcn(4)
determine accessibility of a	file. access:	access(2)
/2645A terminal tape	file archiver.	hpio(1)
tar: tape	file archiver.	tar(1)
cpio: copy	file archives in and out.	cpio(1)
pwck, grpck: password/group	file checkers.	pwck(1M)
chmod: change mode of	file.	chmod(2)
change owner and group of a	file. chown:	chown(2)
mcs: manipulate the object	file comment section.	mcs(1)
diff: differential	file comparator.	diff(1)
diff3: 3-way differential	file comparison.	diff3(1)
fcntl:	file control.	fcntl(2)
fcntl:	file control options.	fcntl(5)
conv: common object	file converter.	conv(1)
rcp: remote	file copy.	rcp(1)
public UNIX-to-UNIX system	file copy. uuto, uupick:	uuto(1C)
core: format of core image	file.	core(4)
cprs: compress a common object	file.	cprs(1)
umask: set and get	file creation mask.	umask(2)
crontab: user crontab	file.	crontab(1)

ctags: create a tags	file.	ctags(1)
fields of each line of a	file. cut: cut out selected	cut(1)
using the mkfs(1) proto	file database. /software	qinstall(1)
dd: convert and copy a	file.	dd(1M)
a delta (change) to an SCCS	file. delta: make	delta(1)
close: close a	file descriptor.	close(2)
dup: duplicate an open	file descriptor.	dup(2)
dup2: duplicate an open	file descriptor.	dup2(3C)
	file: determine file type.	file(1)
hd: hexadecimal and ascii	file dump.	hd(1)
selected parts of an object	file. dump: dump	dump(1)
sact: print current SCCS	file editing activity.	sact(1)
crypt: password and	file encryption functions.	crypt(3X)
endgrent, fgetgrent: get group	file entry. /setgrent,	getgrent(3C)
fgetpwent: get password	file entry. /endpwent,	getpwent(3C)
utmpname: access utmp	file entry. /endutent,	getut(3C)
putpwent: write password	file entry.	putpwent(3C)
write shadow password	file entry. putspent:	putspent(3X)
execlp, execvp: execute a	file. /execv, execle, execve,	exec(2)
systems export configuration	file. exports: NFS file	exports(4)
fgrep: search a	file for a character string.	fgrep(1)
grep: search a	file for a pattern.	grep(1)
regular/ egrep: search a	file for a pattern using full	egrep(1)
path: locate executable	file for command.	path(1)
inetd.conf: configuration	file for inetd (internet/	inetd.conf(4)
ldaopen: open a common object	file for reading. ldopen,	ldopen(3X)
netrc: login	file for remote networks.	netrc(4)
aliases: aliases	file for sendmail.	aliases(4)
lines. Devices: configuration	file for uucp communications	Devices(5)
acct: per-process accounting	file format.	acct(4)
ar: common archive	file format.	ar(4)
errfile: error-log	file format.	errfile(4)
intro: introduction to	file formats.	intro(4)
entries of a common object	file function. /line number	ldlread(3X)
gateways: routed configuration	file.	gateways(4)
get: get a version of an SCCS	file.	get(1)
directory entries and put in a	file. getdents: read	getdents(2)
group: group	file.	group(4)
files. filehdr:	file header for common object	filehdr(4)
limits:	file header for/	limits(4)
constants. unistd:	file header for symbolic	unistd(4)
file. ldfhread: read the	file header of a common object	ldfhread(3X)
ldohseek: seek to the optional	file header of a common object/	ldohseek(3X)
split: split a	file into pieces.	split(1)
issue: issue identification	file.	issue(4)
of a member of an archive	file. /read the archive header	ldahread(3X)
close a common object	file. ldclose, ldaclose:	ldclose(3X)
file header of a common object	file. ldfhread: read the	ldfhread(3X)
a section of a common object	file. /line number entries of	ldlseek(3X)
file header of a common object	file. /seek to the optional	ldohseek(3X)
a section of a common object	file. /relocation entries of	ldrseek(3X)
header of a common object	file. /indexed/named section	ldshread(3X)
section of a common object	file. /to an indexed/named	ldsseek(3X)
table entry of a common object	file. /the index of a symbol	ldtbindex(3X)
table entry of a common object	file. /read an indexed symbol	ldtbread(3X)
table of a common object	file. /seek to the symbol	ldtbseek(3X)
entries in a common object	file. linenum: line number	linenum(4)
link: link to a	file.	link(2)

listing from a common object	file. list: produce C source	list(1)
set links/ qlist: print out	file lists from proto file;	qlist(1)
access to regions of a	file. locking: exclusive	locking(2)
masterupd: update the master	file.	masterupd(1M)
make an ifile from an object	file. mkifile:	mkifile(1M)
mknod: build special	file.	mknod(1M)
or a special or ordinary	file. /make a directory,	mknod(2)
ctermid: generate	file name for terminal.	ctermid(3S)
mktemp: make a unique	file name.	mktemp(3C)
for a file system	file names and statistics	ff(1M)
netcf: Network Configuration	File.	netcf(4)
data base for the mail aliases	file. newaliases: rebuild the	newaliases(1)
change the format of a text	file. newform:	newform(1)
name list of common object	file. nm: print	nm(1)
null: the null	file.	null(7)
/find the slot in the utmp	file of the current user.	ttyslot(3C)
/identify processes using a	file or file structure.	fuser(1M)
one. creat: create a new	file or rewrite an existing	creat(2)
passwd: password	file.	passwd(4)
or subsequent lines of one	file. /lines of several files	paste(1)
pg:	file perusal filter for CRTs.	pg(1)
/rewind, ftell: reposition a	file pointer in a stream.	fseek(3S)
lseek: move read/write	file pointer.	lseek(2)
prs: print an SCCS	file.	prs(1)
queue description	file. /at/batch/cron	queuedefs(4)
read: read from	file.	read(2)
for a common object	file. /relocation information	reloc(4)
resolver configuration	file. resolv.conf:	resolver(4)
Sharing name server master	file. rfmaster: Remote File	rfmaster(4)
remove a delta from an SCCS	file. rmdel:	rmdel(1)
bfs: big	file scanner.	bfs(1)
two versions of an SCCS	file. sccsdiff: compare	sccsdiff(1)
sccsfile: format of SCCS	file.	sccsfile(4)
header for a common object	file. scnhdr: section	scnhdr(4)
format of curses screen image	file.. scr_dump:	scr_dump(4)
/out file lists from proto	file; set links based on.	qlist(1)
shadow: password	file.	shadow(4)
rfadmin: Remote	File Sharing administration.	rfadmin(1M)
rfudaemon: Remote	File Sharing daemon process.	rfudaemon(1M)
network/ dname: print Remote	File Sharing domain and	dname(1M)
rfstop: stop the Remote	File Sharing environment.	rfstop(1M)
rfpasswd: change Remote	File Sharing host password.	rfpasswd(1M)
master file. rfmaster: Remote	File Sharing name server	rfmaster(4)
query. nsquery: Remote	File Sharing name server	nsquery(1M)
shell/ rfuadmin: Remote	File Sharing notification	rfuadmin(1M)
unadv: unadvertise a Remote	File Sharing resource.	unadv(1M)
/mount, unmount Remote	File Sharing (RFS) resources.	rmountall(1M)
rfstart: start Remote	File Sharing.	rfstart(1M)
mapping. idload: Remote	File Sharing user and group	idload(1M)
fsize: report	file size.	fsize(1)
stat, fstat: get	file status.	stat(2)
the ASCII text strings in a	file. strings: extract	strings(1)
from a common object	file. /line number information	strip(1)
processes using a file or	file structure. /identify	fuser(1M)
checksum and block count of a	file. sum: print	sum(1)
swrite: synchronous write on a	file.	swrite(2)
/symbol name for common object	file symbol table entry.	ldgetname(3X)
syms: common object	file symbol table format.	syms(4)

ckbupscd: check	file system backup schedule.	ckbupscd(1M)
fsdb:	file system debugger.	fsdb(1M)
volume. fs:	file system: format of system	fs(4)
fstyp: determine	file system identifier.	fstyp(1M)
directory entry. dirent:	file system independent	dirent(4)
statfs, fstatfs: get	file system information.	statfs(2)
mkfs: construct a	file system.	mkfs(1M)
mount: mount a	file system.	mount(2)
/mount, unmount Network	File System resources.	nmountall(1M)
nfsstat: Network	File System statistics.	nfsstat(1M)
ustat: get	file system statistics.	ustat(2)
fsstat: report	file system status.	fsstat(1M)
mnttab: mounted	file system table.	mnttab(4)
rmtab: remotely mounted	file system table.	rmtab(4)
sysfs: get	file system type information.	sysfs(2)
umount: unmount a	file system.	umount(2)
volcopy: make literal copy of	file system.	volcopy(1M)
system: system description	file.	system(4)
/umount: mount and unmount	file systems and remote/	mount(1M)
configuration/ exports: NFS	file systems export	exports(4)
access time. dcopy: copy	file systems for optimal	dcopy(1M)
fsck, dfsck: check and repair	file systems.	fsck(1M)
labelit: provide labels for	file systems.	labelit(1M)
mount, unmount multiple	file systems. /umountall:	mountall(1M)
and/ checklist: list of	file systems processed by fsck	checklist(4)
deliver the last part of a	file. tail:	tail(1)
term: format of compiled term	file..	term(4)
tmpfile: create a temporary	file.	tmpfile(3S)
create a name for a temporary	file. tmpnam, tempnam:	tmpnam(3S)
and modification times of a	file. touch: update access	touch(1)
ftp: ARPANET	file transfer program.	ftp(1)
ftpd: DARPA Internet	File Transfer Protocol server.	ftpd(1M)
tftpd: DARPA Trivial	File Transfer Protocol server.	tftpd(1M)
uucp system. uucico:	file transport program for the	uucico(1M)
ftw: walk a	file tree.	ftw(3C)
file: determine	file type.	file(1)
undo a previous get of an SCCS	file. unget:	unget(1)
report repeated lines in a	file. uniq:	uniq(1)
directories and permissions	file. uucheck: check the uucp	uucheck(1M)
val: validate SCCS	file.	val(1)
write: write on a	file.	write(2)
umask: set	file-creation mode mask.	umask(1)
common object files.	filehdr: file header for	filehdr(4)
ferror, feof, clearerr,	fileno: stream status/	ferror(3S)
and print process accounting	file(s). acctcom: search	acctcom(1)
merge or add total accounting	files. acctmerg:	acctmerg(1M)
create and administer SCCS	files. admin:	admin(1)
link, unlink: link and unlink	files and directories.	link(1M)
cat: concatenate and print	files.	cat(1)
cmp: compare two	files.	cmp(1)
lines common to two sorted	files. comm: select or reject	comm(1)
ln, mv: copy, link, or move	files. cp,	cp(1)
mark differences between	files. diffmk:	diffmk(1)
file header for common object	files. filehdr:	filehdr(4)
find: find	files.	find(1)
frec: recover	files from a backup tape.	frec(1M)
format specification in text	files. fspec:	fspec(4)
FORTRAN, ratfor, or efl	files. fsplit: split	fsplit(1)

string, format of graphical	files. /graphical primitive	gps(4)
cpset: install object	files in binary directories.	cpset(1M)
language preprocessor include	files. includes: determine C	includes(1)
intro: introduction to special	files.	intro(7)
link editor for common object	files. ld:	ld(1)
lockf: record locking on	files.	lockf(3C)
passmgmt: password	files management.	passmgmt(1M)
rm, rmdir: remove	files or directories.	rm(1)
/merge same lines of several	files or subsequent lines of/	paste(1)
unpack: compress and expand	files. pack, pcat,	pack(1)
pr: print	files.	pr(1)
in bytes of common object	files. /print section sizes	size(1)
sort: sort and/or merge	files.	sort(1)
convert: convert archive	files to common formats.	convert(1)
what: identify SCCS	files.	what(1)
fstab:	file-system-table.	fstab(4)
pg: file perusal	filter for CRTs.	pg(1)
greek: select terminal	filter.	greek(1)
nl: line numbering	filter.	nl(1)
col:	filter reverse line-feeds.	col(1)
tio: tape io	filter.	tio(1)
graphical device routines and	filters. /tekset, td:	gdev(1G)
tplot: graphics	filters.	tplot(1G)
	finc: fast incremental backup.	finc(1M)
find:	find files.	find(1)
hyphen:	find hyphenated words.	hyphen(1)
ttyname, isatty:	find name of a terminal.	ttyname(3C)
object library. lorder:	find ordering relation for an	lorder(1)
hashmake, spellin, hashcheck:	find spelling errors. spell,	spell(1)
of the current user. ttyslot:	find the slot in the utmp file	ttyslot(3C)
lookup program.	finger: user information	finger(1)
information server.	fingerd: remote user	fingerd(1M)
fold: fold long lines for	finite width output device.	fold(1)
dbminit, fetch, store, delete,	firstkey, nextkey: database/	dbm(3X)
fish: play "Go	Fish".	fish(6)
tee: pipe	fitting.	tee(1)
/fpgetsticky, fpsetsticky: IEEE	floating point environment/	fpgetround(3)
isnand, isnanf: test for	floating point NaN/ isnan:	isnan(3C)
ecvt, fcvt, gcvt: convert	floating-point number to/	ecvt(3C)
/modf: manipulate parts of	floating-point numbers.	frexp(3C)
floor, ceil, fmod, fabs:	floor, ceiling, remainder,/	floor(3M)
cflow: generate C	flowgraph.	cflow(1)
fclose, fflush: close or	flush a stream.	fclose(3S)
remainder,/ floor, ceil,	fmod, fabs: floor, ceiling,	floor(3M)
width output device. fold:	fold long lines for finite	fold(1)
stream.	fopen, freopen, fdopen: open a	fopen(3S)
advertised resource. fumount:	forced unmount of an	fumount(1M)
	fork: create a new process.	fork(2)
per-process accounting file	format. acct:	acct(4)
service request/ nlsrequest:	format and send listener	nlsrequest(3n)
ar: common archive file	format.	ar(4)
errfile: error-log file	format.	errfile(4)
nroff or/ eqn, neqn, checkeq:	format mathematical text for	eqn(1)
newform: change the	format of a text file.	newform(1)
inode:	format of an i-node.	inode(4)
term:	format of compiled term file..	term(4)
core:	format of core image file.	core(4)
cpio:	format of cpio archive.	cpio(4)

file.. scr_dump:	format of curses screen image	scr_dump(4)
dir:	format of directories.	dir(4)
/graphical primitive string,	format of graphical files.	gps(4)
sccsfile:	format of SCCS file.	sccsfile(4)
fs: file system:	format of system volume.	fs(4)
files. fspec:	format specification in text	fspec(4)
object file symbol table	format. syms: common	syms(4)
troff. tbl:	format tables for nroff or	tbl(1)
nroff:	format text.	nroff(1)
archive files to common	formats. convert: convert	convert(1)
intro: introduction to file	formats.	intro(4)
wtmp: utmp and wtmp entry	formats. utmp,	utmp(4)
scanf, fscanf, sscanf: convert	formatted input.	scanf(3S)
/vfprintf, vsprintf: print	formatted output of a varargs/	vprintf(3S)
fprintf, sprintf: print	formatted output. printf,	printf(3S)
/checkmm: print/check documents	formatted with the MM macros.	mm(1)
mptx: the macro package for	formatting a permuted index.	mptx(5)
mm: the MM macro package for	formatting documents.	mm(5)
ms: text	formatting macros.	ms(5)
man: macros for	formatting manual pages.	man(5)
me: macros for	formatting papers.	me(5)
ASSIST menus and command	forms. /generate/modify	astgen(1)
ratfor: rational	FORTRAN dialect.	ratfor(1)
efl: extended	FORTRAN language.	efl(1)
files. fsplit: split	FORTRAN, ratfor, or efl	fsplit(1)
hopefully interesting, adage.	fortune: print a random,	fortune(6)
fpgetround, fpsetround,	fpgetmask, fpsetmask,/	fpgetround(3)
fpgetmask, fpsetmask,/	fpgetround, fpsetround,	fpgetround(3)
/fpgetmask, fpsetmask,	fpgetsticky, fpsetsticky: IEEE/	fpgetround(3)
formatted output. printf,	fprintf, sprintf: print	printf(3S)
/fpsetround, fpgetmask,	fpsetmask, fpgetsticky,/	fpgetround(3)
fpsetmask,/ fpgetround,	fpsetround, fpgetmask,	fpgetround(3)
point/ /fpsetmask, fpgetsticky,	fpsetsticky: IEEE floating	fpgetround(3)
word on a/ putc, putchar,	fputc, putw: put character or	putc(3S)
stream. puts,	fputs: put a string on a	puts(3S)
input/output.	fread, fwrite: binary	fread(3S)
backup tape.	frec: recover files from a	frec(1M)
t_free:	free a library structure.	t_free(3n)
df: report number of	free disk blocks and i-nodes.	df(1M)
memory allocator. malloc,	free, realloc, calloc: main	malloc(3C)
mallopt, mallinfo:/ malloc,	free, realloc, calloc,	malloc(3X)
stream. fopen,	freopen, fdopen: open a	fopen(3S)
parts of floating-point/	frexp, ldexp, modf: manipulate	frexp(3C)
frec: recover files	from a backup tape.	frec(1M)
list: produce C source listing	from a common object file.	list(1)
/and line number information	from a common object file.	strip(1)
/receive the confirmation	from a connect request.	t_rcvconnect(3)
recvfrom: receive a message	from a socket. recv,	recv(2)
getw: get character or word	from a stream. /fgetc,	getc(3S)
gets, fgets: get a string	from a stream.	gets(3S)
mkifile: make an ifile	from an object file.	mkifile(1M)
rmdel: remove a delta	from an SCCS file.	rmdel(1)
getopt: get option letter	from argument vector.	getopt(3C)
t_rcvdis: retrieve information	from disconnect.	t_rcvdis(3n)
records and status information	from dump. /extract error	errdead(1M)
/etc/shadow with information	from /etc/passwd. /and update	pwconv(1M)
/etc/shadow with information	from /etc/passwd. /and update	pwunconv(1M)
read: read	from file.	read(2)

ncheck: generate path names	from i-numbers.	ncheck(1M)
nlist: get entries	from name list.	nlist(3C)
acctcms: command summary	from per-process accounting/	acctcms(1M)
qlist: print out file lists	from proto file; set links/	qlist(1)
getpw: get name	from UID.	getpw(3C)
cc1sw, cc2sw, cc2fp:	front-end to the cc command.	cc1sw(1)
gencc: create a	front-end to the cc command.	gencc(1M)
system volume.	fs: file system: format of	fs(4)
formatted input. scanf,	fscanf, sscanf: convert	scanf(3S)
of file systems processed by	fsck and ncheck. /list	checklist(4)
file systems.	fsck, dfsck: check and repair	fsck(1M)
a lost+found directory for	fsck. mklost+found: make	mklostfnd(1M)
	fsdb: file system debugger.	fsdb(1M)
reposition a file pointer in/	fseek, rewind, ftell:	fseek(3S)
	fsize: report file size.	fsize(1)
text files.	fspec: format specification in	fspec(4)
or efl files.	fsplit: split FORTRAN, ratfor,	fsplit(1)
status.	fsstat: report file system	fsstat(1M)
	fstab: file-system-table.	fstab(4)
stat,	fstat: get file status.	stat(2)
information. statfs,	fstatfs: get file system	statfs(2)
identifier.	fstyp: determine file system	fstyp(1M)
pointer in a/ fseek, rewind,	ftell: reposition a file	fseek(3S)
communication/ stdipc,	ftok: standard interprocess	stdipc(3C)
program.	ftp: ARPANET file transfer	ftp(1)
Transfer Protocol server.	ftpd: DARPA Internet File	ftpd(1M)
	ftw: walk a file tree.	ftw(3C)
/a file for a pattern using	full regular expressions.	egrep(1)
shutdown: shut down part of a	full-duplex connection.	shutdown(2)
advertised resource.	fumount: forced unmount of an	fumount(1M)
error/ erf, erfc: error	function and complementary	erf(3M)
gamma: log gamma	function.	gamma(3M)
hypot: Euclidean distance	function.	hypot(3M)
of a common object file	function. /line number entries	ldlread(3X)
matherr: error-handling	function.	matherr(3M)
prof: profile within a	function.	prof(5)
math: math	functions and constants.	math(5)
intro: introduction to	functions and libraries.	intro(3)
j0, j1, jn, y0, y1, yn: Bessel	functions. bessel:	bessel(3M)
password and file encryption	functions. crypt:	crypt(3X)
logarithm, power, square root	functions. /sqrt: exponential,	exp(3M)
remainder, absolute value	functions. /floor, ceiling,	floor(3M)
ocurse: optimized screen	functions.	ocurse(3X)
300, 300s: handle special	functions of DASI 300 and 300s/	300(1)
terminals. hp: handle special	functions of Hewlett-Packard	hp(1)
terminal. 450: handle special	functions of the DASI 450	450(1)
sinh, cosh, tanh: hyperbolic	functions.	sinh(3M)
atan, atan2: trigonometric	functions. /tan, asin, acos,	trig(3M)
	fusage: disk access profiler.	fusage(1M)
using a file or file/	fuser: identify processes	fuser(1M)
fread,	fwrite: binary input/output.	fread(3S)
connect accounting records.	fwtmp, wtmpfix: manipulate	fwtmp(1M)
moo: guessing	game.	moo(6)
back: the	game of backgammon.	back(6)
bj: the	game of black jack.	bj(6)
craps: the	game of craps.	craps(6)
wump: the	game of hunt-the-wumpus.	wump(6)
trk: trekkie	game.	trk(6)

intro: introduction to	games.	intro(6)
gamma: log	gamma function.	gamma(3M)
file.	gateways: routed configuration	gateways(4)
number to string. ecvt, fcvt,	gcvt: convert floating-point	ecvt(3C)
tekset, td: graphical device/	gdev: hpd, erase, hardcopy,	gdev(1G)
	ged: graphical editor.	ged(1G)
the cc command.	gencc: create a front-end to	gencc(1M)
maze:	generate a maze.	maze(6)
abort:	generate a SIGABRT.	abort(3C)
cflow:	generate C flowgraph.	cflow(1)
cross-reference. cxref:	generate C program	cxref(1)
classification and/ chrtbl:	generate character	chrtbl(1M)
by user ID. diskusg:	generate disk accounting data	diskusg(1M)
makekey:	generate encryption key.	makekey(1)
terminal. ctermid:	generate file name for	ctermid(3S)
crypt, setkey, encrypt:	generate hashing encryption.	crypt(3C)
i-numbers. ncheck:	generate path names from	ncheck(1M)
lexical tasks. lex:	generate programs for simple	lex(1)
/srand48, seed48, lcong48:	generate uniformly distributed/	drand48(3C)
and command forms. astgen:	generate/modify ASSIST menus	astgen(1)
srand: simple random-number	generator. rand,	rand(3C)
gets, fgets:	get a string from a stream.	gets(3S)
get:	get a version of an SCCS file.	get(1)
getsockopt, setsockopt:	get and set options on/	getsockopt(2)
ulimit:	get and set user limits.	ulimit(2)
the user. cuserid:	get character login name of	cuserid(3S)
getc, getchar, fgetc, getw:	get character or word from a/	getc(3S)
through the/ nlsgetcall:	get client's data passed	nlsgetcall(3n)
getdtablesize:	get descriptor table size.	getdtablesize(2)
nlist:	get entries from name list.	nlist(3C)
umask: set and	get file creation mask.	umask(2)
stat, fstat:	get file status.	stat(2)
statfs, fstatfs:	get file system information.	statfs(2)
ustat:	get file system statistics.	ustat(2)
information. sysfs:	get file system type	sysfs(2)
file.	get: get a version of an SCCS	get(1)
/setgrent, endgrent, fgetgrent:	get group file entry.	getgrent(3C)
getlogin:	get login name.	getlogin(3C)
logname:	get login name.	logname(1)
msgget:	get message queue.	msgget(2)
getpw:	get name from UID.	getpw(3C)
getpeername:	get name of connected peer.	getpeername(2)
system. uname:	get name of current CTIX	uname(2)
provider. nlsprovider:	get name of transport	nlsprovider(3n)
host. getservaddr:	get network address of service	getservad(1M)
/setnetent, endnetent:	get network entry.	getnetent(3)
/sethostent, endhostent:	get network host entry.	gethostbyname(3)
getmsg:	get next message off a stream.	getmsg(2)
unget: undo a previous	get of an SCCS file.	unget(1)
argument vector. getopt:	get option letter from	getopt(3C)
/setpwent, endpwent, fgetpwent:	get password file entry.	getpwent(3C)
working directory. getcwd:	get path-name of current	getcwd(3C)
times. times:	get process and child process	times(2)
and/ getpid, getpgrp, getppid:	get process, process group,	getpid(2)
/setprotoent, endprotoent:	get protocol entry.	getprotoent(3)
information. t_getinfo:	get protocol-specific service	t_getinfo(3n)
/geteuid, getgid, getegid:	get real user, effective user,/	getuid(2)
getrpcbyname, getrpcbynumber:	get rpc entry. getrpcent,	getrpcent(3)

getrpcport:	get RPC port number.	getrpcport(3)
/setservent, endservent:	get service entry.	getservent(3)
semget:	get set of semaphores.	semget(2)
fgetspent, lckpwdf, ulckpwdf:	get shadow. /endspent,	getspent(3X)
identifier. shmget:	get shared memory segment	shmget(2)
getsockname:	get socket name.	getsockname(2)
t_getstate:	get the current state.	t_getstate(3)
tty:	get the name of the terminal.	tty(1)
time:	get time.	time(2)
get character or word from a/	getc, getchar, fgetc, getw:	getc(3S)
character or word from/ getc,	getchar, fgetc, getw: get	getc(3S)
current working directory.	getcwd: get path-name of	getcwd(3C)
entries and put in a file.	getdents: read directory	getdents(2)
table size.	getdtablesize: get descriptor	getdtablesize(2)
getuid, geteuid, getgid,	getegid: get real user,/	getuid(2)
environment name.	getenv: return value for	getenv(3C)
real user, effective/ getuid,	geteuid, getgid, getegid: get	getuid(2)
user,/ getuid, geteuid,	getgid, getegid: get real	getuid(2)
setgrent, endgrent,/	getgrent, getgrgid, getgrnam,	getgrent(3C)
endgrent,/ getgrent,	getgrgid, getgrnam, setgrent,	getgrent(3C)
getgrent, getgrgid,	getgrnam, setgrent, endgrent,/	getgrent(3C)
sethostent,/ gethostbyname,	gethostbyaddr, gethostent,	gethostbyname(3)
gethostent, sethostent,/	gethostbyname, gethostbyaddr,	gethostbyname(3)
gethostbyname, gethostbyaddr,	gethostent, sethostent,/	gethostbyname(3)
unique identifier of current/	gethostid, sethostid: get/set	gethostid(2)
get/set name of current host.	gethostname, sethostname:	gethostname(2)
	getlogin: get login name.	getlogin(3C)
stream.	getmsg: get next message off a	getmsg(2)
setnetent,/ getnetent,	getnetbyaddr, getnetbyname,	getnetent(3)
getnetent, getnetbyaddr,	getnetbyname, setnetent,/	getnetent(3)
getnetbyname, setnetent,/	getnetent, getnetbyaddr,	getnetent(3)
argument vector.	getopt: get option letter from	getopt(3C)
	getopt: parse command options.	getopt(1)
options. getopts,	getoptcvt: parse command	getopts(1)
command options.	getopts, getoptcvt: parse	getopts(1)
	getpass: read a password.	getpass(3C)
connected peer.	getpeername: get name of	getpeername(2)
process group, and/ getpid,	getpgrp, getppid: get process,	getpid(2)
process, process group, and/	getpid, getpgrp, getppid: get	getpid(2)
group, and/ getpid, getpgrp,	getppid: get process, process	getpid(2)
getprotoent, getprotobynumber,	getprotobyname, setprotoent,/	getprotoent(3)
getprotobyname,/ getprotoent,	getprotobynumber,	getprotoent(3)
getprotobyname, setprotoent,/	getprotoent, getprotobynumber,	getprotoent(3)
	getpw: get name from UID.	getpw(3C)
setpwent, endpwent,/	getpwent, getpwuid, getpwnam,	getpwent(3C)
getpwent, getpwuid,	getpwnam, setpwent, endpwent,/	getpwent(3C)
endpwent,/ getpwent,	getpwuid, getpwnam, setpwent,	getpwent(3C)
get rpc entry. getrpcent,	getrpcbyname, getrpcbynumber:	getrpcent(3)
getrpcbynumber: get rpc/	getrpcent, getrpcbyname,	getrpcent(3)
number.	getrpcport: get RPC port	getrpcport(3)
a stream.	gets, fgets: get a string from	gets(3S)
address of service host.	getservaddr: get network	getservad(1M)
getservent, getservbyport,	getservbyname, setservent,/	getservent(3)
setservent,/ getservent,	getservbyport, getservbyname,	getservent(3)
getservbyname, setservent,/	getservent, getservbyport,	getservent(3)
gettimeofday, settimeofday:	get/set date and time.	gettimeofday(2)
gethostname, sethostname:	get/set name of current host.	gethostname(2)
current/ gethostid, sethostid:	get/set unique identifier of	gethostid(2)

	getsockname: get socket name.	getsockname(2)
and set options on sockets.	getsockopt, setsockopt: get	getsockopt(2)
endspent, fgetspent, lckpwdf,/	getspent, getspnam, setspent,	getspent(3X)
fgetspent, lckpwdf,/ getspent,	getspnam, setspent, endspent,	getspent(3X)
get/set date and time.	gettimeofday, settimeofday:	gettimeofday(2)
and terminal settings used by	getty. gettydefs: speed	gettydefs(4)
modes, speed, and line/	getty: set terminal type,	getty(1M)
ct: spawn	getty to a remote terminal.	ct(1C)
settings used by getty.	gettydefs: speed and terminal	gettydefs(4)
getegid: get real user,/	getuid, geteuid, getgid,	getuid(2)
pututline, setutent,/ getut:	getutent, getutid, getutline,	getut(3C)
setutent,/ getut: getutent,	getutid, getutline, pututline,	getut(3C)
getut: getutent, getutid,	getutline, pututline,/	getut(3C)
from a/ getc, getchar, fgetc,	getw: get character or word	getc(3S)
common CTIX system terms and/	glossary: definitions of	glossary(1)
ascftime,/ ctime, localtime,	gmtime, asctime, cftime,	ctime(3C)
fish: play	''Go Fish''.	fish(6)
setjmp, longjmp: non-local	goto.	setjmp(3C)
string, format of graphical/	gps: graphical primitive	gps(4)
graph: draw a	graph.	graph(1G)
sag: system activity	graph.	sag(1G)
commands. graphics: access	graphical and numerical	graphics(1G)
/network useful with	graphical commands.	stat(1G)
/erase, hardcopy, tekset, td:	graphical device routines and/	gdev(1G)
ged:	graphical editor.	ged(1G)
primitive string, format of	graphical files. /graphical	gps(4)
toc: dtoc, ttoc, vtoc:	graphical table of contents/	toc(1G)
gutil:	graphical utilities.	gutil(1G)
numerical commands.	graphics: access graphical and	graphics(1G)
tplot:	graphics filters.	tplot(1G)
plot:	graphics interface.	plot(4)
subroutines. plot:	graphics interface	plot(3X)
mvt: typeset documents, view	graphs, and slides. mmt,	mmt(1)
package for typesetting view	graphs and slides. /macro	mv(5)
	greek: select terminal filter.	greek(1)
pattern.	grep: search a file for a	grep(1)
/user, effective user, real	group, and effective group/	getuid(2)
/getppid: get process, process	group, and parent process IDs.	getpid(2)
chown, chgrp: change owner or	group.	chown(1)
endgrent, fgetgrent: get	group file entry. /setgrent,	getgrent(3C)
group:	group file.	group(4)
setpgrp: set process	group ID.	setpgrp(2)
id: print user and	group IDs and names.	id(1M)
real group, and effective	group IDs. /effective user,	getuid(2)
setuid, setgid: set user and	group IDs.	setuid(2)
Remote File Sharing user and	group mapping. idload:	idload(1M)
newgrp: log in to a new	group.	newgrp(1M)
chown: change owner and	group of a file.	chown(2)
a signal to a process or a	group of processes. /send	kill(2)
update, and regenerate	groups of programs. /maintain,	make(1)
checkers. pwck,	grpck: password/group file	pwck(1M)
ssignal,	gsignal: software signals.	ssignal(3C)
install or relocate a PT or	GT local printer. /mvtpy:	mktpy(1)
download. tdl,	gtdl, ptdl: RS-232 terminal	tdl(1)
hangman:	guess the word.	hangman(6)
moo:	guessing game.	moo(6)
	gutil: graphical utilities.	gutil(1G)
/for Interphase V/TAPE 3200	half-inch tape controller.	ipt(7)

stape: SCSI quarter-inch and	half-inch tape.	stape(7)
system state. shutdown,	halt: shut down system, change	shutdown(1M)
DASI 300 and 300s/ 300, 300s:	handle special functions of	300(1)
Hewlett-Packard/ hp:	handle special functions of	hp(1)
the DASI 450 terminal. 450:	handle special functions of	450(1)
varargs:	handle variable argument list.	varargs(5)
curses: terminal screen	handling and optimization/	curses(3X)
setchrclass: character	handling. /_tolower, _toupper,	ctype(3C)
	hangman: guess the word.	hangman(6)
nohup: run a command immune to	hangups and quits.	nohup(1)
graphical/ gdev: hpd, erase,	hardcopy, tekset, td:	gdev(1G)
hinv:	hardware inventory.	hinv(1M)
hcreate, hdestroy: manage	hash search tables. hsearch,	hsearch(3C)
spell, hashmake, spellin,	hashcheck: find spelling/	spell(1)
setkey, encrypt: generate	hashing encryption. crypt,	crypt(3C)
find spelling errors. spell,	hashmake, spellin, hashcheck:	spell(1)
search tables. hsearch,	hcreate, hdestroy: manage hash	hsearch(3C)
dump.	hd: hexadecimal and ascii file	hd(1)
tables. hsearch, hcreate,	hdestroy: manage hash search	hsearch(3C)
file. scnhdr: section	header for a common object	scnhdr(4)
files. filehdr: file	header for common object	filehdr(4)
limits: file	header for/	limits(4)
unistd: file	header for symbolic constants.	unistd(4)
file. ldfhread: read the file	header of a common object	ldfhread(3X)
/seek to the optional file	header of a common object/	ldohseek(3X)
/read an indexed/named section	header of a common object/	ldshread(3X)
ldahread: read the archive	header of a member of an/	ldahread(3X)
helpadm: make changes to the	Help Facility database.	helpadm(1M)
help: CTIX system	Help Facility.	help(1)
	help: CTIX system Help Facility.	help(1)
Help Facility database.	helpadm: make changes to the	helpadm(1M)
tape file archiver. hpio:	Hewlett-Packard 2645A terminal	hpio(1)
/handle special functions of	Hewlett-Packard terminals.	hp(1)
dump. hd:	hexadecimal and ascii file	hd(1)
	hinv: hardware inventory.	hinv(1M)
libdev: manipulate Volume	Home Blocks (VHB).	libdev(3X)
fortune: print a random,	hopefully interesting, adage.	fortune(6)
/ntohs: convert values between	host and network byte order.	byteorder(3)
endhostent: get network	host entry. /sethostent,	gethostbyname(3)
unique identifier of current	host. /sethostid: get/set	gethostid(2)
get/set name of current	host. /sethostname:	gethostname(2)
get network address of service	host. getservaddr:	getservad(1M)
/set or print the Internet	host name of the current/	hostname(1)
change Remote File Sharing	host password. rfpasswd:	rfpasswd(1M)
rwhod:	host status server.	rwhod(1M)
or print identifier of current	host system. hostid: set	hostid(1)
identifier of current host/	hostid: set or print	hostid(1)
Internet host name of the/	hostname: set or print the	hostname(1)
packets to network	hosts. /send ICMP ECHO_REQUEST	ping(1M)
of Hewlett-Packard terminals.	hp: handle special functions	hp(1)
td: graphical device/ gdev:	hpd, erase, hardcopy, tekset,	gdev(1G)
terminal tape file archiver.	hpio: Hewlett-Packard 2645A	hpio(1)
manage hash search tables.	hsearch, hcreate, hdestroy:	hsearch(3C)
convert values between host/	htonl, htons, ntohl, ntohs:	byteorder(3)
values between host/ htonl,	htons, ntohl, ntohs: convert	byteorder(3)
wump: the game of	hunt-the-wumpus.	wump(6)
sinh, cosh, tanh:	hyperbolic functions.	sinh(3M)
hyphen: find	hyphenated words.	hyphen(1)

function.	hypot: Euclidean distance	hypot(3M)
network hosts. ping: send	ICMP ECHO_REQUEST packets to	ping(1M)
Protocol.	icmp: Internet Control Message	icmp(7)
disk accounting data by user	ID. diskusg: generate	diskusg(1M)
semaphore set or shared memory	ID. /remove a message queue,	ipcrm(1)
and names.	id: print user and group IDs	id(1M)
setpgrp: set process group	ID.	setpgrp(2)
issue: issue	identi fication file.	issue(4)
fstyp: determine file system	identi fier.	fstyp(1M)
/sethostid: get/set unique	identi fier of current host.	gethostid(2)
system. hostid: set or print	identi fier of current host	hostid(1)
get shared memory segment	identi fier. shmget:	shmget(2)
using keywords. locate:	identify a CTIX system command	locate(1)
file or file/ fuser:	identify processes using a	fuser(1M)
what:	identify SCCS files.	what(1)
user and group mapping.	idload: Remote File Sharing	idload(1M)
id: print user and group	IDs and names.	id(1M)
group, and parent process	IDs. /get process, process	getpid(2)
group, and effective group	IDs. /effective user, real	getuid(2)
setgid: set user and group	IDs. setuid,	setuid(2)
/fpgetsticky, fpsetsticky:	IEEE floating point/	fpgetround(3)
interface parameters.	ifconfig: configure network	ifconfig(1M)
mkifile: make an	ifile from an object file.	mkifile(1M)
core: format of core	image file.	core(4)
format of curses screen	image file.. scr_dump:	scr_dump(4)
crash: examine system	images.	crash(1M)
nohup: run a command	immune to hangups and quits.	nohup(1)
limits: file header for	implementation-speci fic/	limits(4)
C language preprocessor	include files. /determine	includes(1)
finc: fast	incremental backup.	finc(1M)
dirent: file system	independent directory entry.	dirent(4)
/tgoto, tputs: terminal	independent operations.	otermcap(3X)
for formatting a permuted	index. /the macro package	mptx(5)
of a/ ldtbindex: compute the	index of a symbol table entry	ldtbindex(3X)
ptx: permuted	index.	ptx(1)
a common/ ldtbread: read an	indexed symbol table entry of	ldtbread(3X)
ldshread, ldnshread: read an	indexed/named section header/	ldshread(3X)
ldsseek, ldnsseek: seek to an	indexed/named section of a/	ldsseek(3X)
receipt of an orderly release	indication. /acknowledge	t_rcvrel(3n)
receive a unit data error	indication. t_rcvuderr:	t_rcvuderr(3)
family.	inet: Internet protocol	inet(7)
inet_ntoa, inet_makeaddr,/	inet_addr, inet_network,	inet(3)
"super-server".	inetd: internet	inetd(1M)
configuration file for	inetd (internet/ inetd.conf:	inetd.conf(4)
for inetd (internet/	inetd.conf: configuration file	inetd.conf(4)
/inet_ntoa, inet_makeaddr,	inet_lnaof, inet_netof:/	inet(3)
/inet_network, inet_ntoa,	inet_makeaddr, inet_lnaof,/	inet(3)
/inet_makeaddr, inet_lnaof,	inet_netof: Internet address/	inet(3)
inet_makeaddr,/ inet_addr,	inet_network, inet_ntoa,	inet(3)
inet_addr, inet_network,	inet_ntoa, inet_makeaddr,/	inet(3)
terminfo descriptions.	infocmp: compare or print out	infocmp(1M)
inittab: script for the	init process.	inittab(4)
initialization.	init, telinit: process control	init(1M)
init, telinit: process control	initialization.	init(1M)
/drvload, powerfail: system	initialization procedures.	brc(1M)
terminfo database. tput:	initialize a terminal or query	tput(1)
volume. iv:	initialize and maintain	iv(1)
socket. connect:	initiate a connection on a	connect(2)

t_sndrel:	initiate an orderly release.	t_sndrel(3n)
process. popen, pclose:	initiate pipe to/from a	popen(3S)
process.	inittab: script for the init	inittab(4)
clri: clear	i-node.	clri(1M)
inode: format of an	i-node.	inode(4)
number of free disk blocks and	i-nodes. df: report	df(1M)
start and stop terminal	input and output. /manually	rsterm(1M)
sscanf: convert formatted	input. scanf, fscanf,	scanf(3S)
push character back into	input stream. ungetc:	ungetc(3S)
fread, fwrite: binary	input/output.	fread(3S)
poll: STREAMS	input/output multiplexing.	poll(2)
stdio: standard buffered	input/output package.	stdio(3S)
fileno: stream status	inquiries. /feof, clearerr,	ferror(3S)
uustat: uucp status	inquiry and job control.	uustat(1C)
with information from/ pwconv:	install and update /etc/shadow	pwconv(1M)
with information/ pwunconv:	install and update /etc/shadow	pwunconv(1M)
using the mkfs(1)/ qinstall:	install and verify software	qinstall(1)
install:	install commands.	install(1M)
directories. cpset:	install object files in binary	cpset(1M)
local printer. mktpy, mvtpy:	install or relocate a PT or GT	mktpy(1)
ctinstall:	install software.	ctinstall(1)
abs: return	integer absolute value.	abs(3C)
/l64a: convert between long	integer and base-64 ASCII/	a64l(3C)
sputl, sgetl: access long	integer data in a/	sputl(3X)
atol, atoi: convert string to	integer. strtol,	strtol(3C)
3-byte integers and long	integers. /convert between	l3tol(3C)
bcopy:	interactive block copy.	bcopy(1M)
system. mailx:	interactive message processing	mailx(1)
print a random, hopefully	interesting, adage. fortune:	fortune(6)
tset: set terminal, terminal	interface, and terminal/	tset(1)
module. timod: Transport	Interface cooperating STREAMS	timod(7)
err: error-logging	interface.	err(7)
V/TAPE 3200 half-inch/ ipt:	interface for Interphase	ipt(7)
qic:	interface for QIC tape.	qic(7)
lo: software loopback network	interface.	lo(7)
lp: parallel printer	interface.	lp(7)
mem, kmem: system memory	interface.	mem(7)
ifconfig: configure network	interface parameters.	ifconfig(1M)
plot: graphics	interface.	plot(4)
STREAMS/ tirdwr: Transport	Interface read/write interface	tirdwr(7)
/Transport Interface read/write	interface STREAMS module.	tirdwr(7)
plot: graphics	interface subroutines.	plot(3X)
swap: swap administrative	interface.	swap(1M)
termio: general terminal	interface.	termio(7)
tiop: terminal accelerator	interface.	tiop(7)
logging and event/ log:	interface to STREAMS error	log(7)
telnet: user	interface to TELNET protocol.	telnet(1)
protocol. tftp: user	interface to the DARPA TFTP	tftp(1)
tty: controlling terminal	interface.	tty(7)
vme: VME bus	interface.	vme(7)
detach serial lines as network	interfaces. /attach and	slattach(1M)
/inet_lnaof, inet_netof:	Internet address manipulation/	inet(3)
Protocol. icmp:	Internet Control Message	icmp(7)
named:	Internet domain name server.	named(1M)
Protocol server. ftpd: DARPA	Internet File Transfer	ftpd(1M)
hostname: set or print the	Internet host name of the/	hostname(1)
names and numbers for the	internet. networks:	networks(4)
slipd: switched Serial Line	Internet Protocol control/	slipd(1M)

inet:	Internet protocol family.	inet(7)
ip:	Internet Protocol.	ip(7)
protocols: list of	Internet protocols.	protocols(4)
services: list of	Internet services.	services(4)
inetd:	internet "super-server".	inetd(1M)
/configuration file for inetd	(internet "super-server").	inetd.conf(4)
Protocol. tcp:	Internet Transmission Control	tcp(7)
Protocol. udp:	Internet User Datagram	udp(7)
half-inch/ ipt: interface for	Interphase V/TAPE 3200	ipt(7)
spline:	interpolate smooth curve.	spline(1G)
characters. asa:	interpret ASA carriage control	asa(1)
sno: SNOBOL	interpreter.	sno(1)
syntax. csh: a shell (command	interpreter) with C-like	csh(1)
pipe: create an	interprocess channel.	pipe(2)
facilities/ ipcs: report	inter-process communication	ipcs(1)
stdipc, ftok: standard	interprocess communication/	stdipc(3C)
suspend execution for an	interval. sleep:	sleep(1)
sleep: suspend execution for	interval.	sleep(3C)
application programs. intro:	introduction to commands and	intro(1)
intro:	introduction to file formats.	intro(4)
libraries. intro:	introduction to functions and	intro(3)
intro:	introduction to games.	intro(6)
intro:	introduction to miscellany.	intro(5)
intro:	introduction to special files.	intro(7)
and error numbers. intro:	introduction to system calls	intro(2)
generate path names from	i-numbers. ncheck:	ncheck(1M)
hinv: hardware	inventory.	hinv(1M)
tio: tape	io filter.	tio(1)
select: synchronous	I/O multiplexing.	select(2)
table. rtab: Remote	I/O Processor configuration	rtab(4)
riopqry: query Remote	I/O Processor for online data.	riopqry(1M)
configure system for Remote	I/O Processor. riopcfg:	riopcfg(1M)
streamio: STREAMS	ioctl commands.	streamio(7)
	ioctl: control device.	ioctl(2)
	ip: Internet Protocol.	ip(7)
semaphore set or shared/	ipcrm: remove a message queue,	ipcrm(1)
communication facilities/	ipcs: report inter-process	ipcs(1)
V/TAPE 3200 half-inch tape/	ipt: interface for Interphase	ipt(7)
/islower, isupper, isalpha,	isalnum, isspace, iscntrl,/	ctype(3C)
/isxdigit, islower, isupper,	isalpha, isalnum, isspace,/	ctype(3C)
/ispunct, isprint, isgraph,	isascii, tolower, toupper,/	ctype(3C)
terminal. ttyname,	isatty: find name of a	ttyname(3C)
/isalpha, isalnum, isspace,	iscntrl, ispunct, isprint,/	ctype(3C)
isupper, isalpha, isalnum,/	isdigit, isxdigit, islower,	ctype(3C)
/iscntrl, ispunct, isprint,	isgraph, isascii, tolower,/	ctype(3C)
isalnum,/ isdigit, isxdigit,	islower, isupper, isalpha,	ctype(3C)
for floating point NaN/	isnan: isnand, isnanf: test	isnan(3C)
floating point NaN/ isnan:	isnand, isnanf: test for	isnan(3C)
point NaN/ isnan: isnand,	isnanf: test for floating	isnan(3C)
/isspace, iscntrl, ispunct,	isprint, isgraph, isascii,/	ctype(3C)
/isalnum, isspace, iscntrl,	ispunct, isprint, isgraph,/	ctype(3C)
/isupper, isalpha, isalnum,	isspace, iscntrl, ispunct,/	ctype(3C)
system:	issue a shell command.	system(3S)
issue:	issue identification file.	issue(4)
isdigit, isxdigit, islower,	isupper, isalpha, isalnum,/	ctype(3C)
isalpha, isalnum,/ isdigit,	isxdigit, islower, isupper,	ctype(3C)
news: print news	items.	news(1)
volume.	iv: initialize and maintain	iv(1)

functions. bessel:	j0, j1, jn, y0, y1, yn: Bessel	bessel(3M)
functions. bessel: j0,	j1, jn, y0, y1, yn: Bessel	bessel(3M)
bj: the game of black	jack.	bj(6)
functions. bessel: j0, j1,	jn, y0, y1, yn: Bessel	bessel(3M)
operator.	join: relational database	join(1)
/lrand48, nrand48, mrand48,	jrand48, srand48, seed48,/	drand48(3C)
mkdbsym: load symbols in	kernel debugger.	mkdbsym(1M)
port. dbconsole: change the	kernel debugger system console	dbconsole(1M)
makekey: generate encryption	key.	makekey(1)
a CTIX system command using	keywords. locate: identify	locate(1)
killall:	kill all active processes.	killall(1M)
process or a group of/	kill: send a signal to a	kill(2)
	kill: terminate a process.	kill(1)
processes.	killall: kill all active	killall(1M)
mem,	kmem: system memory interface.	mem(7)
quiz: test your	knowledge.	quiz(6)
3-byte integers and long/	l3tol, ltol3: convert between	l3tol(3C)
integer and base-64/ a64l,	l64a: convert between long	a64l(3C)
labelit: provide	labels for file systems.	labelit(1M)
scanning and processing	language. awk: pattern	awk(1)
arbitrary-precision arithmetic	language. bc:	bc(1)
efl: extended FORTRAN	language.	efl(1)
scanning and processing	language. nawk: pattern	nawk(1)
cpp: the C	language preprocessor.	cpp(1)
files. includes: determine C	language preprocessor include	includes(1)
command programming	language. /standard/restricted	sh(1)
cftime:	language specific strings.	cftime(4)
chargefee, ckpacct, dodisk,	lastlogin, monacct, nulladm,/	acctsh(1M)
shl: shell	layer manager.	shl(1)
/setspent, endspent, fgetspent,	lckpwdf, ulckpwdf: get shadow.	getspent(3X)
/jrand48, srand48, seed48,	lcong48: generate uniformly/	drand48(3C)
object files.	ld: link editor for common	ld(1)
object file. ldclose,	ldaclose: close a common	ldclose(3X)
header of a member of an/	ldahread: read the archive	ldahread(3X)
file for reading. ldopen,	ldaopen: open a common object	ldopen(3X)
common object file.	ldclose, ldaclose: close a	ldclose(3X)
drivers.	lddrv: manage loadable	lddrv(1M)
	ldeeprom: load EEPROM.	ldeeprom(1M)
of floating-point/ frexp,	ldexp, modf: manipulate parts	frexp(3C)
access routines.	ldfcn: common object file	ldfcn(4)
of a common object file.	ldfhread: read the file header	ldfhread(3X)
name for common object file/	ldgetname: retrieve symbol	ldgetname(3X)
line number entries/ ldlread,	ldlinit, ldlitem: manipulate	ldlread(3X)
number/ ldlread, ldlinit,	ldlitem: manipulate line	ldlread(3X)
manipulate line number/	ldlread, ldlinit, ldlitem:	ldlread(3X)
line number entries of a/	ldlseek, ldnlseek: seek to	ldlseek(3X)
entries of a section/ ldlseek,	ldnlseek: seek to line number	ldlseek(3X)
entries of a section/ ldrseek,	ldnrseek: seek to relocation	ldrseek(3X)
indexed/named/ ldshread,	ldnshread: read an	ldshread(3X)
indexed/named/ ldsseek,	ldnsseek: seek to an	ldsseek(3X)
file header of a common/	ldohseek: seek to the optional	ldohseek(3X)
object file for reading.	ldopen, ldaopen: open a common	ldopen(3X)
relocation entries of a/	ldrseek, ldnrseek: seek to	ldrseek(3X)
indexed/named section header/	ldshread, ldnshread: read an	ldshread(3X)
socket configuration. slink,	ldsocket: STREAMS linker, load	slink(1)
indexed/named section of a/	ldsseek, ldnsseek: seek to an	ldsseek(3X)
of a symbol table entry of a/	ldtbindex: compute the index	ldtbindex(3X)
symbol table entry of a/	ldtbread: read an indexed	ldtbread(3X)

table of a common object/	ldtbseek: seek to the symbol	ldtbseek(3X)
getopt: get option	letter from argument vector.	getopt(3C)
generate programs for simple	lexical tasks. lex:	lex(1)
update. lsearch,	lfind: linear search and	lsearch(3C)
Blocks (VHB).	libdev: manipulate Volume Home	libdev(3X)
introduction to functions and	libraries. intro:	intro(3)
chkshlib: compare shared	libraries tool.	chkshlib(1)
relation for an object	library. /find ordering	lorder(1)
portable/ ar: archive and	library maintainer for	ar(1)
mkshlib: create a shared	library.	mkshlib(1)
t_alloc: allocate a	library structure.	t_alloc(3n)
t_free: free a	library structure.	t_free(3n)
t_sync: synchronize transport	library.	t_sync(3n)
implementation-speci fic/	limits: file header for	limits(4)
ulimit: get and set user	limits.	ulimit(2)
an out-going terminal	line connection. /establish	dial(3C)
type, modes, speed, and	line discipline. /set terminal	getty(1M)
type, modes, speed, and	line discipline. /set terminal	uugetty(1M)
slipd: switched Serial	Line Internet Protocol control/	slipd(1M)
line: read one	line.	line(1)
common object file. linenum:	line number entries in a	linenum(4)
/ldlinit, ldlitem: manipulate	line number entries of a/	ldlread(3X)
ldlseek, ldnlseek: seek to	line number entries of a/	ldlseek(3X)
strip: strip symbol and	line number information from a/	strip(1)
nl:	line numbering filter.	nl(1)
out selected fields of each	line of a file. cut: cut	cut(1)
send/cancel requests to an LP	line printer. lp, cancel:	lp(1)
lpset: set parallel	line printer options.	lpset(1M)
lpr:	line printer spooler.	lpr(1)
	line: read one line.	line(1)
lsearch, lfind:	linear search and update.	lsearch(3C)
col: filter reverse	line-feeds.	col(1)
in a common object file.	linenum: line number entries	linenum(4)
/attach and detach serial	lines as network interfaces.	slattach(1M)
files. comm: select or reject	lines common to two sorted	comm(1)
file for uucp communications	lines. Devices: configuration	Devices(5)
device. fold: fold long	lines for finite width output	fold(1)
head: give first few	lines.	head(1)
uniq: report repeated	lines in a file.	uniq(1)
subsequent/ paste: merge same	lines of several files or	paste(1)
directories. link, unlink:	link and unlink files and	link(1M)
files. ld:	link editor for common object	ld(1)
a.out: common assembler and	link editor output.	a.out(4)
	link: link to a file.	link(2)
cp, ln, mv: copy,	link, or move files.	cp(1)
link:	link to a file.	link(2)
slink, ldsocket: STREAMS	linker, load socket/	slink(1)
lists from proto file; set	links based on. /out file	qlist(1)
	lint: a C program checker.	lint(1)
ls:	list contents of directory.	ls(1)
nlist: get entries from name	list.	nlist(3C)
and statistics for file system	list file names	ff(1M)
an. bcheck: print the	list of blocks associated with	bcheck(1M)
nm: print name	list of common object file.	nm(1)
by fsck and/ checklist:	list of file systems processed	checklist(4)
hosts:	list of hosts on network.	hosts(4)
protocols:	list of Internet protocols.	protocols(4)
services:	list of Internet services.	services(4)

terminal number. ttytype:	list of terminal types by	ttytype(4)
from a common object file.	list: produce C source listing	list(1)
handle variable argument	list. varargs:	varargs(5)
output of a varargs argument	list. /print formatted	vprintf(3S)
t_listen:	listen for a connect request.	t_listen(3n)
socket. listen:	listen for connections on a	listen(2)
data passed through the	listener. /get client's	nlsgetcall(3n)
nlsadmin: network	listener service/	nlsadmin(1M)
nlsrequest: format and send	listener service request/	nlsrequest(3n)
file. list: produce C source	listing from a common object	list(1)
xargs: construct argument	list(s) and execute command.	xargs(1)
links/ qlist: print out file	lists from proto file; set	qlist(1)
volcopy: make	literal copy of file system.	volcopy(1M)
files. cp,	ln, mv: copy, link, or move	cp(1)
interface.	lo: software loopback network	lo(7)
ldeeprom:	load EEPROM.	ldeeprom(1M)
/ldsocket: STREAMS linker,	load socket configuration.	slink(1)
debugger. mkdbsym:	load symbols in kernel	mkdbsym(1M)
drivers:	loadable device drivers.	drivers(7)
lddrv: manage	loadable drivers.	lddrv(1M)
cftime, ascftime,/ ctime,	localtime, gmtime, asctime,	ctime(3C)
the virtual system/ conlocate:	locate a terminal to use as	conlocate(1M)
command. path:	locate executable file for	path(1)
command using keywords.	locate: identify a CTIX system	locate(1)
end, etext, edata: last	locations in program.	end(3C)
memory. plock:	lock process, text, or data in	plock(2)
files.	lockf: record locking on	lockf(3C)
regions of a file.	locking: exclusive access to	locking(2)
lockf: record	locking on files.	lockf(3C)
gamma:	log gamma function.	gamma(3M)
newgrp:	log in to a new group.	newgrp(1M)
error logging and event/	log: interface to STREAMS	log(7)
exponential, logarithm,/ exp,	log, log10, pow, sqrt:	exp(3M)
/usr/adm/loginlog:	log of failed login attempts.	loginlog(4)
logarithm, power,/ exp, log,	log10, pow, sqrt: exponential,	exp(3M)
/log10, pow, sqrt: exponential,	logarithm, power, square root/	exp(3M)
errpt: process a report of	logged errors.	errpt(1M)
rwho: who is	logged in on local network.	rwho(1)
strclean: STREAMS error	logger cleanup program.	strclean(1M)
strerr: STREAMS error	logger daemon.	strerr(1M)
/interface to STREAMS error	logging and event tracing.	log(7)
/log of failed	login attempts.	loginlog(4)
networks. netrc:	login file for remote	netrc(4)
getlogin: get	login name.	getlogin(3C)
logname: get	login name.	logname(1)
cuserid: get character	login name of the user.	cuserid(3S)
logname: return	login name of user.	logname(3X)
passwd: change	login password.	passwd(1)
rlogin: remote	login.	rlogin(1)
rlogind: remote	login server.	rlogind(1M)
	login: sign on.	login(1)
up a C shell environment at	login time. cprofile: setting	cprofile(4)
setting up an environment at	login time. profile:	profile(4)
	logname: get login name.	logname(1)
user.	logname: return login name of	logname(3X)
a64l, l64a: convert between	long integer and base-64 ASCII/	a64l(3C)
sputl, sgetl: access	long integer data in a/	sputl(3X)
between 3-byte integers and	long integers. /ltol3: convert	l3tol(3C)

output device. fold: fold	long lines for finite width	fold(1)
setjmp,	longjmp: non-local goto.	setjmp(3C)
finger: user information	lookup program.	finger(1)
lo: software	loopback network interface.	lo(7)
for an object library.	lorder: find ordering relation	lorder(1)
mklost+found: make a	lost+found directory for fsck.	mklostfnd(1M)
nice: run a command at	low priority.	nice(1)
send/cancel requests to an	LP line printer. lp, cancel:	lp(1)
interface.	lp: parallel printer	lp(7)
disable: enable/disable	LP printers. enable,	enable(1)
reject: allow or prevent	LP requests. accept,	accept(1M)
/lpshut, lpmove: start/stop the	LP scheduler and move/	lpsched(1M)
lpadmin: configure the	LP spooling system.	lpadmin(1M)
lpstat: print	LP status information.	lpstat(1)
spooling system.	lpadmin: configure the LP	lpadmin(1M)
scheduler/ lpsched, lpshut,	lpmove: start/stop the LP	lpsched(1M)
	lpr: line printer spooler.	lpr(1)
start/stop the LP scheduler/	lpsched, lpshut, lpmove:	lpsched(1M)
printer options.	lpset: set parallel line	lpset(1M)
LP scheduler and/ lpsched,	lpshut, lpmove: start/stop the	lpsched(1M)
information.	lpstat: print LP status	lpstat(1)
jrand48/ drand48, erand48,	lrand48, nrand48, mrand48,	drand48(3C)
directory.	ls: list contents of	ls(1)
and update.	lsearch, lfind: linear search	lsearch(3C)
pointer.	lseek: move read/write file	lseek(2)
integers and long/ l3tol,	ltol3: convert between 3-byte	l3tol(3C)
	m4: macro processor.	m4(1)
mega, unixpc,.	machid: mc68k, miti, mini,	machid(1)
values:	machine-dependent values.	values(5)
/access long integer data in a	machine-independent fashion.	sputl(3X)
permuted index. mptx: the	macro package for formatting a	mptx(5)
documents. mm: the MM	macro package for formatting	mm(5)
view graphs and/ mv: a troff	macro package for typesetting	mv(5)
m4:	macro processor.	m4(1)
pages. man:	macros for formatting manual	man(5)
me:	macros for formatting papers.	me(5)
formatted with the MM	macros. /print/check documents	mm(1)
ms: text formatting	macros.	ms(5)
/rebuild the data base for the	mail aliases file.	newaliases(1)
users or read mail.	mail, rmail: send mail to	mail(1)
sendmail:	mail routing program.	sendmail(1M)
processing system.	mailx: interactive message	mailx(1)
malloc, free, realloc, calloc:	main memory allocator.	malloc(3C)
/mallopt, mallinfo: fast	main memory allocator.	malloc(3X)
regenerate groups of/ make:	maintain, update, and	make(1)
iv: initialize and	maintain volume.	iv(1)
ar: archive and library	maintainer for portable/	ar(1)
SCCS file. delta:	make a delta (change) to an	delta(1)
mkdir:	make a directory.	mkdir(2)
or ordinary file. mknod:	make a directory, or a special	mknod(2)
for fsck. mklost+found:	make a lost+found directory	mklostfnd(1M)
mktemp:	make a unique file name.	mktemp(3C)
file. mkifile:	make an ifile from an object	mkifile(1M)
Facility database. helpadm:	make changes to the Help	helpadm(1M)
mkdir, mkdirs:	make directories.	mkdir(1)
system. volcopy:	make literal copy of file	volcopy(1M)
regenerate groups of/	make: maintain, update, and	make(1)
mkhosts:	make node name commands.	mkhosts(1M)

banner:	make posters.	banner(1)
session. script:	make typescript of terminal	script(1)
key.	makekey: generate encryption	makekey(1)
/realloc, calloc, mallopt,	mallinfo: fast main memory/	malloc(3X)
main memory allocator.	malloc, free, realloc, calloc:	malloc(3C)
mallopt, mallinfo: fast main/	malloc, free, realloc, calloc,	malloc(3X)
malloc, free, realloc, calloc,	mallopt, mallinfo: fast main/	malloc(3X)
manual pages.	man: macros for formatting	man(5)
/tfind, tdelete, twalk:	manage binary search trees.	tsearch(3C)
hsearch, hcreate, hdestroy:	manage hash search tables.	hsearch(3C)
lddrv:	manage loadable drivers.	lddrv(1M)
unnotify, evwait, evnowait:	manage notifications. notify,	notify(2)
endpoint. t_optmgmt:	manage options for a transport	t_optmgmt(3n)
passmgmt: password files	management.	passmgmt(1M)
window: window	management primitives.	window(7)
sigignore, sigpause: signal	management. /sigrelse,	sigset(2)
wm: window	management.	wm(1)
shl: shell layer	manager.	shl(1)
records. fwtmp, wtmpfix:	manipulate connect accounting	fwtmp(1M)
of/ ldlread, ldlinit, ldlitem:	manipulate line number entries	ldlread(3X)
frexp, ldexp, modf:	manipulate parts of/	frexp(3C)
comment section. mcs:	manipulate the object file	mcs(1)
route: manually	manipulate the routing tables.	route(1M)
(VHB). libdev:	manipulate Volume Home Blocks	libdev(3X)
/inet_netof: Internet address	manipulation routines.	inet(3)
man: macros for formatting	manual pages.	man(5)
routing tables. route:	manually manipulate the	route(1M)
terminal input and/ rsterm:	manually start and stop	rsterm(1M)
ascii:	map of ASCII character set.	ascii(5)
port to RPC program number	mapper. portmap: DARPA	portmap(1M)
File Sharing user and group	mapping. idload: Remote	idload(1M)
scsimap: set	mappings for SCSI devices.	scsimap(1M)
files. diffmk:	mark differences between	diffmk(1)
umask: set file-creation mode	mask.	umask(1)
set and get file creation	mask. umask:	umask(2)
table. master:	master device information	master(4)
masterupd: update the	master file.	masterupd(1M)
File Sharing name server	master file. rfmaster: Remote	rfmaster(4)
information table.	master: master device	master(4)
file.	masterupd: update the master	masterupd(1M)
regular expression compile and	match routines. regexp:	regexp(5)
math:	math functions and constants.	math(5)
constants.	math: math functions and	math(5)
eqn, neqn, checkeq: format	mathematical text for nroff or/	eqn(1)
function.	matherr: error-handling	matherr(3M)
maze: generate a	maze.	maze(6)
unixpc,. machid:	mc68k, miti, mini, mega,	machid(1)
file comment section.	mcs: manipulate the object	mcs(1)
machid: mc68k, miti, mini,	mega, unixpc,.	machid(1)
interface.	mem, kmem: system memory	mem(7)
memcpy, memset:/ memory:	memccpy, memchr, memcmp,	memory(3C)
memset:/ memory: memccpy,	memchr, memcmp, memcpy,	memory(3C)
memory: memccpy, memchr,	memcmp, memcpy, memset: memory/	memory(3C)
/memccpy, memchr, memcmp,	memcpy, memset: memory/	memory(3C)
free, realloc, calloc: main	memory allocator. malloc,	malloc(3C)
mallopt, mallinfo: fast main	memory allocator. /calloc,	malloc(3X)
shmctl: shared	memory control operations.	shmctl(2)
queue, semaphore set or shared	memory ID. /remove a message	ipcrm(1)

mem, kmem: system	memory interface.	mem(7)
memcmp, memcpy, memset:/	memory: memccpy, memchr,	memory(3C)
memcmp, memcpy, memset:	memory operations. /memchr,	memory(3C)
shmop: shared	memory operations.	shmop(2)
lock process, text, or data in	memory. plock:	plock(2)
shmget: get shared	memory segment identifier.	shmget(2)
/memchr, memcmp, memcpy,	memset: memory operations.	memory(3C)
astgen: generate/modify ASSIST	menus and command forms.	astgen(1)
sort: sort and/or	merge files.	sort(1)
files. acctmerg:	merge or add total accounting	acctmerg(1M)
files or subsequent/ paste:	merge same lines of several	paste(1)
	mesg: permit or deny messages.	mesg(1)
msgctl:	message control operations.	msgctl(2)
recv, recvfrom: receive a	message from a socket.	recv(2)
send listener service request	message. /format and	nlsrequest(3n)
getmsg: get next	message off a stream.	getmsg(2)
putmsg: send a	message on a stream.	putmsg(2)
msgop:	message operations.	msgop(2)
mailx: interactive	message processing system.	mailx(1)
icmp: Internet Control	Message Protocol.	icmp(7)
msgget: get	message queue.	msgget(2)
or shared/ ipcrm: remove a	message queue, semaphore set	ipcrm(1)
t_error: produce error	message.	t_error(3n)
send, sendto: send a	message to a socket.	send(2)
mesg: permit or deny	messages.	mesg(1)
sys_nerr: system error	messages. /errno, sys_errlist,	perror(3C)
strace: print STREAMS trace	messages.	strace(1M)
machid: mc68k, miti,	mini, mega, unixpc,.	machid(1)
driver. clone: open any	minor device on a STREAMS	clone(7)
machid: mc68k,	miti, mini, mega, unixpc,.	machid(1)
kernel debugger.	mkdbsym: load symbols in	mkdbsym(1M)
	mkdir: make a directory.	mkdir(2)
directories.	mkdir, mkdirs: make	mkdir(1)
	mkfs: construct a file system.	mkfs(1M)
/and verify software using the	mkfs(1) proto file database.	qinstall(1)
commands.	mkhosts: make node name	mkhosts(1M)
object file.	mkifile: make an ifile from an	mkifile(1M)
lost+found directory for/	mklost+found: make a	mklostfnd(1M)
	mknod: build special file.	mknod(1M)
special or ordinary file.	mknod: make a directory, or a	mknod(2)
library.	mkshlib: create a shared	mkshlib(1)
name.	mktemp: make a unique file	mktemp(3C)
relocate a PT or GT local/	mktpy, mvtpy: install or	mktpy(1)
documents formatted with the/	mm, checkmm: print/check	mm(1)
formatting documents. mm: the	MM macro package for	mm(5)
documents formatted with the	MM macros. /print/check	mm(1)
formatting documents.	mm: the MM macro package for	mm(5)
view graphs, and slides.	mmt, mvt: typeset documents,	mmt(1)
table.	mnttab: mounted file system	mnttab(4)
chmod: change	mode.	chmod(1)
umask: set file-creation	mode mask.	umask(1)
chmod: change	mode of file.	chmod(2)
getty: set terminal type,	modes, speed, and line/	getty(1M)
uugetty: set terminal type,	modes, speed, and line/	uugetty(1M)
bs: a compiler/interpreter for	modest-sized programs.	bs(1)
floating-point/ frexp, ldexp,	modf: manipulate parts of	frexp(3C)
touch: update access and	modification times of a file.	touch(1)
utime: set file access and	modification times.	utime(2)

Interface cooperating STREAMS	module. timod: Transport	timod(7)
read/write interface STREAMS	module. /Transport Interface	tirdwr(7)
/ckpacct, dodisk, lastlogin,	monacct, nulladm, prctmp,/	acctsh(1M)
profile.	monitor: prepare execution	monitor(3C)
	moo: guessing game.	moo(6)
	more, page: text perusal.	more(1)
mount:	mount a file system.	mount(2)
and remote/ mount, umount:	mount and unmount file systems	mount(1M)
rmnttry: attempt to	mount remote resources.	rmnttry(1M)
mountd: NFS	mount request server.	mountd(1M)
setmnt: establish	mount table.	setmnt(1M)
systems. mountall, umountall:	mount, unmount multiple file	mountall(1M)
System/ nmountall, numountall:	mount, unmount Network File	nmountall(1M)
rmountall, rumountall:	mount, unmount Remote File/	rmountall(1M)
unmount multiple file/	mountall, umountall: mount,	mountall(1M)
server.	mountd: NFS mount request	mountd(1M)
mnttab:	mounted file system table.	mnttab(4)
rmtab: remotely	mounted file system table.	rmtab(4)
rmntstat: display	mounted resource information.	rmntstat(1M)
rmount: queue remote resource	mounts.	rmount(1M)
showmount: show all remote	mounts.	showmount(1M)
mvdir:	move a directory.	mvdir(1M)
cp, ln, mv: copy, link, or	move files.	cp(1)
lseek:	move read/write file pointer.	lseek(2)
the LP scheduler and	move requests. /start/stop	lpsched(1M)
formatting a permuted index.	mptx: the macro package for	mptx(5)
/erand48, lrand48, nrand48,	mrand48, jrand48, srand48,/	drand48(3C)
	ms: text formatting macros.	ms(5)
operations.	msgctl: message control	msgctl(2)
	msgget: get message queue.	msgget(2)
	msgop: message operations.	msgop(2)
/umountall: mount, unmount	multiple file systems.	mountall(1M)
poll: STREAMS input/output	multiplexing.	poll(2)
select: synchronous I/O	multiplexing.	select(2)
sxt: STREAMS	multiplexor.	sxt(7)
run commands performed for	multi-user environment. /rc3:	rc2(1M)
typesetting view graphs and/	mv: a troff macro package for	mv(5)
cp, ln,	mv: copy, link, or move files.	cp(1)
	mvdir: move a directory.	mvdir(1M)
graphs, and slides. mmt,	mvt: typeset documents, view	mmt(1)
PT or GT local/ mktpy,	mvtpy: install or relocate a	mktpy(1)
server.	named: Internet domain name	named(1M)
test for floating point	NaN (Not-A-Number). /isnanf:	isnan(3C)
processing language.	nawk: pattern scanning and	nawk(1)
systems processed by fsck and	ncheck. /list of file	checklist(4)
from i-numbers.	ncheck: generate path names	ncheck(1M)
mathematical text for/ eqn,	neqn, checkeq: format	eqn(1)
definitions for eqn and	neqn. /special character	eqnchar(5)
File.	netcf: Network Configuration	netcf(4)
networks.	netrc: login file for remote	netrc(4)
	netstat: show network status.	netstat(1)
host. getservaddr: get	network address of service	getservad(1M)
values between host and	network byte order. /convert	byteorder(3)
netcf:	Network Configuration File.	netcf(4)
setnetent, endnetent: get	network entry. /getnetbyname,	getnetent(3)
/numountall: mount, unmount	Network File System resources.	nmountall(1M)
statistics. nfsstat:	Network File System	nfsstat(1M)
/sethostent, endhostent: get	network host entry.	gethostbyname(3)

ICMP ECHO_REQUEST packets to	network hosts. ping: send	ping(1M)
hosts: list of hosts on	network.	hosts(4)
lo: software loopback	network interface.	lo(7)
ifconfig: configure	network interface parameters.	ifconfig(1M)
and detach serial lines as	network interfaces. /attach	slattach(1M)
administration. nlsadmin:	network listener service	nlsadmin(1M)
Remote File Sharing domain and	network names. dname: print	dname(1M)
routed:	network routing daemon.	routed(1M)
status of nodes on local	network. ruptime: display	ruptime(1)
who is logged in on local	network. rwho:	rwho(1)
netstat: show	network status.	netstat(1)
commands. stat: statistical	network useful with graphical	stat(1G)
uucpd, ouucpd:	network uucp servers.	uucpd(1M)
for the internet.	networks: names and numbers	networks(4)
netrc: login file for remote	networks.	netrc(4)
base for the mail aliases/	newaliases: rebuild the data	newaliases(1)
a text file.	newform: change the format of	newform(1)
	newgrp: log in to a new group.	newgrp(1M)
news: print	news items.	news(1)
/store, delete, firstkey,	nextkey: database subroutines.	dbm(3X)
nfsd, biod:	NFS daemons.	nfsd(1M)
configuration file. exports:	NFS file systems export	exports(4)
mountd:	NFS mount request server.	mountd(1M)
nfssys: common shared	NFS system calls.	nfssys(2)
	nfsd, biod: NFS daemons.	nfsd(1M)
statistics.	nfsstat: Network File System	nfsstat(1M)
system calls.	nfssys: common shared NFS	nfssys(2)
process.	nice: change priority of a	nice(2)
of running process by changing	nice. renice: alter priority	renice(1)
priority.	nice: run a command at low	nice(1)
	nl: line numbering filter.	nl(1)
list.	nlist: get entries from name	nlist(3C)
service administration.	nlsadmin: network listener	nlsadmin(1M)
passed through the listener.	nlsgetcall: get client's data	nlsgetcall(3n)
transport provider.	nlsprovider: get name of	nlsprovider(3n)
listener service request/	nlsrequest: format and send	nlsrequest(3n)
object file.	nm: print name list of common	nm(1)
unmount Network File System/	nmountall, numountall: mount,	nmountall(1M)
mkhosts: make	node name commands.	mkhosts(1M)
createdev: create device	nodes for assorted device/	createdev(1M)
ruptime: display status of	nodes on local network.	ruptime(1)
hangups and quits.	nohup: run a command immune to	nohup(1)
setjmp, longjmp:	non-local goto.	setjmp(3C)
test for floating point NaN	(Not-A-Number). /isnanf:	isnan(3C)
rfuadmin: Remote File Sharing	notification shell script.	rfuadmin(1M)
evwait, evnowait: manage	notifications. /unnotify,	notify(2)
evnowait: manage/	notify, unnotify, evwait,	notify(2)
drand48, erand48, lrand48,	nrand48, mrand48, jrand48,/	drand48(3C)
	nroff: format text.	nroff(1)
format mathematical text for	nroff or troff. /checkeq:	eqn(1)
tbl: format tables for	nroff or troff.	tbl(1)
constructs. deroff: remove	nroff/troff, tbl, and eqn	deroff(1)
name server query.	nsquery: Remote File Sharing	nsquery(1M)
between host/ htonl, htons,	ntohl, ntohs: convert values	byteorder(3)
host and/ htonl, htons, ntohl,	ntohs: convert values between	byteorder(3)
null: the	null file.	null(7)
/dodisk, lastlogin, monacct,	nulladm, prctmp, prdaily,/	acctsh(1M)
nl: line	numbering filter.	nl(1)

number: convert Arabic	numerals to English.	number(6)
graphics: access graphical and	numerical commands.	graphics(1G)
Network File/ nmountall,	numountall: mount, unmount	nmountall(1M)
dis:	object code disassembler.	dis(1)
ldfcn: common	object file access routines.	ldfcn(4)
mcs: manipulate the	object file comment section.	mcs(1)
conv: common	object file converter.	conv(1)
cprs: compress a common	object file.	cprs(1)
dump selected parts of an	object file. dump:	dump(1)
ldopen, ldaopen: open a common	object file for reading.	ldopen(3X)
number entries of a common	object file function. /line	ldlread(3X)
ldaclose: close a common	object file. ldclose,	ldclose(3X)
the file header of a common	object file. ldfhread: read	ldfhread(3X)
of a section of a common	object file. /number entries	ldlseek(3X)
file header of a common	object file. /to the optional	ldohseek(3X)
of a section of a common	object file. /entries	ldrseek(3X)
section header of a common	object file. /indexed/named	ldshread(3X)
section of a common	object file. /indexed/named	ldsseek(3X)
symbol table entry of a common	object file. /the index of a	ldtbindex(3X)
symbol table entry of a common	object file. /read an indexed	ldtbread(3X)
the symbol table of a common	object file. /seek to	ldtbseek(3X)
number entries in a common	object file. linenum: line	linenum(4)
C source listing from a common	object file. list: produce	list(1)
mkifile: make an ifile from an	object file.	mkifile(1M)
nm: print name list of common	object file.	nm(1)
information for a common	object file. /relocation	reloc(4)
section header for a common	object file. scnhdr:	scnhdr(4)
information from a common	object file. /and line number	strip(1)
entry. /symbol name for common	object file symbol table	ldgetname(3X)
format. syms: common	object file symbol table	syms(4)
file header for common	object files. filehdr:	filehdr(4)
directories. cpset: install	object files in binary	cpset(1M)
ld: link editor for common	object files.	ld(1)
sizes in bytes of common	object files. /print section	size(1)
find ordering relation for an	object library. lorder:	lorder(1)
number. factor:	obtain the prime factors of a	factor(1)
od:	octal dump.	od(1)
functions.	ocurse: optimized screen	ocurse(3X)
	od: octal dump.	od(1)
query Remote I/O Processor for	online data. riopqry:	riopqry(1M)
reading. ldopen, ldaopen:	open a common object file for	ldopen(3X)
fopen, freopen, fdopen:	open a stream.	fopen(3S)
STREAMS driver. clone:	open any minor device on a	clone(7)
dup: duplicate an	open file descriptor.	dup(2)
dup2: duplicate an	open file descriptor.	dup2(3C)
open:	open for reading or writing.	open(2)
seekdir,/ directory:	opendir, readdir, telldir,	directory(3X)
starter: information about the	operating system for beginning/	starter(1)
prf:	operating system profiler.	prf(7)
/prfdc, prfsnap, prfpr:	operating system profiler.	profiler(1M)
commands performed to stop the	operating system. rc0: run	rc0(1M)
uconf: configure the	operating system.	uconf(1M)
bzero: bit and byte string	operations. bcopy, bcmp,	bstring(3)
rewinddir, closedir: directory	operations. /telldir, seekdir,	directory(3X)
memcmp, memcpy, memset: memory	operations. /memccpy, memchr,	memory(3C)
msgctl: message control	operations.	msgctl(2)
msgop: message	operations.	msgop(2)
tputs: terminal independent	operations. /tgetstr, tgoto,	otermcap(3X)

management.	passmgmt: password files	passmgmt(1M)
	passwd: change login password.	passwd(1)
	passwd: password file.	passwd(4)
functions. crypt:	password and file encryption	crypt(3X)
/endpwent, fgetpwent: get	password file entry.	getpwent(3C)
putpwent: write	password file entry.	putpwent(3C)
putspent: write shadow	password file entry.	putspent(3X)
passwd:	password file.	passwd(4)
shadow:	password file.	shadow(4)
passmgmt:	password files management.	passmgmt(1M)
getpass: read a	password.	getpass(3C)
passwd: change login	password.	passwd(1)
Remote File Sharing host	password. rfpasswd: change	rfpasswd(1M)
pwck, grpck:	password/group file checkers.	pwck(1M)
several files or subsequent/	paste: merge same lines of	paste(1)
for command.	path: locate executable file	path(1)
dirname: deliver portions of	path names. basename,	basename(1)
ncheck: generate	path names from i-numbers.	ncheck(1M)
directory. getcwd: get	path-name of current working	getcwd(3C)
grep: search a file for a	pattern.	grep(1)
processing language. awk:	pattern scanning and	awk(1)
processing language. nawk:	pattern scanning and	nawk(1)
egrep: search a file for a	pattern using full regular/	egrep(1)
signal.	pause: suspend process until	pause(2)
expand files. pack,	pcat, unpack: compress and	pack(1)
a process. popen,	pclose: initiate pipe to/from	popen(3S)
get name of connected	peer. getpeername:	getpeername(2)
rc2, rc3: run commands	performed for multi-user/	rc2(1M)
operating/ rc0: run commands	performed to stop the	rc0(1M)
check the uucp directories and	permissions file. uucheck:	uucheck(1M)
mesg:	permit or deny messages.	mesg(1)
macro package for formatting a	permuted index. mptx: the	mptx(5)
ptx:	permuted index.	ptx(1)
format. acct:	per-process accounting file	acct(4)
acctcms: command summary from	per-process accounting/	acctcms(1M)
sys_nerr: system error/	perror, errno, sys_errlist,	perror(3C)
pg: file	perusal filter for CRTs.	pg(1)
more, page: text	perusal.	more(1)
CRTs.	pg: file perusal filter for	pg(1)
split: split a file into	pieces.	split(1)
packets to network hosts.	ping: send ICMP ECHO_REQUEST	ping(1M)
channel.	pipe: create an interprocess	pipe(2)
tee:	pipe fitting.	tee(1)
popen, pclose: initiate	pipe to/from a process.	popen(3S)
fish:	play "Go Fish".	fish(6)
data in memory.	plock: lock process, text, or	plock(2)
	plot: graphics interface.	plot(4)
subroutines.	plot: graphics interface	plot(3X)
ftell: reposition a file	pointer in a stream. /rewind,	fseek(3S)
lseek: move read/write file	pointer.	lseek(2)
multiplexing.	poll: STREAMS input/output	poll(2)
to/from a process.	popen, pclose: initiate pipe	popen(3S)
kernel debugger system console	port. dbconsole: change the	dbconsole(1M)
serstat: display serial	port error statistics.	serstat(1M)
getrpcport: get RPC	port number.	getrpcport(3)
mapper. portmap: DARPA	port to RPC program number	portmap(1M)
and library maintainer for	portable archives. /archive	ar(1)
basename, dirname: deliver	portions of path names.	basename(1)

program number mapper.	portmap: DARPA port to RPC	portmap(1M)
banner: make	posters.	banner(1)
logarithm,/ exp, log, log10,	pow, sqrt: exponential,	exp(3M)
/sqrt: exponential, logarithm,	power, square root functions.	exp(3M)
brc, bcheckrc, drvload,	powerfail: system/	brc(1M)
	pr: print files.	pr(1)
/lastlogin, monacct, nulladm,	prctmp, prdaily, prtacct,/	acctsh(1M)
/monacct, nulladm, prctmp,	prdaily, prtacct, runacct,/	acctsh(1M)
for troff. cw, checkcw:	prepare constant-width text	cw(1)
monitor:	prepare execution profile.	monitor(3C)
cpp: the C language	preprocessor.	cpp(1)
includes: determine C language	preprocessor include files.	includes(1)
accept, reject: allow or	prevent LP requests.	accept(1M)
unget: undo a	previous get of an SCCS file.	unget(1)
profiler.	prf: operating system	prf(7)
profiler: prfld, prfstat,	prfdc, prfsnap, prfpr:/	profiler(1M)
prfsnap, prfpr:/ profiler:	prfld, prfstat, prfdc,	profiler(1M)
/prfstat, prfdc, prfsnap,	prfpr: operating system/	profiler(1M)
system/ /prfld, prfstat, prfdc,	prfsnap, prfpr: operating	profiler(1M)
prfpr:/ profiler: prfld,	prfstat, prfdc, prfsnap,	profiler(1M)
factor: obtain the	prime factors of a number.	factor(1)
graphical/ gps: graphical	primitive string, format of	gps(4)
types:	primitive system data types.	types(5)
window: window management	primitives.	window(7)
interesting, adage. fortune:	print a random, hopefully	fortune(6)
prs:	print an SCCS file.	prs(1)
date:	print and set the date.	date(1)
cal:	print calendar.	cal(1)
of a file. sum:	print checksum and block count	sum(1)
editing activity. sact:	print current SCCS file	sact(1)
cat: concatenate and	print files.	cat(1)
pr:	print files.	pr(1)
vprintf, vfprintf, vsprintf:	print formatted output of a/	vprintf(3S)
printf, fprintf, sprintf:	print formatted output.	printf(3S)
host system. hostid: set or	print identifier of current	hostid(1)
lpstat:	print LP status information.	lpstat(1)
object file. nm:	print name list of common	nm(1)
system. uname:	print name of current CTIX	uname(1)
news:	print news items.	news(1)
proto file; set links/ qlist:	print out file lists from	qlist(1)
infocmp: compare or	print out terminfo/	infocmp(1M)
file(s). acctcom: search and	print process accounting	acctcom(1)
domain and network/ dname:	print Remote File Sharing	dname(1M)
of common object files. size:	print section sizes in bytes	size(1)
strace:	print STREAMS trace messages.	strace(1M)
of the/ hostname: set or	print the Internet host name	hostname(1)
associated with an. bcheck:	print the list of blocks	bcheck(1M)
names. id:	print user and group IDs and	id(1M)
formatted with/ mm, checkmm:	print/check documents	mm(1)
lp: parallel	printer interface.	lp(7)
requests to an LP line	printer. /cancel: send/cancel	lp(1)
or relocate a PT or GT local	printer. /mvtpy: install	mktpy(1)
lpset: set parallel line	printer options.	lpset(1M)
lpr: line	printer spooler.	lpr(1)
disable: enable/disable LP	printers. enable,	enable(1)
print formatted output.	printf, fprintf, sprintf:	printf(3S)
rtpenable: real-time	priorities enabled/disabled.	rtpenable(1M)
nice: run a command at low	priority.	nice(1)

nice: change	priority of a process.	nice(2)
changing nice. renice: alter	priority of running process by	renice(1)
errors. errpt:	process a report of logged	errpt(1M)
acct: enable or disable	process accounting.	acct(2)
acctprc1, acctprc2:	process accounting.	acctprc(1M)
acctcom: search and print	process accounting file(s).	acctcom(1)
alarm: set a	process alarm clock.	alarm(2)
times. times: get	process and child process	times(2)
/alter priority of running	process by changing nice.	renice(1)
init, telinit:	process control/	init(1M)
timex: time a command; report	process data and system/	timex(1)
exit, _exit: terminate	process.	exit(2)
fork: create a new	process.	fork(2)
/getpgrp, getppid: get process,	process group, and parent/	getpid(2)
setpgrp: set	process group ID.	setpgrp(2)
process group, and parent	process IDs. /get process,	getpid(2)
inittab: script for the init	process.	inittab(4)
kill: terminate a	process.	kill(1)
nice: change priority of a	process.	nice(2)
kill: send a signal to a	process or a group of/	kill(2)
initiate pipe to/from a	process. popen, pclose:	popen(3S)
getpid, getpgrp, getppid: get	process, process group, and/	getpid(2)
Remote File Sharing daemon	process. rfudaemon:	rfudaemon(1M)
ps: report	process status.	ps(1)
memory. plock: lock	process, text, or data in	plock(2)
times: get process and child	process times.	times(2)
wait: wait for child	process to stop or terminate.	wait(2)
ptrace:	process trace.	ptrace(2)
pause: suspend	process until signal.	pause(2)
wait: await completion of	process.	wait(1)
/list of file systems	processed by fsck and ncheck.	checklist(4)
to a process or a group of	processes. /send a signal	kill(2)
killall: kill all active	processes.	killall(1M)
structure. fuser: identify	processes using a file or file	fuser(1M)
awk: pattern scanning and	processing language.	awk(1)
nawk: pattern scanning and	processing language.	nawk(1)
extproc: turn external	processing on or off.	extproc(1M)
mailx: interactive message	processing system.	mailx(1)
rtab: Remote I/O	Processor configuration table.	rtab(4)
en: Ethernet	Processor.	en(7)
enpstart: configure Ethernet	processor.	enpstart(1M)
riopqry: query Remote I/O	Processor for online data.	riopqry(1M)
m4: macro	processor.	m4(1)
system for Remote I/O	Processor. riopcfg: configure	riopcfg(1M)
a common object file. list:	produce C source listing from	list(1)
t_error:	produce error message.	t_error(3n)
	prof: display profile data.	prof(1)
function.	prof: profile within a	prof(5)
profile.	profil: execution time	profil(2)
prof: display	profile data.	prof(1)
monitor: prepare execution	profile.	monitor(3C)
profil: execution time	profile.	profil(2)
environment at login time.	profile: setting up an	profile(4)
prof:	profile within a function.	prof(5)
fusage: disk access	profiler.	fusage(1M)
prf: operating system	profiler.	prf(7)
prfdc, prfsnap, prfpr:/	profiler: prfld, prfstat,	profiler(1M)
prfpr: operating system	profiler. /prfdc, prfsnap,	profiler(1M)

sadp: disk access	profiler.	sadp(1M)
standard/restricted command	programming language. /the	sh(1)
software using the mkfs(1)	proto file database. /verify	qinstall(1)
on. /print out file lists from	proto file; set links based	qlist(1)
arp: Address Resolution	Protocol.	arp(7)
/switched Serial Line Internet	Protocol control facility.	slipd(1M)
/setprotoent, endprotoent: get	protocol entry.	getprotoent(3)
inet: Internet	protocol family.	inet(7)
icmp: Internet Control Message	Protocol.	icmp(7)
ip: Internet	Protocol.	ip(7)
DARPA Internet File Transfer	Protocol server. ftpd:	ftpd(1M)
telnetd: DARPA TELNET	protocol server.	telnetd(1M)
DARPA Trivial File Transfer	Protocol server. tftpd:	tftpd(1M)
Internet Transmission Control	Protocol. tcp:	tcp(7)
user interface to TELNET	protocol. telnet:	telnet(1)
interface to the DARPA TFTP	protocol. tftp: user	tftp(1)
udp: Internet User Datagram	Protocol.	udp(7)
Dialers: ACU/modem calling	protocols.	Dialers(5)
protocols.	protocols: list of Internet	protocols(4)
information. t_getinfo: get	protocol-specific service	t_getinfo(3n)
update:	provide disk synchronization.	update(1M)
arithmetic:	provide drill in number facts.	arithmetic(6)
systems. labelit:	provide labels for file	labelit(1M)
true, false:	provide truth values.	true(1)
get name of transport	provider. nlsprovider:	nlsprovider(3n)
	prs: print an SCCS file.	prs(1)
/nulladm, prctmp, prdaily,	prtacct, runacct, shutacct,/	acctsh(1M)
	ps: report process status.	ps(1)
/generate uniformly distributed	pseudo-random numbers.	drand48(3C)
/mvtpy: install or relocate a	PT or GT local printer.	mktpy(1)
download. tdl, gtdl,	ptdl: RS-232 terminal	tdl(1)
	ptrace: process trace.	ptrace(2)
	ptx: permuted index.	ptx(1)
stream. ungetc:	push character back into input	ungetc(3S)
put character or word on a/	putc, putchar, fputc, putw:	putc(3S)
character or word on a/ putc,	putchar, fputc, putw: put	putc(3S)
environment.	putenv: change or add value to	putenv(3C)
stream.	putmsg: send a message on a	putmsg(2)
entry.	putpwent: write password file	putpwent(3C)
stream.	puts, fputs: put a string on a	puts(3S)
password file entry.	putspent: write shadow	putspent(3X)
/getutent, getutid, getutline,	pututline, setutent, endutent,/	getut(3C)
a/ putc, putchar, fputc,	putw: put character or word on	putc(3S)
file checkers.	pwck, grpck: password/group	pwck(1M)
/etc/shadow with information/	pwconv: install and update	pwconv(1M)
	pwd: working directory name.	pwd(1)
/etc/shadow with information/	pwunconv: install and update	pwunconv(1M)
qic: interface for	QIC tape.	qic(7)
software using the mkfs(1)/	qinstall: install and verify	qinstall(1)
from proto file; set links/	qlist: print out file lists	qlist(1)
	qsort: quicker sort.	qsort(3C)
tape. stape: SCSI	quarter-inch and half-inch	stape(7)
File Sharing name server	query. nsquery: Remote	nsquery(1M)
online data. riopqry:	query Remote I/O Processor for	riopqry(1M)
tput: initialize a terminal or	query terminfo database.	tput(1)
queuedefs: at/batch/cron	queue description file.	queuedefs(4)
msgget: get message	queue.	msgget(2)
rmount:	queue remote resource mounts.	rmount(1M)

ipcrm: remove a message	queue, semaphore set or shared/	ipcrm(1)
request. rumount: cancel	queued remote resource	rumount(1M)
description file.	queuedefs: at/batch/cron queue	queuedefs(4)
qsort:	quicker sort.	qsort(3C)
command immune to hangups and	quits. nohup: run a	nohup(1)
	quiz: test your knowledge.	quiz(6)
random-number generator.	rand, srand: simple	rand(3C)
adage. fortune: print a	random, hopefully interesting,	fortune(6)
rand, srand: simple	random-number generator.	rand(3C)
fsplit: split FORTRAN,	ratfor, or efl files.	fsplit(1)
dialect.	ratfor: rational FORTRAN	ratfor(1)
ratfor:	rational FORTRAN dialect.	ratfor(1)
stop the operating system.	rc0: run commands performed to	rc0(1M)
performed for multi-user/	rc2, rc3: run commands	rc2(1M)
for multi-user/ rc2,	rc3: run commands performed	rc2(1M)
execution.	rcmd: remote shell command	rcmd(1)
routines for returning a/	rcmd, rresvport, ruserok:	rcmd(3)
	rcp: remote file copy.	rcp(1)
getpass:	read a password.	getpass(3C)
entry of a common/ ldtbread:	read an indexed symbol table	ldtbread(3X)
header/ ldshread, ldnshread:	read an indexed/named section	ldshread(3X)
in a file. getdents:	read directory entries and put	getdents(2)
read:	read from file.	read(2)
rmail: send mail to users or	read mail. mail,	mail(1)
line:	read one line.	line(1)
	read: read from file.	read(2)
member of an/ ldahread:	read the archive header of a	ldahread(3X)
common object file. ldfhread:	read the file header of a	ldfhread(3X)
directory: opendir,	readdir, telldir, seekdir,/	directory(3X)
open a common object file for	reading. ldopen, ldaopen:	ldopen(3X)
open: open for	reading or writing.	open(2)
lseek: move	read/write file pointer.	lseek(2)
tirdwr: Transport Interface	read/write interface STREAMS/	tirdwr(7)
allocator. malloc, free,	realloc, calloc: main memory	malloc(3C)
mallinfo: fast/ malloc, free,	realloc, calloc, mallopt,	malloc(3X)
enabled/disabled. rtpenable:	real-time priorities	rtpenable(1M)
reboot:	reboot the system.	reboot(1M)
mail aliases/ newaliases:	rebuild the data base for the	newaliases(1)
specify what to do upon	receipt of a signal. signal:	signal(2)
t_rcvrel: acknowledge	receipt of an orderly release/	t_rcvrel(3n)
t_rcvudata:	receive a data unit.	t_rcvudata(3)
socket. recv, recvfrom:	receive a message from a	recv(2)
indication. t_rcvuderr:	receive a unit data error	t_rcvuderr(3)
sent over a/ t_rcv:	receive data or expedited data	t_rcv(3n)
a connect/ t_rcvconnect:	receive the confirmation from	t_rcvconnect(3)
lockf:	record locking on files.	lockf(3C)
from per-process accounting	records. /command summary	acctcms(1M)
from/ errdead: extract error	records and status information	errdead(1M)
manipulate connect accounting	records. fwtmp, wtmpfix:	fwtmp(1M)
tape. frec:	recover files from a backup	frec(1M)
message from a socket.	recv, recvfrom: receive a	recv(2)
from a socket. recv,	recvfrom: receive a message	recv(2)
ed,	red: text editor.	ed(1)
execute regular expression.	regcmp, regex: compile and	regcmp(3X)
compile.	regcmp: regular expression	regcmp(1)
make: maintain, update, and	regenerate groups of programs.	make(1)
regular expression. regcmp,	regex: compile and execute	regcmp(3X)
compile and match routines.	regexp: regular expression	regexp(5)

locking: exclusive access to	regions of a file.	locking(2)
match routines. regexp:	regular expression compile and	regexp(5)
regcmp:	regular expression compile.	regcmp(1)
regex: compile and execute	regular expression. regcmp,	regcmp(3X)
file for a pattern using full	regular expressions. /search a	egrep(1)
requests. accept,	reject: allow or prevent LP	accept(1M)
sorted files. comm: select or	reject lines common to two	comm(1)
lorder: find ordering	relation for an object/	lorder(1)
join:	relational database operator.	join(1)
/receipt of an orderly	release indication.	t_rcvrel(3n)
t_sndrel: initiate an orderly	release.	t_sndrel(3n)
for a common object file.	reloc: relocation information	reloc(4)
mktpy, mvtpy: install or	relocate a PT or GT local/	mktpy(1)
ldrseek, ldnrseek: seek to	relocation entries of a/	ldrseek(3X)
common object file. reloc:	relocation information for a	reloc(4)
/fmod, fabs: floor, ceiling,	remainder, absolute value/	floor(3M)
calendar:	reminder service.	calendar(1)
adv: advertise a directory for	remote access.	adv(1M)
for returning a stream to a	remote command. /routines	rcmd(3)
uuxqt: execute	remote command requests.	uuxqt(1M)
rexec: return stream to a	remote command.	rexec(3)
rhosts:	remote equivalent users.	rhosts(4)
rexecd:	remote execution server.	rexecd(1M)
rcp:	remote file copy.	rcp(1)
administration. rfadmin:	Remote File Sharing	rfadmin(1M)
process. rfudaemon:	Remote File Sharing daemon	rfudaemon(1M)
network names. dname: print	Remote File Sharing domain and	dname(1M)
environment. rfstop: stop the	Remote File Sharing	rfstop(1M)
password. rfpasswd: change	Remote File Sharing host	rfpasswd(1M)
server master file. rfmaster:	Remote File Sharing name	rfmaster(4)
server query. nsquery:	Remote File Sharing name	nsquery(1M)
notification shell/ rfuadmin:	Remote File Sharing	rfuadmin(1M)
unadv: unadvertise a	Remote File Sharing resource.	unadv(1M)
/rumountall: mount, unmount	Remote File Sharing (RFS)/	rmountall(1M)
rfstart: start	Remote File Sharing.	rfstart(1M)
group mapping. idload:	Remote File Sharing user and	idload(1M)
configuration table. rtab:	Remote I/O Processor	rtab(4)
online data. riopqry: query	Remote I/O Processor for	riopqry(1M)
riopcfg: configure system for	Remote I/O Processor.	riopcfg(1M)
rlogin:	remote login.	rlogin(1)
rlogind:	remote login server.	rlogind(1M)
showmount: show all	remote mounts.	showmount(1M)
netrc: login file for	remote networks.	netrc(4)
rmount: queue	remote resource mounts.	rmount(1M)
rumount: cancel queued	remote resource request.	rumount(1M)
and unmount file systems and	remote resources. /mount	mount(1M)
rmnttry: attempt to mount	remote resources.	rmnttry(1M)
execution. rcmd:	remote shell command	rcmd(1)
rshd:	remote shell server.	rshd(1M)
on. Uutry: try to contact a	remote system with debugging	Uutry(1M)
ct: spawn getty to a	remote terminal.	ct(1C)
server. talkd:	remote user communication	talkd(1M)
server. fingerd:	remote user information	fingerd(1M)
table. rmtab:	remotely mounted file system	rmtab(4)
file. rmdel:	remove a delta from an SCCS	rmdel(1)
rmdir:	remove a directory.	rmdir(2)
semaphore set or/ ipcrm:	remove a message queue,	ipcrm(1)
unlink:	remove directory entry.	unlink(2)

rm, rmdir:	remove files or directories.	rm(1)
eqn constructs. deroff:	remove nroff/troff, tbl, and	deroff(1)
running process by changing/	renice: alter priority of	renice(1)
fsck, dfsck: check and	repair file systems.	fsck(1M)
uniq: report	repeated lines in a file.	uniq(1)
clock:	report CPU time used.	clock(3C)
fsize:	report file size.	fsize(1)
fsstat:	report file system status.	fsstat(1M)
communication/ ipcs:	report inter-process	ipcs(1)
blocks and i-nodes. df:	report number of free disk	df(1M)
errpt: process a	report of logged errors.	errpt(1M)
sa2, sadc: system activity	report package. sar: sa1,	sar(1M)
timex: time a command;	report process data and system/	timex(1)
ps:	report process status.	ps(1)
file. uniq:	report repeated lines in a	uniq(1)
rpcinfo:	report RPC information.	rpcinfo(1M)
sar: system activity	reporter.	sar(1)
stream. fseek, rewind, ftell:	reposition a file pointer in a	fseek(3S)
and send listener service	request message. /format	nlsrequest(3n)
cancel queued remote resource	request. rumount:	rumount(1M)
mountd: NFS mount	request server.	mountd(1M)
t_accept: accept a connect	request.	t_accept(3n)
t_listen: listen for a connect	request.	t_listen(3n)
confirmation from a connect	request. /receive the	t_rcvconnect(3)
send user-initiated disconnect	request. t_snddis:	t_snddis(3n)
reject: allow or prevent LP	requests. accept,	accept(1M)
the LP scheduler and move	requests. /lpmove: start/stop	lpsched(1M)
syslocal: special system	requests.	syslocal(2)
lp, cancel: send/cancel	requests to an LP line/	lp(1)
uuxqt: execute remote command	requests.	uuxqt(1M)
res_mkquery, res_send,	res_init, dn_comp, dn_expand:/	resolver(3)
res_init, dn_comp, dn_expand:/	res_mkquery, res_send,	resolver(3)
control. arp: address	resolution display and	arp(1M)
arp: Address	Resolution Protocol.	arp(7)
configuration file.	resolv.conf: resolver	resolver(4)
resolv.conf:	resolver configuration file.	resolver(4)
res_init, dn_comp, dn_expand:	resolver routines. /res_send,	resolver(3)
unmount of an advertised	resource. fumount: forced	fumount(1M)
rmntstat: display mounted	resource information.	rmntstat(1M)
rmount: queue remote	resource mounts.	rmount(1M)
rumount: cancel queued remote	resource request.	rumount(1M)
a Remote File Sharing	resource. unadv: unadvertise	unadv(1M)
file systems and remote	resources. /mount and unmount	mount(1M)
unmount Network File System	resources. /numountall: mount,	nmountall(1M)
attempt to mount remote	resources. rmnttry:	rmnttry(1M)
Remote File Sharing (RFS)	resources. /mount, unmount	rmountall(1M)
dn_expand:/ res_mkquery,	res_send, res_init, dn_comp,	resolver(3)
and usage examples. usage:	retrieve a command description	usage(1)
disconnect. t_rcvdis:	retrieve information from	t_rcvdis(3n)
common object file/ ldgetname:	retrieve symbol name for	ldgetname(3X)
abs:	return integer absolute value.	abs(3C)
logname:	return login name of user.	logname(3X)
command. rexec:	return stream to a remote	rexec(3)
name. getenv:	return value for environment	getenv(3C)
stat: data	returned by stat system call.	stat(5)
/ruserok: routines for	returning a stream to a remote/	rcmd(3)
col: filter	reverse line-feeds.	col(1)
file pointer in a/ fseek,	rewind, ftell: reposition a	fseek(3S)

/readdir, telldir, seekdir,	rewinddir, closedir: directory/	directory(3X)
creat: create a new file or	rewrite an existing one.	creat(2)
remote command.	rexec: return stream to a	rexec(3)
server.	rexecd: remote execution	rexecd(1M)
administration.	rfadmin: Remote File Sharing	rfadmin(1M)
name server master file.	rfmaster: Remote File Sharing	rfmaster(4)
Sharing host password.	rfpasswd: change Remote File	rfpasswd(1M)
unmount Remote File Sharing	(RFS) resources. /mount,	rmountall(1M)
Sharing.	rfstart: start Remote File	rfstart(1M)
Sharing environment.	rfstop: stop the Remote File	rfstop(1M)
notification shell script.	rfuadmin: Remote File Sharing	rfuadmin(1M)
daemon process.	rfudaemon: Remote File Sharing	rfudaemon(1M)
users.	rhosts: remote equivalent	rhosts(4)
Remote I/O Processor.	riopcfg: configure system for	riopcfg(1M)
Processor for online data.	riopqry: query Remote I/O	riopqry(1M)
	rlogin: remote login.	rlogin(1)
	rlogind: remote login server.	rlogind(1M)
directories.	rm, rmdir: remove files or	rm(1)
read mail. mail,	rmail: send mail to users or	mail(1)
SCCS file.	rmdel: remove a delta from an	rmdel(1)
	rmdir: remove a directory.	rmdir(2)
directories. rm,	rmdir: remove files or	rm(1)
resource information.	rmntstat: display mounted	rmntstat(1M)
remote resources.	rmnttry: attempt to mount	rmnttry(1M)
mounts.	rmount: queue remote resource	rmount(1M)
unmount Remote File Sharing/	rmountall, rumountall: mount,	rmountall(1M)
system table.	rmtab: remotely mounted file	rmtab(4)
chroot: change	root directory.	chroot(2)
chroot: change	root directory for a command.	chroot(1M)
logarithm, power, square	root functions. /exponential,	exp(3M)
routing tables.	route: manually manipulate the	route(1M)
gateways:	routed configuration file.	gateways(4)
daemon.	routed: network routing	routed(1M)
/tekset, td: graphical device	routines and filters.	gdev(1G)
rcmd, rresvport, ruserok:	routines for returning a/	rcmd(3)
Internet address manipulation	routines. /inet_netof:	inet(3)
common object file access	routines. ldfcn:	ldfcn(4)
expression compile and match	routines. regexp: regular	regexp(5)
dn_comp, dn_expand: resolver	routines. /res_send, res_init,	resolver(3)
graphical table of contents	routines. /dtoc, ttoc, vtoc:	toc(1G)
routed: network	routing daemon.	routed(1M)
sendmail: mail	routing program.	sendmail(1M)
route: manually manipulate the	routing tables.	route(1M)
getrpcbynumber: get	rpc entry. /getrpcbyname,	getrpcent(3)
rpcinfo: report	RPC information.	rpcinfo(1M)
getrpcport: get	RPC port number.	getrpcport(3)
rpc: Sun	rpc program number data base.	rpc(4)
portmap: DARPA port to	RPC program number mapper.	portmap(1M)
data base.	rpc: Sun rpc program number	rpc(4)
information.	rpcinfo: report RPC	rpcinfo(1M)
for returning a stream/ rcmd,	rresvport, ruserok: routines	rcmd(3)
controlling terminal's local	RS-232 channels. tp:	tp(7)
tdl, gtdl, ptdl:	RS-232 terminal download.	tdl(1)
standard/restricted/ sh,	rsh: shell, the	sh(1)
	rshd: remote shell server.	rshd(1M)
stop terminal input and/	rsterm: manually start and	rsterm(1M)
configuration table.	rtab: Remote I/O Processor	rtab(4)
priorities enabled/disabled.	rtpenable: real-time	rtpenable(1M)

	scsi: scsi control device.	scsi(7)
devices.	scsimap: set mappings for SCSI	scsimap(1M)
	sdb: symbolic debugger.	sdb(1)
program.	sdiff: side-by-side difference	sdiff(1)
string. fgrep:	search a file for a character	fgrep(1)
grep:	search a file for a pattern.	grep(1)
using full regular/ egrep:	search a file for a pattern	egrep(1)
bsearch: binary	search a sorted table.	bsearch(3C)
accounting file(s). acctcom:	search and print process	acctcom(1)
lsearch, lfind: linear	search and update.	lsearch(3C)
hcreate, hdestroy: manage hash	search tables. hsearch,	hsearch(3C)
tdelete, twalk: manage binary	search trees. tsearch, tfind,	tsearch(3C)
object file. scnhdr:	section header for a common	scnhdr(4)
object/ /read an indexed/named	section header of a common	ldshread(3X)
the object file comment	section. mcs: manipulate	mcs(1)
/to line number entries of a	section of a common object/	ldlseek(3X)
/to relocation entries of a	section of a common object/	ldrseek(3X)
/seek to an indexed/named	section of a common object/	ldsseek(3X)
common object/ size: print	section sizes in bytes of	size(1)
	sed: stream editor.	sed(1)
/mrand48, jrand48, srand48,	seed48, lcong48: generate/	drand48(3C)
section of/ ldsseek, ldnsseek:	seek to an indexed/named	ldsseek(3X)
a section/ ldlseek, ldnlseek:	seek to line number entries of	ldlseek(3X)
a section/ ldrseek, ldnrseek:	seek to relocation entries of	ldrseek(3X)
header of a common/ ldohseek:	seek to the optional file	ldohseek(3X)
common object file. ldtbseek:	seek to the symbol table of a	ldtbseek(3X)
/opendir, readdir, telldir,	seekdir, rewinddir, closedir:/	directory(3X)
shmget: get shared memory	segment identifier.	shmget(2)
brk, sbrk: change data	segment space allocation.	brk(2)
to two sorted files. comm:	select or reject lines common	comm(1)
multiplexing.	select: synchronous I/O	select(2)
greek:	select terminal filter.	greek(1)
of a file. cut: cut out	selected fields of each line	cut(1)
file. dump: dump	selected parts of an object	dump(1)
semctl:	semaphore control operations.	semctl(2)
semop:	semaphore operations.	semop(2)
ipcrm: remove a message queue,	semaphore set or shared memory/	ipcrm(1)
semget: get set of	semaphores.	semget(2)
operations.	semctl: semaphore control	semctl(2)
	semget: get set of semaphores.	semget(2)
	semop: semaphore operations.	semop(2)
t_sndudata:	send a data unit.	t_sndudata(3)
putmsg:	send a message on a stream.	putmsg(2)
send, sendto:	send a message to a socket.	send(2)
a group of processes. kill:	send a signal to a process or	kill(2)
over a connection. t_snd:	send data or expedited data	t_snd(3n)
to network hosts. ping:	send ICMP ECHO_REQUEST packets	ping(1M)
nlsrequest: format and	send listener service request/	nlsrequest(3n)
mail. mail, rmail:	send mail to users or read	mail(1)
to a socket.	send, sendto: send a message	send(2)
request. t_snddis:	send user-initiated disconnect	t_snddis(3n)
line printer. lp, cancel:	send/cancel requests to an LP	lp(1)
aliases: aliases file for	sendmail.	aliases(4)
program.	sendmail: mail routing	sendmail(1M)
socket. send,	sendto: send a message to a	send(2)
/receive data or expedited data	sent over a connection.	t_rcv(3n)
control/ slipd: switched	Serial Line Internet Protocol	slipd(1M)
/sldetach: attach and detach	serial lines as network/	slattach(1M)

serstat: display	serial port error statistics.	serstat(1M)
error statistics.	serstat: display serial port	serstat(1M)
remote user information	server. fingerd:	fingerd(1M)
File Transfer Protocol	server. ftpd: DARPA Internet	ftpd(1M)
Remote File Sharing name	server master file. rfmaster:	rfmaster(4)
mountd: NFS mount request	server.	mountd(1M)
named: Internet domain name	server.	named(1M)
Remote File Sharing name	server query. nsquery:	nsquery(1M)
rexecd: remote execution	server.	rexecd(1M)
rlogind: remote login	server.	rlogind(1M)
rshd: remote shell	server.	rshd(1M)
rwhod: host status	server.	rwhod(1M)
remote user communication	server. talkd:	talkd(1M)
telnetd: DARPA TELNET protocol	server.	telnetd(1M)
Trivial File Transfer Protocol	server. tftpd: DARPA	tftpd(1M)
uucpd, ouucpd: network uucp	servers.	uucpd(1M)
make typescript of terminal	session. script:	script(1)
buffering to a stream.	setbuf, setvbuf: assign	setbuf(3S)
/toascci, _tolower, _toupper,	setchrclass: character/	ctype(3C)
IDs. setuid,	setgid: set user and group	setuid(2)
getgrent, getgrgid, getgrnam,	setgrent, endgrent, fgetgrent:/	getgrent(3C)
/gethostbyaddr, gethostent,	sethostent, endhostent: get/	gethostbyname(3)
identifier of/ gethostid,	sethostid: get/set unique	gethostid(2)
current host. gethostname,	sethostname: get/set name of	gethostname(2)
goto.	setjmp, longjmp: non-local	setjmp(3C)
hashing encryption. crypt,	setkey, encrypt: generate	crypt(3C)
	setmnt: establish mount table.	setmnt(1M)
/getnetbyaddr, getnetbyname,	setnetent, endnetent: get/	getnetent(3)
	setpgrp: set process group ID.	setpgrp(2)
protocol/ /getprotobyname,	setprotoent, endprotoent: get	getprotoent(3)
getpwent, getpwuid, getpwnam,	setpwent, endpwent, fgetpwent:/	getpwent(3C)
/getservbyport, getservbyname,	setservent, endservent: get/	getservent(3)
options on/ getsockopt,	setsockopt: get and set	getsockopt(2)
lckpwdf,/ getspent, getspnam,	setspent, endspent, fgetspent,	getspent(3X)
time. gettimeofday,	settimeofday: get/set date and	gettimeofday(2)
environment at/ cprofile:	setting up a C shell	cprofile(4)
login time. profile:	setting up an environment at	profile(4)
gettydefs: speed and terminal	settings used by getty.	gettydefs(4)
group IDs.	setuid, setgid: set user and	setuid(2)
	setuname: set name of system.	setuname(1M)
/getutid, getutline, pututline,	setutent, endutent, utmpname:/	getut(3C)
stream. setbuf,	setvbuf: assign buffering to a	setbuf(3S)
data in a/ sputl,	sgetl: access long integer	sputl(3X)
standard/restricted command/	sh, rsh: shell, the	sh(1)
lckpwdf, ulckpwdf: get	shadow. /endspent, fgetspent,	getspent(3X)
putspent: write	shadow password file entry.	putspent(3X)
	shadow: password file.	shadow(4)
xstr: extract and	share strings in C programs.	xstr(1)
chkshlib: compare	shared libraries tool.	chkshlib(1)
mkshlib: create a	shared library.	mkshlib(1)
operations. shmctl:	shared memory control	shmctl(2)
queue, semaphore set or	shared memory ID. /a message	ipcrm(1)
shmop:	shared memory operations.	shmop(2)
identifier. shmget: get	shared memory segment	shmget(2)
nfssys: common	shared NFS system calls.	nfssys(2)
rfadmin: Remote File	Sharing administration.	rfadmin(1M)
rfudaemon: Remote File	Sharing daemon process.	rfudaemon(1M)
dname: print Remote File	Sharing domain and network/	dname(1M)

rfstop: stop the Remote File	Sharing environment.	rfstop(1M)
rfpasswd: change Remote File	Sharing host password.	rfpasswd(1M)
file. rfmaster: Remote File	Sharing name server master	rfmaster(4)
nsquery: Remote File	Sharing name server query.	nsquery(1M)
script. rfuadmin: Remote File	Sharing notification shell	rfuadmin(1M)
unadvertise a Remote File	Sharing resource. unadv:	unadv(1M)
/mount, unmount Remote File	Sharing (RFS) resources.	rmountall(1M)
rfstart: start Remote File	Sharing.	rfstart(1M)
mapping. idload: Remote File	Sharing user and group	idload(1M)
rcmd: remote	shell command execution.	rcmd(1)
with C-like syntax. csh: a	shell (command interpreter)	csh(1)
system: issue a	shell command.	system(3S)
cprofile: setting up a C	shell environment at login/	cprofile(4)
shl:	shell layer manager.	shl(1)
shutacct, startup, turnacct:	shell procedures for/ /runacct,	acctsh(1M)
File Sharing notification	shell script. /Remote	rfuadmin(1M)
rshd: remote	shell server.	rshd(1M)
command programming/ sh, rsh:	shell, the standard/restricted	sh(1)
	shl: shell layer manager.	shl(1)
operations.	shmctl: shared memory control	shmctl(2)
segment identifier.	shmget: get shared memory	shmget(2)
operations.	shmop: shared memory	shmop(2)
mounts.	showmount: show all remote	showmount(1M)
/prdaily, prtacct, runacct,	shutacct, startup, turnacct:/	acctsh(1M)
system, change system state.	shutdown, halt: shut down	shutdown(1M)
full-duplex connection.	shutdown: shut down part of a	shutdown(2)
program. sdiff:	side-by-side difference	sdiff(1)
abort: generate a	SIGABRT.	abort(3C)
sigpause: signal/ sigset,	sighold, sigrelse, sigignore,	sigset(2)
sigset, sighold, sigrelse,	sigignore, sigpause: signal/	sigset(2)
login:	sign on.	login(1)
sigrelse, sigignore, sigpause:	signal management. /sighold,	sigset(2)
pause: suspend process until	signal.	pause(2)
what to do upon receipt of a	signal. signal: specify	signal(2)
of processes. kill: send a	signal to a process or a group	kill(2)
ssignal, gsignal: software	signals.	ssignal(3C)
/sighold, sigrelse, sigignore,	sigpause: signal management.	sigset(2)
signal/ sigset, sighold,	sigrelse, sigignore, sigpause:	sigset(2)
sigignore, sigpause: signal/	sigset, sighold, sigrelse,	sigset(2)
lex: generate programs for	simple lexical tasks.	lex(1)
generator. rand, srand:	simple random-number	rand(3C)
atan, atan2:/ trig:	sin, cos, tan, asin, acos,	trig(3M)
functions.	sinh, cosh, tanh: hyperbolic	sinh(3M)
fsize: report file	size.	fsize(1)
get descriptor table	size. getdtablesize:	getdtablesize(2)
object/ size: print section	sizes in bytes of common	size(1)
detach serial lines as/	slattach, sldetach: attach and	slattach(1M)
serial lines as/ slattach,	sldetach: attach and detach	slattach(1M)
an interval.	sleep: suspend execution for	sleep(1)
interval.	sleep: suspend execution for	sleep(3C)
documents, view graphs, and	slides. mmt, mvt: typeset	mmt(1)
typesetting view graphs and	slides. /macro package for	mv(5)
linker, load socket/	slink, ldsocket: STREAMS	slink(1)
Internet Protocol control/	slipd: switched Serial Line	slipd(1M)
current/ ttyslot: find the	slot in the utmp file of the	ttyslot(3C)
spline: interpolate	smooth curve.	spline(1G)
sno:	SNOBOL interpreter.	sno(1)
bind: bind a name to a	socket.	bind(2)

ldsocket: STREAMS linker, load	socket configuration. slink,	slink(1)
initiate a connection on a	socket. connect:	connect(2)
communication.	socket: create an endpoint for	socket(2)
listen for connections on a	socket. listen:	listen(2)
getsockname: get	socket name.	getsockname(2)
receive a message from a	socket. recv, recvfrom:	recv(2)
sendto: send a message to a	socket. send,	send(2)
get and set options on	sockets. /setsockopt:	getsockopt(2)
ctinstall: install	software.	ctinstall(1)
interface. lo:	software loopback network	lo(7)
ssignal, gsignal:	software signals.	ssignal(3C)
qinstall: install and verify	software using the mkfs(1)/	qinstall(1)
sort:	sort and/or merge files.	sort(1)
qsort: quicker	sort.	qsort(3C)
	sort: sort and/or merge files.	sort(1)
tsort: topological	sort.	tsort(1)
or reject lines common to two	sorted files. comm: select	comm(1)
bsearch: binary search a	sorted table.	bsearch(3C)
object file. list: produce C	source listing from a common	list(1)
brk, sbrk: change data segment	space allocation.	brk(2)
/unexpand: expand tabs to	spaces, and vice versa.	expand(1)
terminal. ct:	spawn getty to a remote	ct(1C)
the/ tapedrives: tape drive	specific information used by	tapedrives(4)
cftime: language	specific strings.	cftime(4)
fspec: format	specification in text files.	fspec(4)
receipt of a signal. signal:	specify what to do upon	signal(2)
/set terminal type, modes,	speed, and line discipline.	getty(1M)
/set terminal type, modes,	speed, and line discipline.	uugetty(1M)
used by getty. gettydefs:	speed and terminal settings	gettydefs(4)
spelling/ spell, hashmake,	spellin, hashcheck: find	spell(1)
spellin, hashcheck: find	spelling errors. /hashmake,	spell(1)
curve.	spline: interpolate smooth	spline(1G)
split:	split a file into pieces.	split(1)
csplit: context	split.	csplit(1)
efl files. fsplit:	split FORTRAN, ratfor, or	fsplit(1)
uucleanup: uucp	spool directory clean-up.	uucleanup(1M)
lpr: line printer	spooler.	lpr(1)
lpadmin: configure the LP	spooling system.	lpadmin(1M)
output. printf, fprintf,	sprintf: print formatted	printf(3S)
integer data in a/	sputl, sgetl: access long	sputl(3X)
power,/ exp, log, log10, pow,	sqrt: exponential, logarithm,	exp(3M)
exponential, logarithm, power,	square root functions. /sqrt:	exp(3M)
generator. rand,	srand: simple random-number	rand(3C)
/nrand48, mrand48, jrand48,	srand48, seed48, lcong48:/	drand48(3C)
input. scanf, fscanf,	sscanf: convert formatted	scanf(3S)
signals.	ssignal, gsignal: software	ssignal(3C)
package. stdio:	standard buffered input/output	stdio(3S)
communication/ stdipc, ftok:	standard interprocess	stdipc(3C)
sh, rsh: shell, the	standard/restricted command/	sh(1)
half-inch tape.	stape: SCSI quarter-inch and	stape(7)
and output. rsterm: manually	start and stop terminal input	rsterm(1M)
rfstart:	start Remote File Sharing.	rfstart(1M)
operating system for/	starter: information about the	starter(1)
and/ lpsched, lpshut, lpmove:	start/stop the LP scheduler	lpsched(1M)
/prtacct, runacct, shutacct,	startup, turnacct: shell/	acctsh(1M)
	stat, fstat: get file status.	stat(2)
useful with graphical/	stat: statistical network	stat(1G)
stat: data returned by	stat system call.	stat(5)

system information.	statfs, fstatfs: get file	statfs(2)
with graphical/ stat:	statistical network useful	stat(1G)
ff: file name and	statistics for a file system.	ff(1M)
nfsstat: Network File System	statistics.	nfsstat(1M)
display serial port error	statistics. serstat:	serstat(1M)
ustat: get file system	statistics.	ustat(2)
fsstat: report file system	status.	fsstat(1M)
/extract error records and	status information from dump.	errdead(1M)
lpstat: print LP	status information.	lpstat(1)
feof, clearerr, fileno: stream	status inquiries. ferror,	ferror(3S)
control. uustat: uucp	status inquiry and job	uustat(1C)
communication facilities	status. /report inter-process	ipcs(1)
netstat: show network	status.	netstat(1)
network. ruptime: display	status of nodes on local	ruptime(1)
ps: report process	status.	ps(1)
rwhod: host	status server.	rwhod(1M)
stat, fstat: get file	status.	stat(2)
input/output package.	stdio: standard buffered	stdio(3S)
interprocess communication/	stdipc, ftok: standard	stdipc(3C)
	stime: set time.	stime(2)
wait for child process to	stop or terminate. wait:	wait(2)
rsterm: manually start and	stop terminal input and/	rsterm(1M)
rc0: run commands performed to	stop the operating system.	rc0(1M)
environment. rfstop:	stop the Remote File Sharing	rfstop(1M)
nextkey:/ dbminit, fetch,	store, delete, firstkey,	dbm(3X)
messages.	strace: print STREAMS trace	strace(1M)
strcmp, strncmp,/ string:	strcat, strdup, strncat,	string(3C)
/strcpy, strncpy, strlen,	strchr, strrchr, strpbrk,/	string(3C)
cleanup program.	strclean: STREAMS error logger	strclean(1M)
/strcat, strdup, strncat,	strcmp, strncmp, strcpy,/	string(3C)
/strncat, strcmp, strncmp,	strcpy, strncpy, strlen,/	string(3C)
/strrchr, strpbrk, strspn,	strcspn, strtok: string/	string(3C)
strncmp,/ string: strcat,	strdup, strncat, strcmp,	string(3C)
sed:	stream editor.	sed(1)
fflush: close or flush a	stream. fclose,	fclose(3S)
fopen, freopen, fdopen: open a	stream.	fopen(3S)
reposition a file pointer in a	stream. fseek, rewind, ftell:	fseek(3S)
get character or word from a	stream. /getchar, fgetc, getw:	getc(3S)
getmsg: get next message off a	stream.	getmsg(2)
fgets: get a string from a	stream. gets,	gets(3S)
put character or word on a	stream. /putchar, fputc, putw:	putc(3S)
putmsg: send a message on a	stream.	putmsg(2)
puts, fputs: put a string on a	stream.	puts(3S)
setvbuf: assign buffering to a	stream. setbuf,	setbuf(3S)
/feof, clearerr, fileno:	stream status inquiries.	ferror(3S)
/routines for returning a	stream to a remote command.	rcmd(3)
rexec: return	stream to a remote command.	rexec(3)
push character back into input	stream. ungetc:	ungetc(3S)
commands.	streamio: STREAMS ioctl	streamio(7)
open any minor device on a	STREAMS driver. clone:	clone(7)
program. strclean:	STREAMS error logger cleanup	strclean(1M)
strerr:	STREAMS error logger daemon.	strerr(1M)
event/ log: interface to	STREAMS error logging and	log(7)
multiplexing. poll:	STREAMS input/output	poll(2)
streamio:	STREAMS ioctl commands.	streamio(7)
slink, ldsocket:	STREAMS linker, load socket/	slink(1)
Interface cooperating	STREAMS module. /Transport	timod(7)
Interface read/write interface	STREAMS module. /Transport	tirdwr(7)

sxt:	STREAMS multiplexor.	sxt(7)
strace: print	STREAMS trace messages.	strace(1M)
daemon.	strerr: STREAMS error logger	strerr(1M)
long integer and base-64 ASCII	string. /l64a: convert between	a64l(3C)
convert date and time to	string. /ascftime, tzset:	ctime(3C)
floating-point number to	string. /fcvt, gcvt: convert	ecvt(3C)
search a file for a character	string. fgrep:	fgrep(1)
gps: graphical primitive	string, format of graphical/	gps(4)
gets, fgets: get a	string from a stream.	gets(3S)
puts, fputs: put a	string on a stream.	puts(3S)
bcmp, bzero: bit and byte	string operations. bcopy,	bstring(3)
strspn, strcspn, strtok:	string operations. /strpbrk,	string(3C)
number. strtod, atof: convert	string to double-precision	strtod(3C)
strtol, atol, atoi: convert	string to integer.	strtol(3C)
cftime: language specific	strings.	cftime(4)
text strings in a file.	strings: extract the ASCII	strings(1)
extract the ASCII text	strings in a file. strings:	strings(1)
xstr: extract and share	strings in C programs.	xstr(1)
number information from a/	strip: strip symbol and line	strip(1)
information from a/ strip:	strip symbol and line number	strip(1)
/strncmp, strcpy, strncpy,	strlen, strchr, strrchr,/	string(3C)
string: strcat, strdup,	strncat, strcmp, strncmp,/	string(3C)
/strdup, strncat, strcmp,	strncmp, strcpy, strncpy,/	string(3C)
/strcmp, strncmp, strcpy,	strncpy, strlen, strchr,/	string(3C)
/strlen, strchr, strrchr,	strpbrk, strspn, strcspn,/	string(3C)
/strncpy, strlen, strchr,	strrchr, strpbrk, strspn,/	string(3C)
/strchr, strrchr, strpbrk,	strspn, strcspn, strtok:/	string(3C)
to double-precision number.	strtod, atof: convert string	strtod(3C)
/strpbrk, strspn, strcspn,	strtok: string operations.	string(3C)
string to integer.	strtol, atol, atoi: convert	strtol(3C)
processes using a file or file	structure. fuser: identify	fuser(1M)
t_alloc: allocate a library	structure.	t_alloc(3n)
t_free: free a library	structure.	t_free(3n)
terminal.	stty: set the options for a	stty(1)
another user.	su: become super-user or	su(1M)
firstkey, nextkey: database	subroutines. /store, delete,	dbm(3X)
dbm_clearerr: database	subroutines. /dbm_error,	ndbm(3X)
plot: graphics interface	subroutines.	plot(3X)
/same lines of several files or	subsequent lines of one file.	paste(1)
count of a file.	sum: print checksum and block	sum(1)
du:	summarize disk usage.	du(1M)
accounting/ acctcms: command	summary from per-process	acctcms(1M)
base. rpc:	Sun rpc program number data	rpc(4)
sync: update the	super block.	sync(1M)
sync: update	super block.	sync(2)
inetd: internet	''super-server''.	inetd(1M)
/file for inetd (internet	''super-server'').	inetd.conf(4)
su: become	super-user or another user.	su(1M)
interval. sleep:	suspend execution for an	sleep(1)
interval. sleep:	suspend execution for	sleep(3C)
pause:	suspend process until signal.	pause(2)
	swab: swap bytes.	swab(3C)
swap:	swap administrative interface.	swap(1M)
swab:	swap bytes.	swab(3C)
interface.	swap: swap administrative	swap(1M)
Protocol control/ slipd:	switched Serial Line Internet	slipd(1M)
file.	swrite: synchronous write on a	swrite(2)
	sxt: STREAMS multiplexor.	sxt(7)

information from/ strip: strip	symbol and line number	strip(1)
file/ ldgetname: retrieve	symbol name for common object	ldgetname(3X)
name for common object file	symbol table entry. /symbol	ldgetname(3X)
object/ /compute the index of a	symbol table entry of a common	ldtbindex(3X)
ldtbread: read an indexed	symbol table entry of a common/	ldtbread(3X)
syms: common object file	symbol table format.	syms(4)
object/ ldtbseek: seek to the	symbol table of a common	ldtbseek(3X)
unistd: file header for	symbolic constants.	unistd(4)
sdb:	symbolic debugger.	sdb(1)
common CTIX system terms and	symbols. /definitions of	glossary(1)
mkdbsym: load	symbols in kernel debugger.	mkdbsym(1M)
symbol table format.	syms: common object file	syms(4)
	sync: update super block.	sync(2)
	sync: update the super block.	sync(1M)
/correct the time to allow	synchronization of the system/	adjtime(2)
update: provide disk	synchronization.	update(1M)
t_sync:	synchronize transport library.	t_sync(3n)
select:	synchronous I/O multiplexing.	select(2)
swrite:	synchronous write on a file.	swrite(2)
interpreter) with C-like	syntax. csh: a shell (command	csh(1)
definition.	sysdef: output system	sysdef(1M)
error/ perror, errno,	sys_errlist, sys_nerr: system	perror(3C)
information.	sysfs: get file system type	sysfs(2)
requests.	syslocal: special system	syslocal(2)
perror, errno, sys_errlist,	sys_nerr: system error/	perror(3C)
shutdown, halt: shut down	system, change system state.	shutdown(1M)
binary search a sorted	table. bsearch:	bsearch(3C)
for common object file symbol	table entry. /symbol name	ldgetname(3X)
/compute the index of a symbol	table entry of a common object/	ldtbindex(3X)
file. /read an indexed symbol	table entry of a common object	ldtbread(3X)
common object file symbol	table format. syms:	syms(4)
master device information	table. master:	master(4)
mnttab: mounted file system	table.	mnttab(4)
ldtbseek: seek to the symbol	table of a common object file.	ldtbseek(3X)
/dtoc, ttoc, vtoc: graphical	table of contents routines.	toc(1G)
remotely mounted file system	table. rmtab:	rmtab(4)
I/O Processor configuration	table. rtab: Remote	rtab(4)
setmnt: establish mount	table.	setmnt(1M)
getdtablesize: get descriptor	table size.	getdtablesize(2)
classification and conversion	tables. /generate character	chrtbl(1M)
tbl: format	tables for nroff or troff.	tbl(1)
hdestroy: manage hash search	tables. hsearch, hcreate,	hsearch(3C)
manipulate the routing	tables. route: manually	route(1M)
tabs: set	tabs on a terminal.	tabs(1)
expand, unexpand: expand	tabs to spaces, and vice/	expand(1)
request.	t_accept: accept a connect	t_accept(3n)
ctags: create a	tags file.	ctags(1)
a file.	tail: deliver the last part of	tail(1)
talk:	talk to another user.	talk(1)
communication server.	talkd: remote user	talkd(1M)
structure.	t_alloc: allocate a library	t_alloc(3n)
trigonometric/ trig: sin, cos,	tan, asin, acos, atan, atan2:	trig(3M)
sinh, cosh,	tanh: hyperbolic functions.	sinh(3M)
V/TAPE 3200 half-inch	tape controller. /Interphase	ipt(7)
set drive parameters for	tape controllers. tapeset:	tapeset(1M)
information used/ tapedrives:	tape drive specific	tapedrives(4)
tsioctl: facilitate usage of a	tape drive.	tsioctl(1)
Hewlett-Packard 2645A terminal	tape file archiver. hpio:	hpio(1)

tabs: set tabs on a	terminal.	tabs(1)
hpio: Hewlett-Packard 2645A	terminal tape file archiver.	hpio(1)
and terminal/ tset: set	terminal, terminal interface,	tset(1)
system/ conlocate: locate a	terminal to use as the virtual	conlocate(1M)
tty: get the name of the	terminal.	tty(1)
isatty: find name of a	terminal. ttyname,	ttyname(3C)
and line/ getty: set	terminal type, modes, speed,	getty(1M)
and line/ uugetty: set	terminal type, modes, speed,	uugetty(1M)
number. ttytype: list of	terminal types by terminal	ttytype(4)
vt: virtual	terminal.	vt(7)
functions of DASI 300 and 300s	terminals. /handle special	300(1)
functions of Hewlett-Packard	terminals. hp: handle special	hp(1)
channels. tp: controlling	terminal' s local RS-232	tp(7)
term: conventional names for	terminals.	term(5)
kill:	terminate a process.	kill(1)
exit, _exit:	terminate process.	exit(2)
demon. errstop:	terminate the error-logging	errstop(1M)
for child process to stop or	terminate. wait: wait	wait(2)
tic:	terminfo compiler.	tic(1M)
initialize a terminal or query	terminfo database. tput:	tput(1)
a termcap description into a	terminfo description. /convert	captoinfo(1M)
infocmp: compare or print out	terminfo descriptions.	infocmp(1M)
data base.	terminfo: terminal capability	terminfo(4)
interface.	termio: general terminal	termio(7)
/of common CTIX system	terms and symbols.	glossary(1)
message.	t_error: produce error	t_error(3n)
command.	test: condition evaluation	test(1)
isnan: isnand, isnanf:	test for floating point NaN/	isnan(3C)
quiz:	test your knowledge.	quiz(6)
ed, red:	text editor.	ed(1)
ex:	text editor.	ex(1)
casual users). edit:	text editor (variant of ex for	edit(1)
change the format of a	text file. newform:	newform(1)
fspec: format specification in	text files.	fspec(4)
/checkeq: format mathematical	text for nroff or troff.	eqn(1)
prepare constant-width	text for troff. cw, checkcw:	cw(1)
ms:	text formatting macros.	ms(5)
nroff: format	text.	nroff(1)
plock: lock process,	text, or data in memory.	plock(2)
more, page:	text perusal.	more(1)
strings: extract the ASCII	text strings in a file.	strings(1)
troff: typeset	text.	troff(1)
binary search trees. tsearch,	tfind, tdelete, twalk: manage	tsearch(3C)
structure.	t_free: free a library	t_free(3n)
user interface to the DARPA	TFTP protocol. tftp:	tftp(1)
DARPA TFTP protocol.	tftp: user interface to the	tftp(1)
Transfer Protocol server.	tftpd: DARPA Trivial File	tftpd(1M)
tgetstr, tgoto, tputs:/	tgetent, tgetnum, tgetflag,	otermcap(3X)
tputs:/ tgetent, tgetnum,	tgetflag, tgetstr, tgoto,	otermcap(3X)
protocol-specific service/	t_getinfo: get	t_getinfo(3n)
tgoto, tputs:/ tgetent,	tgetnum, tgetflag, tgetstr,	otermcap(3X)
state.	t_getstate: get the current	t_getstate(3)
tgetent, tgetnum, tgetflag,	tgetstr, tgoto, tputs:/	otermcap(3X)
/tgetnum, tgetflag, tgetstr,	tgoto, tputs: terminal/	otermcap(3X)
	tic: terminfo compiler.	tic(1M)
ttt, cubic:	tic-tac-toe.	ttt(6)
data and system/ timex:	time a command; report process	timex(1)
time:	time a command.	time(1)

execute commands at a later	time. at, batch:	at(1)
a C shell environment at login	time. cprofile: setting up	cprofile(4)
systems for optimal access	time. dcopy: copy file	dcopy(1M)
	time: get time.	time(2)
settimeofday: get/set date and	time. gettimeofday,	gettimeofday(2)
profil: execution	time profile.	profil(2)
up an environment at login	time. profile: setting	profile(4)
stime: set	time.	stime(2)
time: get	time.	time(2)
of the/ adjtime: correct the	time to allow synchronization	adjtime(2)
tzset: convert date and	time to string. /ascftime,	ctime(3C)
clock: report CPU	time used.	clock(3C)
timezone: set default system	time zone.	timezone(4)
process times.	times: get process and child	times(2)
update access and modification	times of a file. touch:	touch(1)
get process and child process	times. times:	times(2)
file access and modification	times. utime: set	utime(2)
process data and system/	timex: time a command; report	timex(1)
time zone.	timezone: set default system	timezone(4)
cooperating STREAMS module.	timod: Transport Interface	timod(7)
	tio: tape io filter.	tio(1)
interface.	tiop: terminal accelerator	tiop(7)
read/write interface STREAMS/	tirdwr: Transport Interface	tirdwr(7)
request.	t_listen: listen for a connect	t_listen(3n)
event on a transport/	t_look: look at the current	t_look(3n)
file.	tmpfile: create a temporary	tmpfile(3S)
for a temporary file.	tmpnam, tempnam: create a name	tmpnam(3S)
/isascii, tolower, toupper,	toascci, _tolower, _toupper,/	ctype(3C)
/tolower, _toupper, _tolower,	toascii: translate characters.	conv(3C)
graphical table of contents/	toc: dtoc, ttoc, vtoc:	toc(1G)
popen, pclose: initiate pipe	to/from a process.	popen(3S)
/toupper, tolower, _toupper,	_tolower, toascii: translate/	conv(3C)
tolower, toupper, toascci,	_tolower, _toupper,/ /isascii,	ctype(3C)
toascii:/ conv: toupper,	tolower, _toupper, _tolower,	conv(3C)
compare shared libraries	tool. chkshlib:	chkshlib(1)
endpoint.	t_open: establish a transport	t_open(3n)
tsort:	topological sort.	tsort(1)
a transport endpoint.	t_optmgmt: manage options for	t_optmgmt(3n)
acctmerg: merge or add	total accounting files.	acctmerg(1M)
modification times of a file.	touch: update access and	touch(1)
/toupper, toascci, _tolower,	_toupper, setchrclass:/	ctype(3C)
conv: toupper, tolower,	_toupper, _tolower, toascii:/	conv(3C)
local RS-232 channels.	tp: controlling terminal's	tp(7)
	tplot: graphics filters.	tplot(1G)
query terminfo database.	tput: initialize a terminal or	tput(1)
/tgetflag, tgetstr, tgoto,	tputs: terminal independent/	otermcap(3X)
	tr: translate characters.	tr(1)
strace: print STREAMS	trace messages.	strace(1M)
ptrace: process	trace.	ptrace(2)
error logging and event	tracing. /interface to STREAMS	log(7)
ftp: ARPANET file	transfer program.	ftp(1)
ftpd: DARPA Internet File	Transfer Protocol server.	ftpd(1M)
tftpd: DARPA Trivial File	Transfer Protocol server.	tftpd(1M)
/_toupper, _tolower, toascii:	translate characters.	conv(3C)
tr:	translate characters.	tr(1)
tcp: Internet	Transmission Control Protocol.	tcp(7)
t_bind: bind an address to a	transport endpoint.	t_bind(3n)
t_close: close a	transport endpoint.	t_close(3n)

look at the current event on a	transport endpoint. t_look:	t_look(3n)
t_open: establish a	transport endpoint.	t_open(3n)
/manage options for a	transport endpoint.	t_optmgmt(3n)
t_unbind: disable a	transport endpoint.	t_unbind(3n)
cooperating STREAMS/ timod:	Transport Interface	timod(7)
interface STREAMS/ tirdwr:	Transport Interface read/write	tirdwr(7)
t_sync: synchronize	transport library.	t_sync(3n)
system. uucico: file	transport program for the uucp	uucico(1M)
nlsprovider: get name of	transport provider.	nlsprovider(3n)
a connection with another	transport user. /establish	t_connect(3n)
expedited data sent over a/	t_rcv: receive data or	t_rcv(3n)
confirmation from a connect/	t_rcvconnect: receive the	t_rcvconnect(3)
from disconnect.	t_rcvdis: retrieve information	t_rcvdis(3n)
of an orderly release/	t_rcvrel: acknowledge receipt	t_rcvrel(3n)
unit.	t_rcvudata: receive a data	t_rcvudata(3)
data error indication.	t_rcvuderr: receive a unit	t_rcvuderr(3)
ftw: walk a file	tree.	ftw(3C)
twalk: manage binary search	trees. /tfind, tdelete,	tsearch(3C)
trk:	trekkie game.	trk(6)
tan, asin, acos, atan, atan2:	trigonometric functions. /cos,	trig(3M)
server. tftpd: DARPA	Trivial File Transfer Protocol	tftpd(1M)
	trk: trekkie game.	trk(6)
constant-width text for	troff. cw, checkcw: prepare	cw(1)
mathematical text for nroff or	troff. /neqn, checkeq: format	eqn(1)
typesetting view graphs/ mv: a	troff macro package for	mv(5)
format tables for nroff or	troff. tbl:	tbl(1)
	troff: typeset text.	troff(1)
true, false: provide	truth values.	true(1)
with debugging on. Uutry:	try to contact a remote system	Uutry(1M)
twalk: manage binary search/	tsearch, tfind, tdelete,	tsearch(3C)
interface, and terminal/	tset: set terminal, terminal	tset(1)
tape drive.	tsioctl: facilitate usage of a	tsioctl(1)
data over a connection.	t_snd: send data or expedited	t_snd(3n)
disconnect request.	t_snddis: send user-initiated	t_snddis(3n)
release.	t_sndrel: initiate an orderly	t_sndrel(3n)
	t_sndudata: send a data unit.	t_sndudata(3)
	tsort: topological sort.	tsort(1)
library.	t_sync: synchronize transport	t_sync(3n)
contents routines. toc: dtoc,	ttoc, vtoc: graphical table of	toc(1G)
	ttt, cubic: tic-tac-toe.	ttt(6)
interface.	tty: controlling terminal	tty(7)
terminal.	tty: get the name of the	tty(1)
a terminal.	ttyname, isatty: find name of	ttyname(3C)
utmp file of the current/	ttyslot: find the slot in the	ttyslot(3C)
types by terminal number.	ttytype: list of terminal	ttytype(4)
endpoint.	t_unbind: disable a transport	t_unbind(3n)
/runacct, shutacct, startup,	turnacct: shell procedures for/	acctsh(1M)
tsearch, tfind, tdelete,	twalk: manage binary search/	tsearch(3C)
file: determine file	type.	file(1)
sysfs: get file system	type information.	sysfs(2)
getty: set terminal	type, modes, speed, and line/	getty(1M)
uugetty: set terminal	type, modes, speed, and line/	uugetty(1M)
ttytype: list of terminal	types by terminal number.	ttytype(4)
nodes for assorted device	types. /create device	createdev(1M)
types.	types: primitive system data	types(5)
types: primitive system data	types.	types(5)
session. script: make	typescript of terminal	script(1)
graphs, and slides. mmt, mvt:	typeset documents, view	mmt(1)

troff:	typeset text.	troff(1)
mv: a troff macro package for	typesetting view graphs and/	mv(5)
to/ /asctime, cftime, ascftime,	tzset: convert date and time	ctime(3C)
control.	uadmin: administrative	uadmin(1M)
control.	uadmin: administrative	uadmin(2)
system.	uconf: configure the operating	uconf(1M)
Protocol.	udp: Internet User Datagram	udp(7)
getpw: get name from	UID.	getpw(3C)
	ul: do underlining.	ul(1)
/endspent, fgetspent, lckpwdf,	ulckpwdf: get shadow.	getspent(3X)
limits.	ulimit: get and set user	ulimit(2)
creation mask.	umask: set and get file	umask(2)
mask.	umask: set file-creation mode	umask(1)
systems and remote/ mount,	umount: mount and unmount file	mount(1M)
	umount: unmount a file system.	umount(2)
multiple file/ mountall,	umountall: mount, unmount	mountall(1M)
File Sharing resource.	unadv: unadvertise a Remote	unadv(1M)
Sharing resource. unadv:	unadvertise a Remote File	unadv(1M)
CTIX system.	uname: get name of current	uname(2)
CTIX system.	uname: print name of current	uname(1)
ul: do	underlining.	ul(1)
file. unget:	undo a previous get of an SCCS	unget(1)
spaces, and vice/ expand,	unexpand: expand tabs to	expand(1)
an SCCS file.	unget: undo a previous get of	unget(1)
into input stream.	ungetc: push character back	ungetc(3S)
/seed48, lcong48: generate	uniformly distributed/	drand48(3C)
a file.	uniq: report repeated lines in	uniq(1)
mktemp: make a	unique file name.	mktemp(3C)
gethostid, sethostid: get/set	unique identifier of current/	gethostid(2)
symbolic constants.	unistd: file header for	unistd(4)
t_rcvuderr: receive a	unit data error indication.	t_rcvuderr(3)
t_rcvudata: receive a data	unit.	t_rcvudata(3)
t_sndudata: send a data	unit.	t_sndudata(3)
	units: conversion program.	units(1)
mc68k, miti, mini, mega,	unixpc,. machid:	machid(1)
execution. uux:	UNIX-to-UNIX system command	uux(1C)
uucp, uulog, uuname:	UNIX-to-UNIX system copy.	uucp(1C)
uuto, uupick: public	UNIX-to-UNIX system file copy.	uuto(1C)
link, unlink: link and	unlink files and directories.	link(1M)
entry.	unlink: remove directory	unlink(2)
umount:	unmount a file system.	umount(2)
mount, umount: mount and	unmount file systems and/	mount(1M)
mountall, umountall: mount,	unmount multiple file systems.	mountall(1M)
nmountall, numountall: mount,	unmount Network File System/	nmountall(1M)
resource. fumount: forced	unmount of an advertised	fumount(1M)
rmountall, rumountall: mount,	unmount Remote File Sharing/	rmountall(1M)
manage notifications. notify,	unnotify, evwait, evnowait:	notify(2)
files. pack, pcat,	unpack: compress and expand	pack(1)
times of a file. touch:	update access and modification	touch(1)
of programs. make: maintain,	update, and regenerate groups	make(1)
pwconv: install and	update /etc/shadow with/	pwconv(1M)
pwunconv: install and	update /etc/shadow with/	pwunconv(1M)
lfind: linear search and	update. lsearch,	lsearch(3C)
synchronization.	update: provide disk	update(1M)
sync:	update super block.	sync(2)
masterupd:	update the master file.	masterupd(1M)
sync:	update the super block.	sync(1M)
du: summarize disk	usage.	du(1M)

a command description and	usage examples. /retrieve	usage(1)
tsioctl: facilitate	usage of a tape drive.	tsioctl(1)
description and usage/	usage: retrieve a command	usage(1)
stat: statistical network	useful with graphical/	stat(1G)
id: print	user and group IDs and names.	id(1M)
setuid, setgid: set	user and group IDs.	setuid(2)
idload: Remote File Sharing	user and group mapping.	idload(1M)
talkd: remote	user communication server.	talkd(1M)
crontab:	user crontab file.	crontab(1)
character login name of the	user. cuserid: get	cuserid(3S)
udp: Internet	User Datagram Protocol.	udp(7)
/getgid, getegid: get real	user, effective user, real/	getuid(2)
environ:	user environment.	environ(5)
disk accounting data by	user ID. diskusg: generate	diskusg(1M)
program. finger:	user information lookup	finger(1)
fingerd: remote	user information server.	fingerd(1M)
protocol. telnet:	user interface to TELNET	telnet(1)
TFTP protocol. tftp:	user interface to the DARPA	tftp(1)
ulimit: get and set	user limits.	ulimit(2)
logname: return login name of	user.	logname(3X)
/get real user, effective	user, real group, and/	getuid(2)
become super-user or another	user. su:	su(1M)
talk: talk to another	user.	talk(1)
with another transport	user. /establish a connection	t_connect(3n)
the utmp file of the current	user. /find the slot in	ttyslot(3C)
write: write to another	user.	write(1)
request. t_snddis: send	user-initiated disconnect	t_snddis(3n)
(variant of ex for casual	users). edit: text editor	edit(1)
mail, rmail: send mail to	users or read mail.	mail(1)
rhosts: remote equivalent	users.	rhosts(4)
operating system for beginning	users. /information about the	starter(1)
wall: write to all	users.	wall(1)
fuser: identify processes	using a file or file/	fuser(1M)
search a file for a pattern	using full regular/ egrep:	egrep(1)
identify a CTIX system command	using keywords. locate:	locate(1)
assist: assistance	using CTIX system commands.	assist(1)
/install and verify software	using the mkfs(1) proto file/	qinstall(1)
failed login attempts.	/usr/adm/loginlog: log of	loginlog(4)
statistics.	ustat: get file system	ustat(2)
gutil: graphical	utilities.	gutil(1G)
modification times.	utime: set file access and	utime(2)
utmp, wtmp:	utmp and wtmp entry formats.	utmp(4)
endutent, utmpname: access	utmp file entry. /setutent,	getut(3C)
ttyslot: find the slot in the	utmp file of the current user.	ttyslot(3C)
/pututline, setutent, endutent,	utmpname: access utmp file/	getut(3C)
directories and permissions/	uucheck: check the uucp	uucheck(1M)
for the uucp system.	uucico: file transport program	uucico(1M)
directory clean-up.	uucleanup: uucp spool	uucleanup(1M)
/configuration file for	uucp communications lines.	Devices(5)
uucheck: check the	uucp directories and/	uucheck(1M)
uucpd, ouucpd: network	uucp servers.	uucpd(1M)
uucleanup:	uucp spool directory clean-up.	uucleanup(1M)
control. uustat:	uucp status inquiry and job	uustat(1C)
file transport program for the	uucp system. uucico:	uucico(1M)
uusched: the scheduler for the	UUCP system.	uusched(1M)
UNIX-to-UNIX system copy.	uucp, uulog, uuname:	uucp(1C)
servers.	uucpd, ouucpd: network uucp	uucpd(1M)
modes, speed, and line/	uugetty: set terminal type,	uugetty(1M)

	whodo: who is doing what.	whodo(1M)
fold long lines for finite	width output device. fold:	fold(1)
window:	window management primitives.	window(7)
wm:	window management.	wm(1)
primitives.	window: window management	window(7)
	wm: window management.	wm(1)
cd: change	working directory.	cd(1)
chdir: change	working directory.	chdir(2)
get path-name of current	working directory. getcwd:	getcwd(3C)
pwd:	working directory name.	pwd(1)
swrite: synchronous	write on a file.	swrite(2)
write:	write on a file.	write(2)
putpwent:	write password file entry.	putpwent(3C)
entry. putspent:	write shadow password file	putspent(3X)
wall:	write to all users.	wall(1)
write:	write to another user.	write(1)
	write: write on a file.	write(2)
open: open for reading or	writing.	open(2)
utmp, wtmp: utmp and	wtmp entry formats.	utmp(4)
accounting records. fwtmp,	wtmpfix: manipulate connect	fwtmp(1M)
hunt-the-wumpus.	wump: the game of	wump(6)
list(s) and execute command.	xargs: construct argument	xargs(1)
strings in C programs.	xstr: extract and share	xstr(1)
bessel: j0, j1, jn,	y0, y1, yn: Bessel functions.	bessel(3M)
bessel: j0, j1, jn, y0,	y1, yn: Bessel functions.	bessel(3M)
compiler-compiler.	yacc: yet another	yacc(1)
bessel: j0, j1, jn, y0, y1,	yn: Bessel functions.	bessel(3M)
set default system time	zone. timezone:	timezone(4)

NAME

intro - introduction to system calls and error numbers

SYNOPSIS

#include <errno.h>

DESCRIPTION

This section describes the system calls, most of which have one or more error returns. An error condition is indicated by an otherwise impossible returned value. This is almost always -1 or the NULL pointer; the individual descriptions specify the details. An error number is also made available in the external variable *errno*. *Errno* is not cleared on successful calls, so it should be tested only after an error has been indicated.

Each system call description attempts to list all possible error numbers. The following is a complete list of the error numbers and their names as defined in **<errno.h>**.

1 **EPERM Not owner or super-user**

Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or super-user. It is also returned for attempts by ordinary users to do things allowed only to the super-user.

2 **ENOENT No such file or directory**

This error occurs when a file name is specified and the file should exist but doesn't, or when one of the directories in a path name does not exist.

3 **ESRCH No such process**

No process can be found corresponding to that specified by *pid* in *kill*(2) or *ptrace*(2).

4 **EINTR Interrupted system call**

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

5 **EIO I/O error**

Some physical I/O error has occurred. This error may in some cases occur on a call following the one to which it actually applies.

6 **ENXIO No such device or address**

I/O on a special file refers to a subdevice which does not exist, or beyond the limits of the device. It may also occur when, for example, a tape drive is not on-line or no disk pack is loaded on a drive.

7 **E2BIG Arg list too long**

An argument list longer than 5,120 bytes is presented to a member of the *exec*(2) family.

8 **ENOEXEC Exec format error**

A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid magic number [see *a.out*(4)].

9 **EBADF Bad file number**

Either a file descriptor refers to no open file, or a *read*(2) [*write*(2)] request is made to a file which is open only for writing (respectively, reading).

10 **ECHILD No child processes**

A *wait* was executed by a process that had no existing or unwaited-for child processes.

11 **EAGAIN No more processes**

A *fork* failed because the system's process table is full or the user is not allowed to create any more processes, a system call failed because of insufficient memory or swap space, or an IPC call is made with the IPC_NOWAIT and the caller would block.

12 **ENOMEM Not enough space**

During an *exec*(2), *brk*(2), or *sbrk*(2), a program asks for more space than the system is able to supply. This may not be 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*(2). If this error occurs on a resource associated with Remote File Sharing (RFS), it indicates a memory depletion which may be temporary, dependent on system activity at the time the call was invoked.

13 **EACCES Permission denied**

An attempt was made to access a file or an IPC structure in a way forbidden by the protection system.

14 **EFAULT Bad address**

The system encountered a hardware fault in attempting to use an argument of a system call.

15 **ENOTBLK Block device required**

A non-block file was mentioned where a block device was required: for example, in *mount*(2).

16 **EBUSY Device or resource 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 will also occur if an attempt is made to enable accounting when it is already enabled. The device or resource is currently unavailable.

17 **EEXIST File exists**

An existing file was mentioned in an inappropriate context: for example, *link*(2).

18 **EXDEV Cross-device link**

A link to a file on another device was attempted.

19 **ENODEV No such device**

An attempt was made to apply an inappropriate system call to a device: for example, read a write-only device.

20 **ENOTDIR Not a directory**

A non-directory was specified where a directory is required, for example in a path prefix or as an argument to *chdir*(2).

21 **EISDIR Is a directory**

An attempt was made to write on a directory.

22 **EINVAL Invalid argument**

Some invalid argument [for example, dismounting a non-mounted device; mentioning an undefined signal in *signal*(2) or *kill*(2); reading or writing a file for which *lseek*(2) has generated a negative pointer]. Also set by the math functions described in the (3M) entries of this manual.

23 **ENFILE File table overflow**

The system file table is full, and temporarily no more *opens* can be accepted.

24 **EMFILE Too many open files**

No process may have more than NOFILES (default 20) descriptors open at a time. When a record lock is being created with *fcntl*, there are too many files with record locks on them.

25 **ENOTTY Not a character device (or) Not a typewriter**

An attempt was made to *ioctl*(2) a file that is not a special character device.

26 **ETXTBSY Text file busy**

An attempt was made to execute a pure-procedure program that is currently open for writing. Also an attempt to open for writing or to remove a pure-procedure program that is being executed.

27 **EFBIG File too large**

The size of a file exceeded the maximum file size or ULIMIT [see *ulimit*(2)].

28 **ENOSPC No space left on device**

During a *write*(2) to an ordinary file, there is no free space left on the device. In an IPC call, no IPC identifiers are available.

29 **ESPIPE Illegal seek**

An *lseek*(2) was issued to a pipe.

30 **EROFS Read-only file system**

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

31 **EMLINK Too many links**

An attempt to make more than the maximum number of links (1000) to a file.

32 **EPIPE Broken pipe**

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

33 **EDOM Math argument**

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

34 **ERANGE Result too large**

The value of a function in the math package (3M) is not representable within machine precision.

35 **ENOMSG No message of desired type**

An attempt was made to receive a message of a type that does not exist on the specified message queue [see *msgop*(2)].

36 **EIDRM Identifier removed**

This error is returned to processes that resume execution due to the removal of an identifier from the file system's name space [see *msgctl*(2), *semctl*(2), and *shmctl*(2)].

37 **ECHRNG Channel number out of range**

Not used; retained for compatibility.

38 **EL2NSYNC Level 2 not synchronized**

Not used; retained for compatibility.

39 **EL3HALT Level 3 halted**

Not used; retained for compatibility.

40 **EL3RST Level 3 reset**

Not used; retained for compatibility.

41 **ELNRNG Link number out of range**

Not used; retained for compatibility.

42 **EVNATCH Protocol driver not attached**

Not used; retained for compatibility.

43 **ENOCSI No CSI structure available**

Not used; retained for compatibility.

44 **EL2HLT Level 2 halted**

Not used; retained for compatibility.

45 **EDEADLK Deadlock**

A deadlock situation was detected and avoided. This error pertains to file and record locking provided by *fcntl* (2).

46 **ENOLCK No lock**

In *fcntl*(2), the setting or removing of record locks on a file cannot be accomplished because there are no more record entries left on the system.

50 **EBADE Invalid exchange**

A user-specified exchange descriptor is out of range or specifies an unallocated exchange.

51 **EBADR Invalid request descriptor**

An attempt has been made to reference a request that is not outstanding.

52 **EXFULL Exchange full**

No request descriptors are currently available for this exchange.

53 **ENOANO No anode**

Not used; retained for compatibility.

54 **EBADRQC Invalid request code**

No routing is currently available for this request code.

55 **EBADSLT Invalid slot**

The slot number specified for an ICC request is not present in the system. (No longer used; retained for compatibility.)

56 **EDEADLOCK Deadlock error**

Call cannot be honored because of potential deadlock or because lock table is full. [Note that this return value is associated with *locking*(2) and differs from the **EDEADLK** of *fcntl*(2); see the WARNING on *locking*(2).]

57 **EBFONT Bad font file format**

Not used; retained for compatibility.

60 **ENOSTR Not a stream**

A *putmsg*(2) or *getmsg*(2) system call was attempted on a file descriptor that is not a STREAMS device.

62 **ETIME Stream ioctl timeout**

The timer set for a STREAMS *ioctl*(2) call has expired. The cause of this error is device specific and could indicate either a hardware or software failure, or perhaps a timeout value that is too short for the specific operation. The status of the *ioctl*(2) operation is indeterminate.

63 **ENOSR No stream resources**

During a STREAMS *open*(2), either no STREAMS queues or no STREAMS head data structures were available.

64 **ENONET Machine is not on the network**

This error is Remote File Sharing (RFS) specific. It occurs when users try to advertise, unadvertise, mount, or unmount remote resources while the machine has not done the proper startup to connect to the network.

65 **ENOPKG No package**

This error occurs when users attempt to use a system call from a package which has not been installed.

66 **EREMOTE Resource is remote**

This error is RFS specific. It occurs when users try to advertise a resource which is not on the local machine, or try to mount/unmount a device (or pathname) that is on a remote machine.

67 **ENOLINK Virtual circuit is gone**

This error is RFS specific. It occurs when the link (virtual circuit) connecting to a remote machine is gone.

68 **EADV Advertise error**

This error is RFS specific. It occurs when users try to advertise a resource which 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.

69 **ESRMNT Srmount error**

This error is RFS specific. It occurs when users try to stop RFS while there are resources still mounted by remote machines.

70 **ECOMM Communication error**

This error is RFS specific. It occurs when trying to send messages to remote machines but no virtual circuit can be found.

71 **EPROTO Protocol error**

Some protocol error occurred. This error is device specific, but is generally not related to a hardware failure.

74 **EMULTIHOP Multihop attempted**

This error is RFS specific. It occurs when users try to access remote resources which are not directly accessible.

77 **EBADMSG Bad message**

During a *read*(2), *getmsg*(2), or *ioctl*(2) I_RECVFD system call to a STREAMS device, something has come to the head of the queue that can't be processed. That something depends on the system call:

read(2)	Control information or a passed file descriptor.
getmsg(2)	Passed file descriptor.
ioctl(2)	Control or data information.

83 **ELIBACC Cannot access a needed shared library**

Trying to *exec*(2) an *a.out* that requires a shared library (to be linked in) and the shared library doesn't exist or the user doesn't have permission to use it.

84 **ELIBBAD Accessing a corrupted shared library**

Trying to *exec*(2) an *a.out* that requires a shared library (to be linked in) and *exec*(2) could not load the shared library. The shared library is probably corrupted.

85 **ELIBSCN .lib section in a.out corrupted**

Trying to *exec*(2) an *a.out* that requires a shared library (to be linked in) and there was erroneous data in the .lib section of the *a.out*. The .lib section tells *exec*(2) what shared libraries are needed. The *a.out* is probably corrupted.

86 **ELIBMAX Attempting to link in more shared libraries than system limit**

Trying to *exec*(2) an *a.out* that requires more shared libraries (to be linked in) than is allowed on the current configuration of the system. See the *S/Series*CTIX*Administrator's*Guide.

87 **ELIBEXEC Cannot exec a shared library directly**

Trying to *exec*(2) a shared library directly. This is not allowed.

224 **ENOHDW No hardware available for operation**

The address specification exceeds the allowable limits or the required hardware does not exist (for example, the executable file requires hardware that is not available). See *exec* (2).

225 **EBADFS Bit-mapped file system is marked dirty**

An attempt to mount a bit-mapped file system failed due to the dirty flag being set for that file system.

226 **EWOULDBLOCK Operation would block**

An operation which would cause a process to block was attempted on an object in non-blocking mode.

227 **EINPROGRESS Operation now in progress**

An operation which takes a long time to complete [such as a *connect*(2)] was attempted on a non-blocking object.

228 **EALREADY Operation already in progress**

An operation was attempted on a non-blocking object which already had an operation in progress.

229 **ENOTSOCK Socket operation on non-socket**

Self-explanatory.

230 **EDESTADDRREQ Destination address required**

A required address was omitted from an operation on a socket.

231 EMSGSIZE **Message too long**

A message sent on a socket was larger than the internal message buffer.

232 EPROTOTYPE **Protocol wrong type for socket**

A protocol was specified which does not support the semantics of the socket type requested. For example, you cannot use the ARPA Internet UDP protocol with type SOCK_STREAM.

233 EPROTONOSUPPORT **Protocol not supported**

The protocol has not been configured into the system or no implementation for it exists.

234 ESOCKTNOSUPPORT **Socket type not supported**

The support for the socket type has not been configured into the system or no implementation for it exists.

235 EOPNOTSUPP **Operation not supported on socket**

For example, trying to *accept* a connection on a datagram socket.

236 EPFNOSUPPORT **Protocol family not supported**

The protocol family has not been configured into the system or no implementation for it exists.

237 EAFNOSUPPORT **Address family not supported by protocol**

An address incompatible with the requested protocol was used. For example, you shouldn't necessarily expect to be able to use PUP Internet addresses with ARPA Internet protocols.

238 EADDRINUSE **Address already in use**

Only one usage of each address is normally permitted.

239 EADDRNOTAVAIL **Can't assign requested address**

Normally results from an attempt to create a socket with an address not on this machine.

240 ENETDOWN **Network is down**

A socket operation encountered a dead network.

241 ENETUNREACH **Network is unreachable**

A socket operation was attempted to an unreachable network.

242 **ENETRESET Network dropped connection on reset**

The host you were connected to crashed and rebooted.

243 **ECONNABORTED Software caused connection abort**

A connection abort was caused internal to your host machine.

244 **ECONNRESET Connection reset by peer**

A connection was forcibly closed by a peer. This normally results from the peer executing a *shutdown* (2) call.

245 **ENOBUFS No buffer space available**

An operation on a socket or pipe was not performed because the system lacked sufficient buffer space.

246 **EISCONN Socket is already connected**

A *connect* request was made on an already connected socket; or, a *sendto* or *sendmsg* request on a connected socket specified a destination other than the connected party.

247 **ENOTCONN Socket is not connected**

An request to send or receive data was disallowed because the socket is not connected.

248 **ESHUTDOWN Can't send after socket shutdown**

A request to send data was disallowed because the socket had already been shut down with a previous *shutdown*(2) call.

249 **ETOOMANYREFS Too many references: can't splice**

Not in use; included for compatibility only.

250 **ETIMEDOUT Connection timed out**

A *connect* request failed because the connected party did not properly respond after a period of time. (The timeout period is dependent on the communication protocol.)

251 **ECONNREFUSED Connection refused**

No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service which is inactive on the foreign host.

252 **EHOSTDOWN Host is down**

The host is down.

253 **EHOSTUNREACH No route to host**

The gateway does not recognize the requested host via the route specified.

254 **ENOPROTOOPT Protocol not available**

A bad option was specified in a *getsockopt*(2) or *setsockopt*(2) call.

DEFINITIONS

Process ID Each active process in the system is uniquely identified by a positive integer called a process ID. The range of this ID is from 1 to 30,000.

Parent Process ID A new process is created by a currently active process [see *fork*(2)]. The parent process ID of a process is the process ID of its creator.

Process Group ID Each active process is a member of a process group that is identified by a positive integer called the process group ID. This ID is the process ID of the group leader. This grouping permits the signaling of related processes [see *kill*(2)].

Tty Group ID Each active process can be a member of a terminal group that is identified by a positive integer called the tty group ID. This grouping is used to terminate a group of related processes upon termination of one of the processes in the group [see *exit*(2) and *signal*(2)].

Real User ID and Real Group ID Each user allowed on the system is identified by a positive integer (0 to 65535) called a real user ID.

Each user is also a member of a group. The group is identified by a positive integer called the real group ID.

An active process has a real user ID and real group ID that are set to the real user ID and real group ID, respectively, of the user responsible for the creation of the process.

Effective User ID and Effective Group ID An active process has an effective user ID and an effective group ID that are used to determine file access permissions (see below). The effective user ID and effective group ID are equal to the process's real user ID and real group ID, respectively, unless the process or one of its ancestors evolved from a file that had the set-user-ID bit or set-group- ID bit set [see *exec*(2)].

Super-user A process is recognized as a *super-user* process and is granted special privileges, such as immunity from file permissions, if its effective user ID is 0.

Special Processes The processes with a process ID of 0 and a process ID of 1 are special processes and are referred to as *proc0* and *proc1*.

Proc0 is the scheduler. *Proc1* is the initialization process (*init*). Proc1 is the ancestor of every other process in the system and is used to control the process structure.

File Descriptor A file descriptor is a small integer used to do I/O on a file. The value of a file descriptor is from 0 to (NOFILES - 1). A process may have no more than NOFILES file descriptors open simultaneously. A file descriptor is returned by system calls such as *open*(2), or *pipe*(2). The file descriptor is used as an argument by calls such as *read*(2), *write*(2), *ioctl*(2), and *close*(2).

File Name Names consisting of 1 to 14 characters may be used to name an ordinary file, special file or directory.

These characters may be selected from the set of all character values excluding \0 (null) and the ASCII code for / (slash).

Note that it is generally unwise to use *, ?, [, or] as part of file names because of the special meaning attached to these characters by the shell [see *sh*(1)]. Although permitted, the use of unprintable characters in file names should be avoided.

Path Name and Path Prefix A path name is a null-terminated character string starting with an optional slash (/), followed by zero or more directory names separated by slashes, optionally followed by a file name.

If a path name begins with a slash, the path search begins at the *root* directory. Otherwise, the search begins from the current working directory.

A slash by itself names the root directory.

Unless specifically stated otherwise, the null path name is treated as if it named a non-existent file.

Directory Directory entries are called links. By convention, a directory contains at least two links, . and .., referred to as *dot* and *dot-dot*, respectively. Dot refers to the directory itself and dot-dot refers to its parent directory.

Root Directory and Current Working Directory Each process has associated with it a concept of a root directory and a current working directory for the purpose of resolving path name searches. The root directory of a process need not be the root directory of the root file system.

File Access Permissions Read, write, and execute/search permissions on a file are granted to a process if one or more of the following are true:

- The effective user ID of the process is super-user.
- The effective user ID of the process matches the user ID of the owner of the file and the appropriate access bit of the "owner" portion (0700) of the file mode is set.
- The effective user ID of the process does not match the user ID of the owner of the file, and the effective group ID of the process matches the group of the file and the appropriate access bit of the "group" portion (0070) of the file mode is set.
- The effective user ID of the process does not match the user ID of the owner of the file, and the effective group ID of the process does not match the group ID of the file, and the appropriate access bit of the "other" portion (0007) of the file mode is set.

Otherwise, the corresponding permissions are denied.

Message Queue Identifier A message queue identifier (msqid) is a unique positive integer created by a *msgget*(2) system call. Each msqid has a message queue and a data structure associated with it. The data structure is referred to as *msqid_ds* and contains the following members:

```
struct   ipc_perm msg_perm;
struct   msg *msg_first;
struct   msg *msg_last;
ushort   msg_cbytes;
ushort   msg_qnum;
ushort   msg_qbytes;
ushort   msg_lspid;
ushort   msg_lrpid;
time_t   msg_stime;
time_t   msg_rtime;
time_t   msg_ctime;
```

msg_perm Is an **ipc_perm** structure that specifies the message operation permission (see below). This structure includes the following members:

```
ushort   cuid;  /* creator user id */
ushort   cgid;  /* creator group id */
ushort   uid;   /* user id */
ushort   gid;   /* group id */
ushort   mode;  /* r/w permission */
ushort   seq;   /* slot usage sequence # */
key_t    key;   /* key */
```

msg *msg_first	Is a pointer to the first message on the queue.
msg *msg_last	Is a pointer to the last message on the queue.
msg_cbytes	Is the current number of bytes on the queue.
msg_qnum	Is the number of messages currently on the queue.
msg_qbytes	Is the maximum number of bytes allowed on the queue.
msg_lspid	Is the process ID of the last process that performed a *msgsnd* operation.
msg_lrpid	Is the process ID of the last process that performed a *msgrcv* operation.
msg_stime	Is the time of the last *msgsnd* operation.
msg_rtime	Is the time of the last *msgrcv* operation.
msg_ctime	Is the time of the last *msgctl*(2) operation that changed a member of the above structure.

Message Operation Permissions In the *msgop*(2) and *msgctl*(2) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed, interpreted as follows:

00400	Read by user
00200	Write by user
00040	Read by group
00020	Write by group
00004	Read by others
00002	Write by others

Read and write permissions on a msqid are granted to a process if one or more of the following are true:

- The effective user ID of the process is super-user.
- The effective user ID of the process matches **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with *msqid* and the appropriate bit of the ''user'' portion (0600) of **msg_perm.mode** is set.

- The effective group ID of the process matches **msg_perm.cgid** or **msg_perm.gid** and the appropriate bit of the "group" portion (060) of **msg_perm.mode** is set.
- The appropriate bit of the "other" portion (006) of **msg_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

Semaphore Identifier A semaphore identifier (semid) is a unique positive integer created by a *semget*(2) system call. Each semid has a set of semaphores and a data structure associated with it. The data structure is referred to as *semid_ds* and contains the following members:

```
struct   ipc_perm sem_perm;    /* operation permission*/
                               /* struct */
struct   sem *sem_base;        /* ptr to first semaphore in set */
ushort   sem_nsems;            /* number of sems in set */
time_t   sem_otime;            /* last operation time */
time_t   sem_ctime;            /* last change time */
                               /* Times measured in secs since */
                               /* 00:00:00 GMT, Jan. 1, 1970 */
```

sem_perm Is an **ipc_perm** structure that specifies the semaphore operation permission (see below). This structure includes the following members:

```
ushort   uid;     /* user id */
ushort   gid;     /* group id */
ushort   cuid;    /* creator user id */
ushort   cgid;    /* creator group id */
ushort   mode;    /* r/a permission */
ushort   seq;     /* slot usage sequence number */
key_t    key;     /* key */
```

sem_nsems Is equal to the number of semaphores in the set. Each semaphore in the set is referenced by a positive integer referred to as a *sem_num*. Sem_num values run sequentially from 0 to the value of sem_nsems minus 1.

sem_otime Is the time of the last *semop*(2) operation.

sem_ctime Is the time of the last *semctl*(2) operation that changed a member of the above structure.

A semaphore is a data structure called *sem* that contains the following members:

```
ushort  semval;    /* semaphore value */
short   sempid;    /* pid of last operation */
ushort  semncnt;   /* # awaiting semval > cval */
ushort  semzcnt;   /* # awaiting semval = 0 */
```

semval	Is a non-negative integer which is the actual value of the semphore.
sempid	Is equal to the process ID of the last process that performed a semaphore operation on this semaphore.
semncnt	Is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become greater than its current value.
semzcnt	Is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become zero.

Semaphore Operation Permissions In the *semop*(2) and *semctl*(2) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed interpreted as follows:

00400	Read by user
00200	Alter by user
00040	Read by group
00020	Alter by group
00004	Read by others
00002	Alter by others

Read and alter permissions on a semid are granted to a process if one or more of the following are true:

- The effective user ID of the process is super-user.
- The effective user ID of the process matches **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with *semid* and the appropriate bit of the "user" portion (0600) of **sem_perm.mode** is set.
- The effective group ID of the process matches **sem_perm.cgid** or **sem_perm.gid** and the appropriate bit of the "group" portion (060) of **sem_perm.mode** is set.
- The appropriate bit of the "other" portion (006) of **sem_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

Shared-Memory Identifier A shared-memory identifier (shmid) is a unique positive integer created by a *shmget*(2) system call. Each shmid has a segment of memory (referred to as a shared memory segment) and a data structure associated with it. (Note that these shared memory segments must be explicitly removed by the user after the last reference to them is removed.) The data structure is referred to as *shmid_ds* and contains the following members:

```
struct  ipc_perm shm_perm;     /* operation permission */
                               /*struct */
int     shm_segsz;             /* size of segment */
ushort  shm_lpid;              /* pid of last operation */
ushort  shm_cpid;              /* creator pid */
ushort  shm_nattch;            /* number of current attaches */
time_t  shm_atime;             /* last attach time */
time_t  shm_dtime;             /* last detach time */
time_t  shm_ctime;             /* last change time */
                               /* Times measured in secs since */
                               /* 00:00:00 GMT, Jan. 1, 1970 */
```

shm_perm Is an **ipc_perm** structure that specifies the shared memory operation permission (see below). This structure includes the following members:

```
ushort  cuid;   /* creator user id */
ushort  cgid;   /* creator group id */
ushort  uid;    /* user id */
ushort  gid;    /* group id */
ushort  mode;   /* r/w permission */
ushort  seq;    /* slot usage sequence number */
key_t   key;    /* key */
```

shm_segsz Specifies the size of the shared memory segment in bytes.

shm_cpid Is the process ID of the process that created the shared memory identifier.

shm_lpid Is the process ID of the last process that performed a *shmop*(2) operation.

shm_nattch Is the number of processes that currently have this segment attached.

shm_atime	Is the time of the last *shmat*(2) operation.
shm_dtime	Is the time of the last *shmdt*(2) operation.
shm_ctime	Is the time of the last *shmctl*(2) operation that changed one of the members of the above structure.

Shared-Memory Operation Permissions In the *shmop*(2) and *shmctl*(2) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed interpreted as follows:

00400	Read by user
00200	Write by user
00040	Read by group
00020	Write by group
00004	Read by others
00002	Write by others

Read and write permissions on a shmid are granted to a process if one or more of the following are true:

- The effective user ID of the process is super-user.
- The effective user ID of the process matches **shm_perm.cuid** or **shm_perm.uid** in the data structure associated with *shmid* and the appropriate bit of the ''user'' portion (0600) of **shm_perm.mode** is set.
- The effective group ID of the process matches **shm_perm.cgid** or **shm_perm.gid** and the appropriate bit of the ''group'' portion (060) of **shm_perm.mode** is set.
- The appropriate bit of the ''other'' portion (06) of **shm_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

STREAMS A set of kernel mechanisms that support the development of network services and data communication *drivers*. It defines interface standards for character input/output within the kernel and between the kernel and user level processes. The STREAMS mechanism is composed of utility routines, kernel facilities and a set of data structures.

Stream A stream is a full-duplex data path within the kernel between a user process and driver routines. The primary components are a *stream head*, a *driver* and zero or more *modules* between the *stream head* and *driver*. A *stream* is analogous to a shell pipeline except that data flow and processing are bidirectional.

Stream Head In a *stream*, the *stream head* is the end of the *stream* that provides the interface between the *stream* and a user process. The principle functions of the *stream head* are processing STREAMS-related system calls, and passing data and information between a user process and the *stream*.

Driver In a *stream*, the *driver* provides the interface between peripheral hardware and the *stream*. A *driver* can also be a pseudo-*driver*, such as a *multiplexor* or log *driver* [see *log*(7)], which is not associated with a hardware device.

Module A module is an entity containing processing routines for input and output data. It always exists in the middle of a *stream*, between the stream's head and a *driver*. A *module* is the STREAMS counterpart to the commands in a Shell pipeline except that a module contains a pair of functions which allow independent bidirectional (*downstream* and *upstream*) data flow and processing.

Downstream In a *stream*, the direction from *stream head* to *driver*.

Upstream In a *stream*, the direction from *driver* to *stream head*.

Message In a *stream*, one or more blocks of data or information, with associated STREAMS control structures. *Messages* can be of several defined types, which identify the *message* contents. *Messages* are the only means of transferring data and communicating within a *stream*.

Message Queue In a *stream*, a linked list of *messages* awaiting processing by a *module* or *driver*.

Read Queue In a *stream*, the *message queue* in a *module* or *driver* containing *messages* moving *upstream*.

Write Queue In a *stream*, the *message queue* in a *module* or *driver* containing *messages* moving *downstream*.

Multiplexor A multiplexor is a driver that allows *streams* associated with several user processes to be connected to a single *driver*, or several *drivers* to be connected to a single user process. STREAMS does not provide a general multiplexing *driver*, but does provide the facilities for constructing them, and for connecting multiplexed configurations of *streams*.

Sockets and Address Families

A socket is an endpoint for communication between processes. Each socket has queues for sending and receiving data.

Sockets are typed according to their communications properties. These properties include whether messages sent and received at a socket require the name of the partner, whether communication is reliable, the format used in naming message recipients, etc.

Each instance of the system supports some collection of socket types; consult *socket*(2) for more information about the types available and their properties.

Each instance of the system supports some number of sets of communications protocols. Each protocol set supports addresses of a certain format. An Address Family is the set of addresses for a specific group of protocols. Each socket has an address chosen from the address family in which the socket was created.

Two interchangeable structures are used by socket calls: *sockaddr* (defined in **<sys/socket.h>**) and *sockaddr_in* (defined in **<sys/in.h>**). The *sa_data* field of the *sockaddr* structure is interpreted according to the address family. (Note that AF_INET is the only currently supported address family.) The structure *sockaddr_in* has been defined specifically for the Internet family (the first field must be AF_INET); this structure is described in *inet*(7).

SEE ALSO

close(2), exit(2), getmsg(2), getpid(2), getuid(2), msgctl(2), msgget(2), msgop(2), open(2), poll(2), putmsg(2), read(2), semctl(2), semget(2), semop(2), setpgrp(2), setuid(2), shmctl(2), shmget(2), shmop(2), signal(2), wait(2), write(2), intro(3).
CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.
UNIX System V Release 3.2 Streams Programmer's Guide.
UNIX System V Release 3.2 Streams Primer.

NAME

access - determine accessibility of a file

SYNOPSIS

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

DESCRIPTION

The *path* argument points to a path name naming a file; *access* checks the named file for accessibility according to the bit pattern contained in *amode*, using the real user ID in place of the effective user ID and the real group ID in place of the effective group ID. The bit pattern contained in *amode* is constructed as follows:

04	read
02	write
01	execute (search)
00	check existence of file

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

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	Read, write, or execute (search) permission is requested for a null path name.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[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.
[EACCES]	Permission bits of the file mode do not permit the requested access.
[EFAULT]	*Path* points outside the allocated address space for the process.
[EINTR]	A signal was caught during the *access* 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.

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(2), stat(2).

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.

NAME

acct - enable or disable process accounting

SYNOPSIS

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 will be written on an accounting file for each process that terminates. Termination can be caused by one of two things: an *exit* call or a signal [see *exit*(2) and *signal*(2)]. The effective user ID of the calling process must be super-user to use this call.

path points to a pathname naming the accounting file. The accounting file format is given in *acct*(4).

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

acct will fail if one or more of the following are true:

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

SEE ALSO

exit(2), signal(2), acct(4).

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.

NAME

adjtime - correct the time to allow synchronization of the system clock

SYNOPSIS

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

int adjtime(delta, olddelta)
struct timeval *delta;
struct timeval *olddelta;
```

DESCRIPTION

The *adjtime* call makes small adjustments to the system time, as returned by *gettimeofday*(2), advancing or retarding it by the time specified by the timeval *delta*. If *delta* is negative, the clock is slowed down by incrementing it more slowly than normal until the correction is complete. If *delta* is positive, a larger increment than normal is used. The skew used to perform the correction is generally a fraction of one percent. Thus, the time is always a monotonically increasing function.

This call can be used by time servers that synchronize the clocks of computers in a local area network. Such time servers would slow down the clocks of some machines and speed up the clocks of others to bring them to the average network time.

RETURN VALUE

A return value of 0 indicates that the call succeeded. A return value of -1 indicates that an error occurred, and in this case an error code is stored in the global variable *errno*.

ERRORS

The following error codes may be set in *errno*:

[EFAULT]	An argument points outside the process's allocated address space.
[EPERM]	The process's effective user ID is not that of the super-user.

SEE ALSO

date(1), gettimeofday(2).

WARNINGS

A time correction from an earlier call to *adjtime* may not be finished when *adjtime* is called again. If *olddelta* is non-zero, then the structure pointed to will contain, upon return, the number of microseconds still to be corrected from the earlier call.

The *adjtime*(2) call is restricted to the super-user.

NAME

alarm - set a process alarm clock

SYNOPSIS

unsigned alarm (sec)
unsigned sec;

DESCRIPTION

alarm instructs the alarm clock of the calling process to send the signal **SIGALRM** to the calling process after the number of real time seconds specified by *sec* have elapsed [see *signal*(2)].

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

If *sec* is 0, any previously made alarm request is canceled.

SEE ALSO

pause(2), signal(2), sigpause(2).

DIAGNOSTICS

alarm returns the amount of time previously remaining in the alarm clock of the calling process.

NAME

bind - bind a name to a socket

SYNOPSIS

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

int bind (s, name, namelen)
int s;
struct sockaddr *name;
int namelen;

DESCRIPTION

The *bind* call assigns a name to an unnamed socket. When a socket is created with *socket*(2), it exists in a name space (address family) but has no name assigned. (Currently, only the Internet address family is supported.) The *bind* call requests that *name* be assigned to the socket.

SEE ALSO

connect(2), getsockname(2), intro(2), listen(2), socket(2), inet(7), intro(7).
CTIX Network Programmer's Primer.

NOTES

The rules used in name binding vary between communication domains [see *protocols*(4)]. Consult the manual entries in Section 7 for detailed information.

RETURN VALUE

If the bind is successful, a 0 value is returned. A return value of -1 indicates an error, which is further specified in the global *errno.*

ERRORS

The *bind* call fails if any of the following are true:

[EBADF]	*S* is not a valid descriptor.
[ENOTSOCK]	*S* is not a socket.
[EADDRNOTAVAIL]	The specified address is not available from the local machine.
[EADDRINUSE]	The specified address is already in use.
[EINVAL]	The socket is already bound to an address.
[EACCESS]	The requested address is protected, and the current user has inadequate permission to access it.
[EFAULT]	The *name* parameter is not in a valid part of the user address space.

NAME

brk, sbrk - change data segment space allocation

SYNOPSIS

int brk (endds)
char *endds;

char *sbrk (incr)
int incr;

DESCRIPTION

brk and *sbrk* are used to change dynamically the amount of space allocated for the calling process's data segment [see *exec*(2)]. The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases. Newly allocated space is set to zero. If, however, the same memory space is reallocated to the same process its contents are undefined.

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

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

brk and *sbrk* will fail without making any change in the allocated space if one or more of the following are true:

[ENOMEM]	Such a change would result in more space being allocated than is allowed by the system-imposed maximum process size [see *ulimit*(2)].
[EAGAIN]	Total amount of system memory available for a read during physical I/O is temporarily insufficient [see *shmop*(2)]. This may occur even though the space requested was less than the system-imposed maximum process size [see *ulimit*(2)].

SEE ALSO

exec(2), shmop(2), ulimit(2), end(3C).

DIAGNOSTICS

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.

NAME

chdir - change working directory

SYNOPSIS

int chdir (path)
char *path;

DESCRIPTION

Path points to the path name of a directory. *chdir* causes the named directory to become the current working directory, the starting point for path searches for path names not beginning with /.

chdir will fail and the current working directory will be unchanged if one or more of the following are true:

[ENOTDIR]	A component of the path name is not a directory.
[ENOENT]	The named directory does not exist.
[EACCES]	Search permission is denied for any component of the path name.
[EFAULT]	*Path* points outside the allocated address space of the process.
[EINTR]	A signal was caught during the *chdir* 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.

SEE ALSO

chroot(2).

DIAGNOSTICS

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

NAME

chmod - change mode of file

SYNOPSIS

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

DESCRIPTION

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

Access permission bits are interpreted as follows:

04000	Set user ID on execution.
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, 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 does 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 does not delete the program text from memory when the last user process terminates. In either case, if the sticky bit is set the text is 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 be removed only if one or more of the following is true [see *unlink(2)*]:

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 exists on a regular file. This can affect future calls to open(2), creat(2), read(2), and write(2) on this file.

The *chmod* fails and the file mode is unchanged if one or more of the following are 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.

SEE ALSO

chmod(1), chown(2), creat(2), fcntl(2), mknod(2), open(2), read(2), write(2).

DIAGNOSTICS

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

NAME

chown - change owner and group of a file

SYNOPSIS

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

DESCRIPTION

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

Only processes with effective user ID equal to the file owner or super-user may change the ownership of a file.

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

The *chown* call fails and the owner and group of the named file remains unchanged if one or more of the following are 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 *chown* 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.

SEE ALSO

chown(1), chmod(2).

DIAGNOSTICS

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

NAME

chroot - change root directory

SYNOPSIS

int chroot (path)
char *path;

DESCRIPTION

The *path* argument points to a path name naming a directory. The *chroot* call causes the named directory to become the root directory, the starting point for path searches for path names beginning with **root** (/). 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* call fails and the root directory remains unchanged if one or more of the following are true:

[ENOTDIR]	Any component of the path name is not a directory.
[ENOENT]	The named directory does not exist.
[EPERM]	The effective user ID is not super-user.
[EFAULT]	*Path* points outside the allocated address space of the process.
[EINTR]	A signal was caught during the *chroot* 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.

SEE ALSO

chdir(2).

DIAGNOSTICS

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

NAME

close - close a file descriptor

SYNOPSIS

int close (fildes)
int fildes;

DESCRIPTION

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

If a STREAMS [see *intro*(2)] file is closed, and the calling process had previously registered to receive a SIGPOLL signal [see *signal*(2) and *sigset*(2)] for events associated with that file [see I_SETSIG in *streamio*(7)], the calling process will be unregistered for events associated with the file. The last *close* for a *stream* causes the *stream* associated with *fildes* to be dismantled. If O_NDELAY 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_NDELAY 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 are true:

[EBADF]	*fildes* 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.

SEE ALSO

creat(2), dup(2), exec(2), fcntl(2), intro(2), open(2), pipe(2), signal(2), sigset(2), streamio(7).

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.

NAME

connect - initiate a connection on a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int connect (s, name, namelen)
int s;
struct sockaddr *name;
int namelen;
```

DESCRIPTION

The *connect* call initiates a connection on a socket. The parameter *s* is a socket. If it is of type SOCK_DGRAM, then this call permanently specifies the peer to which datagrams are to be sent; if it is of type SOCK_STREAM, then this call attempts to make a connection to another socket. The other socket is specified by *name*; *namelen* is the length of *name*, which is an address in the address family of the socket. Each address family interprets the *name* parameter in its own way.

RETURN VALUE

If the connection or binding succeeds, then 0 is returned. Otherwise a -1 is returned, and a more specific error code is stored in *errno*.

ERRORS

The call fails if:

[EBADF]	*S* is not a valid descriptor.
[ENOTSOCK]	*S* is a descriptor for a file, not a socket.
[EADDRNOTAVAIL]	The specified address is not available on this machine.
[EAFNOSUPPORT]	Addresses in the specified address family cannot be used with this socket.
[EISCONN]	The socket is already connected.
[ETIMEDOUT]	Connection establishment timed out without establishing a connection.
[ECONNREFUSED]	The attempt to connect was forcefully rejected.
[ENETUNREACH]	The network is not reachable from this host.
[EADDRINUSE]	The address is already in use.
[EFAULT]	The *name* parameter specifies an area outside the process address space.

SEE ALSO

accept(2), getsockname(2), intro(2), socket(2), intro(7).
CTIX Network Programmer's Primer.

NAME

creat - create a new file or rewrite an existing one

SYNOPSIS

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

DESCRIPTION

The *creat* 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*(2)].
- The ``save text image after execution bit'' of the mode is cleared [see *chmod*(2)].

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*(2)]. No process can have more than NOFILES files open simultaneously. NOFILES is a system-imposed maximum per process, which can be changed by *uconf*(1M): the range, as specified in **param.h**, is 20 (NOFILES_MIN) to 100 (NOFILES_MAX). The current value of NOFILES can be determined by *ulimit*(2). A new file can be created with a mode that forbids writing.

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

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[ENOENT]	The path name is null.
[EACCES]	The file does not exist and the directory in which the file is to be created does not permit writing.

[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.
[EACCES]	The file exists and write permission is denied.
[EISDIR]	The named file is an existing directory.
[EMFILE]	NOFILES file descriptors are currently open.
[EFAULT]	*Path* points outside the allocated address space of the process.
[ENFILE]	The system file table is full.
[EAGAIN]	The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file [see *chmod*(2)].
[EINTR]	A signal was caught during the *creat* 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.
[ENOSPC]	The file system is out of inodes.
[EDEADLOCK]	A side effect of a previous *locking*(2) call. [See the WARNING on the *locking*(2) manpage.]

SEE ALSO

chmod(2), close(2), dup(2), fcntl(2), lseek(2), open(2), read(2), umask(2), write(2).

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.

NAME

dup - duplicate an open file descriptor

SYNOPSIS

int dup (fildes)
int fildes;

DESCRIPTION

fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *dup* returns a new file descriptor having the following in common with the original:

- Same open file (or pipe).
- Same file pointer (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*(2)].

The file descriptor returned is the lowest one available.

dup will fail if one or more of the following are true:

[EBADF]	*fildes* is not a valid open file descriptor.
[EINTR]	A signal was caught during the *dup* system call.
[EMFILE]	NOFILES file descriptors are currently open.
[ENOLINK]	*fildes* is on a remote machine and the link to that machine is no longer active.

SEE ALSO

close(2), creat(2), exec(2), fcntl(2), open(2), pipe(2), lockf(3C).

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.

NAME

exec: execl, execv, execle, execve, execlp, execvp - execute a file

SYNOPSIS

```
int execl (path, arg0, arg1, ..., argn, (char *)0)
char *path, *arg0, *arg1, ..., *argn;

int execv (path, argv)
char *path, *argv[ ];

int execle (path, arg0, arg1, ..., argn, (char *)0, envp)
char *path, *arg0, *arg1, ..., *argn, *envp[ ];

int execve (path, argv, envp)
char *path, *argv[ ], *envp[ ];

int execlp (file, arg0, arg1, ..., argn, (char *)0)
char *file, *arg0, *arg1, ..., *argn;

int execvp (file, argv)
char *file, *argv[ ];
```

DESCRIPTION

The *exec* call in all its forms transforms the calling process into a new process. The new process is constructed from an ordinary, executable file called the *new process file*. This file consists of a header [see *a.out*(4)], a text segment, and a data segment. The data segment contains an initialized portion and an uninitialized portion (bss). There can be no return from a successful *exec* because the calling process is overlaid by the new process.

When a C program is executed, it is called as follows:

```
main (argc, argv, envp)
int argc;
char **argv, **envp;
```

where *argc* is the argument count, *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.

Path points to a path name that identifies the new process file.

File points to the new process file. The path prefix for this file is obtained by a search of the directories passed as the *environment* line **PATH** [see *environ*(5)]. The environment is supplied by the shell [see *sh*(1)].

arg0, *arg1*, ..., *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, at least *arg0* must be present and point to a string that is the same as *path* (or its last component).

argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process. By convention, *argv* must have at least one member, and it must point to a string that is the same as *path* (or its last component). *argv* is terminated by a null pointer.

envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process. *envp* is terminated by a null pointer. For *execl* and *execv*, the C run-time start-off routine places a pointer to the environment of the calling process in the global cell:

```
        extern char **environ;
```

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*(2). 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*(2).

For signals set by *sigset*(2), *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*(2)], *exec* sets the effective user ID of the new process to the owner ID of the new process file. Similarly, if the set-group-ID mode bit of the new process file is set, the effective group ID of the new process is set to the group ID of the new process file. The real user ID and real group ID of the new process remain the same as those of the calling process.

The shared memory segments attached to the calling process are not attached to the new process [see *shmop*(2)].

Profiling is disabled for the new process; see *profil*(2).

The new process also inherits the following attributes from the calling process:

nice value [see *nice*(2)]
process ID
parent process ID
process group ID
semadj values [see *semop*(2)]
tty group ID [see *exit*(2) and *signal*(2)]
trace flag [see *ptrace*(2) request 0]
time left until an alarm clock signal [see *alarm*(2)]
current working directory
root directory
file mode creation mask [see *umask*(2)]
file size limit [see *ulimit*(2)]
utime, *stime*, *cutime*, and *cstime* [see *times*(2)]
file-locks [see *fcntl*(2) and *lockf*(3C)]

exec fails and returns to the calling process if one or more of the following are true:

[ENOENT]	One or more components of the new process path name of the file do not exist.
[ENOTDIR]	A component of the new process path of the file prefix is not a directory.
[EACCES]	Search permission is denied for a directory listed in the new process file's path prefix.
[EACCES]	The new process file is not an ordinary file.
[EACCES]	The new process file mode denies execution permission.
[ENOEXEC]	The exec is not an *execlp* or *execvp* , and the new process file has the appropriate access permission but an invalid magic number in its header.
[ETXTBSY]	The new process file is a pure procedure (shared text) file that is currently open for writing by some process.
[ENOMEM]	The new process requires more memory than is allowed by the system-imposed maximum MAXMEM.
[E2BIG]	The number of bytes in the new process's argument list is greater than the system-imposed limit of 10,240 bytes.
[EFAULT]	*Path*, *argv*, or *envp* point to an illegal address.

[EAGAIN]	Not enough memory.
[ELIBACC]	Required shared library does not have execute permission.
[ELIBEXEC]	Trying to *exec*(2) a shared library directly.
[EINTR]	A signal was caught during the *exec* 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.
[ENOHDW]	The executable file requires hardware that does not exist (such as floating-point).
[ENOEXEC]	The file format does not correspond to that expected as specified with the magic number (such as a hole in the file).
[ENOEXEC]	The virtual address specification in the header(s) exceeds the allowed system limits.

SEE ALSO

alarm(2), exit(2), fcntl(2), fork(2), nice(2), ptrace(2), semop(2), signal(2), sigset(2), times(2), ulimit(2), umask(2), lockf(3C), a.out(4), environ(5).

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.

NAME

exit, _exit - terminate process

SYNOPSIS

void exit (status)
int status;
void _exit (status)
int status;

DESCRIPTION

The *exit* 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 (bits 0377) of *status* are made available to it [see *wait*(2)].
- If the parent process of the calling process is not executing a *wait*, the calling process is transformed into a zombie process. A *zombie process* is a process that only occupies a slot in the process table. It has no other space allocated either in user or kernel space. The process table slot that it occupies is partially overlaid with time accounting information (see <sys/**proc.h**>) to be used by *times*.
- The parent process ID of all of the calling processes' existing child processes and zombie processes is set to 1. This means the initialization process [see *intro*(2)] inherits each of these processes.
- Each attached shared memory segment is detached and the value of **shm_nattach** in the data structure associated with its shared memory identifier is decremented by 1.
- For each semaphore for which the calling process has set a semadj value [see *semop*(2)], that semadj value is added to the semval of the specified semaphore.
- If the process has a process, text, or data lock, an *unlock* is performed [see *plock*(2)].
- An accounting record is written on the accounting file if the system's accounting routine is enabled [see *acct*(2)].
- If the process ID, tty group ID, and process group ID of the calling process are equal (it is a process group leader), the **SIGHUP** signal is sent to each process that has a process group ID equal to that of the calling process.

- 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(2), intro(2), plock(2), semop(2), signal(2), sigset(2), wait(2).

DIAGNOSTICS

None. There can be no return from an *exit* system call.

WARNING

See *WARNING* in *signal*(2).

NAME

fcntl - file control

SYNOPSIS

#include <fcntl.h>

int fcntl (fildes, cmd, arg)
int fildes, cmd, arg;

DESCRIPTION

The *fcntl* call provides for control over open files. *fildes* is an open file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

The data type, value, and use of *arg* are specific to the type of command specified by *cmd*. *cmd* specifies the operation to be performed by *fcntl*, and can be one of the following:

The commands available are:

F_DUPFD	Return a new file descriptor as follows:
	Lowest numbered available file descriptor greater than or equal to *arg*.
	Same open file (or pipe) as the original file.
	Same file pointer as the original file (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* (2) system calls.
F_GETFD	Get the close-on-exec flag associated with the file descriptor *fildes*. If the low-order bit is **0** the file will remain open across *exec*, otherwise the file will be closed upon execution of *exec*.
F_SETFD	Set the close-on-exec flag associated with *fildes* to the low-order bit of *arg* (**0** or **1** as above).
F_GETFL	Get *file* status flags.
F_SETFL	Set *file* status flags to *arg*. Only certain flags can be set [see *fcntl* (5)].

F_GETLK	Get the first lock which blocks the lock description given by the variable of type *struct flock* pointed to by *arg*. The information retrieved overwrites the information passed to *fcntl* in the *flock* structure. If no lock is found that would prevent this lock from being created, then the structure is passed back unchanged except for the lock type which will be set to F_UNLCK.
F_SETLK	Set or clear a file segment lock according to the variable of type *struct flock* pointed to by *arg* [see *fcntl*(5)]. The *cmd* F_SETLK is used to establish read (F_RDLCK) and write (F_WRLCK) locks, as well as remove either type of lock (F_UNLCK). If a read or write lock cannot be set *fcntl* will return immediately with an error value of -1.
F_SETLKW	This *cmd* is the same as F_SETLK except that if a read or write lock is blocked by other locks, the process will sleep until the segment is free to be locked.

A read lock prevents any process from write locking the protected area. More than one read lock may exist for a given segment of a file at a given time. The file descriptor on which a read lock is being placed must have been opened with read access.

A write lock prevents any process from read locking or write locking the protected area. Only one write lock may exist for a given segment of a file at a given time. The file descriptor on which a write lock is being placed must have been opened with write access.

The structure *flock* describes the type (*l_type*), starting offset (*l_whence*), relative offset (*l_start*), size (*l_len*), process ID (*l_pid*), and RFS system ID (*l_sysid*) of the segment of the file to be affected. The process ID and system ID fields are used only with the F_GETLK *cmd* to return the values for a blocking lock. Locks may start and extend beyond the current end of a file, but may not be negative relative to the beginning of the file. A lock may be set to always extend to the end of 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 will be locked. Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments for either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process in a *fork*(2) system call.

When mandatory file and record locking is active on a file, [see *chmod*(2)], *read* and *write* system calls issued on the file are affected by the record locks in effect.

The *fcntl* call fails if one or more of the following are true:

[EBADF]	*fildes* is not a valid open file descriptor.
[EBADF]	*cmd* is F_SETLK or F_SETLKW the type of lock (*l_type*) is a read lock (F_RDLCK) and *fildes* is not a valid open file descriptor open for reading.
[EBADF]	*cmd* is F_SETLK or F_SETLKW the type of lock (*l_type*) is a write lock (F_RDLCK) and *fildes* is not a valid open file descriptor open for writing.
[EMFILE]	*cmd* is F_DUPFD and the number of file descriptors currently open in the calling process is the configured value for the maximum number of open file descriptors allowed each user.
[EINVAL]	*cmd* is F_DUPFD. *arg* is either negative, or greater than or equal to the configured value for the maximum number of open file descriptors allowed each user.
[EINVAL]	*cmd* is F_GETLK, F_SETLK, or SETLKW and *arg* or the data it points to is not valid.
[EACCES]	*cmd* is F_SETLK the type of lock (*l_type*) is a read (F_RDLCK) lock and the segment of a file to be locked is already write locked by another process or the type is a write (F_WRLCK) lock and the segment of a file to be locked is already read or write locked by another process.
[ENOLCK]	*cmd* is F_SETLK or F_SETLKW, the type of lock is a read or write lock, and there are no more record locks available (too many file segments locked) because the system maximum has been exceeded.
[EDEADLK]	*cmd* is F_SETLKW, the lock is blocked by some lock from another process, and putting the calling-process to sleep, waiting for that lock to become free, would cause a deadlock.
[EFAULT]	*cmd* is F_SETLK, *arg* points outside the program address space.
[EINTR]	A signal was caught during the *fcntl* system call.
[ENOLINK]	*fildes* is on a remote machine and the link to that machine is no longer active.

SEE ALSO

close(2), creat(2), dup(2), exec(2), fork(2), open(2), pipe(2), fcntl(5).

DIAGNOSTICS

Upon successful completion, the value returned depends on *cmd* as follows:

F_DUPFD	A new file descriptor.
F_GETFD	Value of flag (only the low-order bit is defined).
F_SETFD	Value other than -1.
F_GETFL	Value of file flags.
F_SETFL	Value other than -1.
F_GETLK	Value other than -1.
F_SETLK	Value other than -1.
F_SETLKW	Value other than -1.

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.

Two forms of file locking are available: *locking*(2) and *fcntl*(2). *locking*(2) is retained for compatibility with previous versions of CTIX. Although both forms are compatible and interchangeable, new programs should use only *fcntl*(2) for record locking. Note that the error return values differ.

NAME

fork - create a new process

SYNOPSIS

int fork ()

DESCRIPTION

fork causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process). This means the child process inherits the following attributes from the parent process:

> environment
> close-on-exec flag [see *exec*(2)]
> signal handling settings (that is, **SIG_DFL**, **SIG_IGN**, **SIG_HOLD**, function address)
> set-user-ID mode bit
> set-group-ID mode bit
> profiling on/off status
> nice value [see *nice*(2)]
> all attached shared memory segments [see *shmop*(2)]
> process group ID
> tty group ID [see *exit*(2)]
> current working directory
> root directory
> file mode creation mask [see *umask*(2)]
> file size limit [see *ulimit*(2)]

The child process differs from the parent process in the following ways:

- The child process has a unique process ID.
- The child process has a different parent process ID (that is, the process ID of the parent process).
- The child process has its own copy of the parent's file descriptors. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.
- All semadj values are cleared [see *semop*(2)].
- Process locks, text locks and data locks are not inherited by the child [see *plock*(2)].
- The child process's *utime*, *stime*, *cutime*, and *cstime* are set to 0. The time left until an alarm clock signal is reset to 0.

fork will fail and no child process will be created if one or more of the following are true:

[EAGAIN]	The system-imposed limit on the total number of processes under execution would be exceeded.
[EAGAIN]	The system-imposed limit on the total number of processes under execution by a single user would be exceeded.
[EAGAIN]	Total amount of system memory available when reading via raw IO is temporarily insufficient.

SEE ALSO

exec(2), nice(2), plock(2), ptrace(2), semop(2), shmop(2), signal(2), sigset(2), times(2), ulimit(2), umask(2), wait(2).

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.

NAME

getdents - read directory entries and put in a file system independent format

SYNOPSIS

```
#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*(2) or *dup*(2) system call.

The *getdents* call attempts to read *nbyte* bytes from the directory associated with *fildes* and to format them as file system independent directory entries in the buffer pointed to by *buf*. Since the file system independent directory entries are of variable length, in most cases the actual number of bytes returned is strictly less than *nbyte*.

The file system independent directory entry is specified by the *dirent* structure. For a description of this see *dirent*(4).

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 in order to implement the *readdir*(3X) routine [for a description see *directory*(3X)], and should not be used for other purposes.

The *getdents* call fails if one or more of the following are 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.
[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.
[EIO]	An I/O error occurred while accessing the file system.

SEE ALSO

directory(3X), dirent(4).

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.

NAME

getdtablesize - get descriptor table size

SYNOPSIS

nfds = getdtablesize( )
int nfds;

DESCRIPTION

Each process has a fixed size descriptor table, which is guaranteed to have at least 20 slots. The size of the descriptor table determines how many files and sockets a process can have open simultaneously. The entries in the descriptor table are numbered with small integers starting at 0. The call *getdtablesize* returns the size of this table. It is equivalent to the *ulimit*(2) system call as issued with an argument as shown below:

ulimit(4)

SEE ALSO

close(2), dup(2), open(2), select(2), ulimit(2).
CTIX Network Programmer's Primer.

NAME

gethostid, sethostid - get/set unique identifier of current host

SYNOPSIS

hostid = gethostid( )
long hostid;

sethostid(hostid)
long hostid;

DESCRIPTION

The *sethostid* call establishes a 32-bit identifier for the current system that is intended to be unique among all UNIX systems in existence. This is normally a DARPA Internet address for the local machine. This call is allowed only to the super-user and is normally performed at boot time. The *sethostid* call returns an int -1 if the ID can not be set.

The *gethostid* call returns the 32-bit identifier for the current processor.

SEE ALSO

hostid(1), gethostname(2).
CTIX Network Programmer's Primer.

NAME

gethostname, sethostname - get/set name of current host

SYNOPSIS

int gethostname(name, namelen)
char *name;
int namelen;

int sethostname(name, namelen)
char *name;
int namelen;

DESCRIPTION

The *gethostname* call returns the standard host name for the current processor, as previously set by *sethostname*. The parameter *namelen* specifies the size of the *name* array. The returned name is null-terminated unless insufficient space is provided.

The *sethostname* call sets the name of the host machine to be *name*, which has length *namelen*. This call is restricted to the super-user and is normally used only when the system is booted up. In order to maintain consistency between the system nodename and the local hostname, *sethostname* interacts with *setuname*. See *hostname*(1) for the specifics.

RETURN VALUE

If the call succeeds a value of 0 is returned. If the call fails, then a value of -1 is returned and an error code is placed in the global location *errno*.

ERRORS

The following errors may be returned by these calls:

[EFAULT]	The *name* or *namelen* parameter gave an invalid address.
[EPERM]	The caller tried to set the hostname and was not the super-user.

SEE ALSO

hostname(1), uname(1), setuname(2), gethostid(2).
CTIX Network Programmer's Primer.

WARNING

Host names are limited to MAXHOSTNAMELEN (from <*sys/param.h*>) characters, currently 64. The left-most qualifier, or nodename, is limited to the size of a system nodename, currently 9 characters. The right-most qualifier, or Internet Domain name, is limited to 54 characters.

NAME

getmsg - get next message off a stream

SYNOPSIS

```
#include <stropts.h>

int getmsg(fd, ctlptr, dataptr, flags)
int fd;
struct strbuf *ctlptr;
struct strbuf *dataptr;
int *flags;
```

DESCRIPTION

getmsg retrieves the contents of a message [see *intro*(2)] 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.

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;        /* 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.

ctlptr is used to hold the control part from the message and *dataptr* is used to hold the data part from the message. If *ctlptr* (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 *ctlptr* 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, a user may choose to retrieve only priority messages by setting *flags* to RS_HIPRI. In this case, *getmsg* will only process the next message if it is a priority message.

If O_NDELAY 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_NDELAY 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* will continue to operate normally, as described above, until the *stream head* read queue is empty. Thereafter, it will return 0 in the *len* fields of *ctlptr* and *dataptr*.

getmsg fails if one or more of the following are true:

[EAGAIN]	The O_NDELAY 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]	*ctlptr*, *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 multiplexor.
[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.

SEE ALSO

intro(2), read(2), poll(2), putmsg(2), write(2).
UNIX System V Release 3.2 Streams Primer.
UNIX System V Release 3.2 Streams Programmer's Guide.

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 will retrieve the remainder of the message.

NAME

getpeername - get name of connected peer

SYNOPSIS

int getpeername(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;

DESCRIPTION

The *getpeername* call returns the name of the peer connected to socket *s*. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes). The interpretation of *name* depends on the ''communication domain'' [see *protocols*(4)].

SEE ALSO

bind(2), getsockname(2), intro(2), socket(2), intro(7).
CTIX Network Programmer's Primer.

DIAGNOSTICS

A 0 is returned if the call succeeds, -1 if it fails.

ERRORS

The call succeeds unless one of the following is true:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOTCONN]	The socket is not connected.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *name* parameter points to memory not in a valid part of the process address space.

NAME

getpid, getpgrp, getppid - get process, process group, and parent process IDs

SYNOPSIS

int getpid ()

int getpgrp ()

int getppid ()

DESCRIPTION

The *getpid* call returns the process ID of the calling process.

The *getpgrp* call returns the process group ID of the calling process.

The *getppid* call returns the parent process ID of the calling process.

SEE ALSO

exec(2), fork(2), intro(2), setpgrp(2), signal(2).

NAME

getsockname - get socket name

SYNOPSIS

int getsockname(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;

DESCRIPTION

The *getsockname* call returns the current *name* for the specified socket *(s).* The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return *namelen* contains the actual size of the name returned (in bytes).

SEE ALSO

bind(2), intro(2), socket(2), intro(7).
CTIX Network Programmer's Primer.

RETURN VALUE

A 0 is returned if the call succeeds, -1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *name* parameter points to memory not in a valid part of the process address space.

NAME

getsockopt, setsockopt - get and set options on sockets

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int getsockopt(s, level, optname, optval, optlen)
int s, level, optname;
char *optval;
int *optlen;

int setsockopt(s, level, optname, optval, optlen)
int s, level, optname;
char *optval;
int optlen;
```

DESCRIPTION

The *getsockopt* and *setsockopt* calls manipulate *options* associated with a socket. Options can exist at multiple protocol levels; they are always present at the uppermost "socket" level.

When manipulating socket options the level at which the option resides and the name of the option must be specified. To manipulate options at the "socket" level, *level* is specified as SOL_SOCKET. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, *level* should be set to the protocol number of TCP; see *getprotoent*(3).

The parameters *optval* and *optlen* are used to access option values for *setsockopt*. For *getsockopt* they identify a buffer in which the value for the requested option(s) are to be returned. For *getsockopt*, *optlen* is a value-result parameter, initially containing the size of the buffer pointed to by *optval*, and modified on return to indicate the actual size of the value returned. If no option value is to be supplied or returned, *optval* may be supplied as 0.

optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file *<sys/socket.h>* contains definitions for "socket" level options, described below. Options at other protocol levels vary in format and name; consult the appropriate entries in section (4).

Most socket-level options take an *int* parameter for *optval*. For *setsockopt*, the parameter should be non-zero to enable a boolean option, or zero if the option is to be disabled. SO_LINGER uses a *struct linger* parameter, defined in

<*sys/socket.h*>, which specifies the desired state of the option and the linger interval (see below).

The following options are recognized at the socket level. Except as noted, each may be examined with *getsockopt* and set with *setsockopt*.

SO_DEBUG	Toggle recording of debugging information.
SO_REUSEADDR	Toggle on/off local address reuse.
SO_KEEPALIVE	Toggle keep connections alive.
SO_DONTROUTE	Toggle routing bypass for outgoing messages.
SO_LINGER	Linger on close if data present.
SO_BROADCAST	Toggle permission to transmit broadcast messages.
SO_OOBINLINE	Toggle reception of out-of-band data in band.
SO_SNDBUF	Set buffer size for output.
SO_RCVBUF	Set buffer size for input.
SO_TYPE	Get the type of the socket (get only).
SO_ERROR	Get and clear error on the socket (get only).

SO_DEBUG enables debugging in the underlying protocol modules.

SO_REUSEADDR indicates that the rules used in validating addresses supplied in a *bind*(2) call should allow reuse of local addresses.

SO_KEEPALIVE enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and processes using the socket are notified via a SIGPIPE signal.

SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER controls the action taken when unsent messags are queued on socket and a *close*(2) is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system blocks the process on the *close* attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the *setsockopt* call when SO_LINGER is requested). If SO_LINGER is disabled and a *close* is issued, the system processes the close in a manner that allows the process to continue as quickly as possible.

SO_BROADCAST requests permission to send broadcast datagrams on the socket. Broadcast was a privileged operation in earlier versions of the system.

With protocols that support out-of-band data, SO_OOBINLINE requests that out-of-band data be placed in the normal data input queue as received; it is then accessible with *recv* or *read* calls without the MSG_OOB flag.

SO_SNDBUF and SO_RCVBUF are options to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. The system places an absolute limit on these values.

SO_TYPE and SO_ERROR are options used only with *setsockopt*. SO_TYPE returns the type of the socket, such as SOCK_STREAM; it is useful for servers that inherit sockets on startup. SO_ERROR returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

SEE ALSO

ioctl(2), socket(2), getprotoent(3).
CTIX Network Programmer's Primer.

RETURN VALUE

A 0 is returned if the call succeeds, -1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOPROTOOPT]	The option is unknown at the level indicated.
[EFAULT]	The address pointed to by *optval* is not in a valid part of the process address space. For *getsockopt*, this error may also be returned if *optlen* is not in a valid part of the process address space.

BUGS

Several of the socket options should be handled at lower levels of the system.

NAME

gettimeofday, settimeofday - get/set date and time

SYNOPSIS

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

int gettimeofday(tp, tzp)
struct timeval *tp;
struct timezone *tzp;

int settimeofday(tp, tzp)
struct timeval *tp;
struct timezone *tzp;
```

DESCRIPTION

The system's notion of the current Greenwich time and the current time zone is obtained with the *gettimeofday* call, and set with the *settimeofday* call. The time is expressed in seconds and microseconds since midnight (0 hour), January 1, 1970. The resolution of the system clock is hardware dependent, and the time may be updated continuously or in "ticks." If *tzp* is zero, the time zone information will not be returned or set.

The structures pointed to by *tp* and *tzp* are defined in <*sys/time.h*> as:

```
struct timeval {
        long  tv_sec;      /* seconds since Jan. 1, 1970 */
        long  tv_usec;     /* and microseconds */
};

struct timezone {
        int   tz_minuteswest; /* of Greenwich */
        int   tz_dsttime;  /* type of dst correction to apply */
};
```

The *timezone* structure indicates 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.

Only the super-user can set the time of day or time zone.

SEE ALSO

date(1), adjtime(2), ctime(3C).

RETURN VALUE

A 0 return value indicates that the call succeeded. A -1 return value indicates an error occurred, and in this case an error code is stored into the global variable *errno*.

ERRORS

The following error codes may be set in *errno*:

[EFAULT]	An argument address referenced invalid memory.
[EPERM]	A user other than the super-user attempted to set the time.

NAME

getuid, geteuid, getgid, getegid - get real user, effective user, real group, and effective group IDs

SYNOPSIS

unsigned short getuid ()

unsigned short geteuid ()

unsigned short getgid ()

unsigned short getegid ()

DESCRIPTION

getuid returns the real user ID of the calling process.

geteuid returns the effective user ID of the calling process.

getgid returns the real group ID of the calling process.

getegid returns the effective group ID of the calling process.

SEE ALSO

intro(2), setuid(2).

NAME

ioctl - control device

SYNOPSIS

int ioctl (fildes, request, arg)
int fildes, request;

DESCRIPTION

The *ioctl* 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. This control is infrequently used on non-STREAMS devices, with the basic input/output functions performed through the *read*(2) and *write*(2) system calls.

For STREAMS files, specific functions are performed by the *ioctl* call as described in *streamio*(7).

The *fildes* argument 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 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*(7)].

The *ioctl* call fails for any type of file if one or more of the following are true:

[EBADF]	*fildes* is not a valid open file descriptor.
[ENOTTY]	*fildes* is not associated with a device driver that accepts control functions.
[EINTR]	A signal was caught during the *ioctl* system call.

The *ioctl* call also fails if the device driver detects an error; 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 fail if one or more of the following are 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.
[ENXIO]	The *request* and *arg* are valid for this device driver, but the service requested can not be performed on this particular subdevice.
[ENOLINK]	*fildes* is on a remote machine and the link to that machine is no longer active.

STREAMS errors are described in *streamio*(7).

SEE ALSO

streamio(7), termio(7).

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.

NAME

kill - send a signal to a process or a group of processes

SYNOPSIS

int kill (pid, sig)
int pid, sig;

DESCRIPTION

kill sends a signal to a process or a group of processes. The process or group of processes to which the signal is to be sent is specified by *pid*. The signal that is to be sent is specified by *sig* and is either one from the list given in *signal*(2), or 0. If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

The real or effective user ID of the sending process must match the real or effective user ID of the receiving process, unless the effective user ID of the sending process is super-user.

The processes with a process ID of 0 and a process ID of 1 are special processes [see *intro*(2)] and will be referred to below as *proc0* and *proc1*, respectively.

If *pid* is greater than zero, *sig* will be sent to the process whose process ID is equal to *pid*. *pid* may equal 1.

If *pid* is 0, *sig* will be sent to all processes excluding *proc0* and *proc1* whose process group ID is equal to the process group ID of the sender.

If *pid* is -1 and the effective user ID of the sender is not super-user, *sig* will be sent to all processes excluding *proc0* and *proc1* whose real user ID is equal to the effective user ID of the sender.

If *pid* is -1 and the effective user ID of the sender is super-user, *sig* will be sent to all processes excluding *proc0* and *proc1*.

If *pid* is negative but not -1, *sig* will be sent to all processes whose process group ID is equal to the absolute value of *pid*.

kill will fail and no signal will be sent if one or more of the following are true:

[EINVAL]	*sig* is not a valid signal number.
[EINVAL]	*sig* is SIGKILL and *pid* is 1 (proc1).
[ESRCH]	No process can be found corresponding to that specified by *pid*.
[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.

SEE ALSO

kill(1), getpid(2), setpgrp(2), signal(2), sigset(2).

DIAGNOSTICS

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

NAME

link - link to a file

SYNOPSIS

int link (path1, path2)
char *path1, *path2;

DESCRIPTION

The *path1* argument points to a path name naming an existing file; *path2* points to a path name naming the new directory entry to be created.

The *link* call creates a new link (directory entry) for the existing file. It fails and no link is created if one or more of the following are true:

[ENOTDIR]	A component of either path prefix is not a directory.
[ENOENT]	A component of either path prefix does not exist.
[EACCES]	A component of either path prefix denies search permission.
[ENOENT]	The file named by *path1* does not exist.
[EEXIST]	The link named by *path2* exists.
[EPERM]	The file named by *path1* is a directory and the effective user ID is not super-user.
[EXDEV]	The link named by *path2* and the file named by *path1* are on different logical devices (file systems).
[ENOENT]	*path2* points to a null path name.
[EACCES]	The requested link requires writing in a directory with a mode that denies write permission.
[EROFS]	The requested link requires writing in a directory on a read-only file system.
[EFAULT]	*path* points outside the allocated address space of the process.
[EMLINK]	The maximum number of links to a file would be exceeded.
[EINTR]	A signal was caught during the *link* 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.

SEE ALSO

unlink(2).

DIAGNOSTICS

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

NAME

listen - listen for connections on a socket

SYNOPSIS

int listen (s, backlog)
int s, backlog;

DESCRIPTION

To accept connections, a socket is first created with *socket*(2), a backlog for incoming connections is specified with *listen*, and then the connections are accepted with *accept*(2). The *listen* call applies only to sockets of type SOCK_STREAM.

The *backlog* parameter defines the maximum length to which the queue of pending connections may grow. If a connection request arrives with the queue full the client will receive an error with an indication of ECONNREFUSED.

SEE ALSO

accept(2), connect(2), socket(2).
CTIX Network Programmer's Primer.

RETURN VALUE

A 0 return value indicates success; -1 indicates an error.

ERRORS

The call fails if:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is not a socket.
[EOPNOTSUPP]	The socket is not of a type that supports the operation *listen*.

BUGS

The *backlog* is currently limited (silently) to 5.

NAME

locking - exclusive access to regions of a file

SYNOPSIS

int locking (filedes, mode, size);
int fildes, mode;
long size;

DESCRIPTION

The *locking* call places or removes a kernel-enforced lock on a region of a file. The calling process has exclusive access to regions it has locked. If another process uses *read*(2), *write*(2), *creat*(2), or *open*(2) (with O_TRUNC) in a way that reads or modifies part of the locked region, the second process's system call does not return until the lock is released, unless deadlock or some other error is detected. A process whose execution is suspended in such a manner is said to be *blocked*.

Parameters specify the file to be locked or unlocked, the kind of lock or unlock, and the region affected:

filedes Specifies the file to be locked or unlocked; *filedes* is a file descriptor returned by an *open*, *create*, *pipe*, *fcntl*, or *dup* system call.

mode Specifies the action: 0 for lock removal; 1 for blocking lock; 2 for checking lock. Blocking and checking locks differ only if the attempted lock is itself locked out: a blocking lock waits until the existing lock or locks are removed; a checking lock immediately returns an error.

size The region affected begins at the current file offset associated with *filedes* and is *size* bytes long. If *size* is zero, the region affected ends at the end of the file.

Locking imposes no structure on a CTIX file. A process can arbitrarily lock any unlocked byte and unlock any locked byte. However, creating a large number of noncontiguous locked regions can fill up the system's lock table and make further locks impossible. It is advisable that a program's use of *locking* segment the file in the same way as does the program's use of *read* and *write*.

A process is said to be deadlocked if it is sleeping until an unlocking which is indirectly prevented by that same sleeping process. The kernel will not permit a *read*, *write*, *creat*, *open* with O_TRUNC, or blocking *locking* if such a call would deadlock the calling process. *Errno* is set to **EDEADLOCK.** The standard response to such a situation is for the program to release all its existing locked areas and try again. If a *locking* call fails because the kernel's table of locked areas is full, again, *errno* is set to **EDEADLOCK** and, again, the calling program should release its existing locked areas.

Special files and pipes can be locked, but no input/output is blocked.

Locks are automatically removed if the process that placed the lock terminates or closes the file descriptor used to place the lock.

SEE ALSO

create(2), close(2), dup(2), open(2), read(2), write(2).

RETURN VALUE

A return value of *-1* indicates an error, with the error value in *errno*.

[EACCES]	A checking lock on a region already locked.
[EDEADLOCK]	A lock that would cause deadlock or overflow the system's lock table.

WARNING

Do not apply any standard input/output library function to a locked file: this library does not know about *locking*.

Two forms of file locking are available: *locking*(2) and *fcntl*(2). *locking*(2) is retained for compatibility with previous versions of CTIX. Although both forms are compatible and interchangeable, new programs should use only *fcntl*(2) for record locking. Note that the error return values differ.

NAME

lseek - move read/write file pointer

SYNOPSIS

long lseek (fildes, offset, whence)
int fildes;
long offset;
int whence;

DESCRIPTION

fildes is a file descriptor returned from a *creat*, *open*, *dup*, or *fcntl* system call. *lseek* sets the file pointer associated with *fildes* as follows:

- If *whence* is 0, the pointer is set to *offset* bytes.
- If *whence* is 1, the pointer is set to its current location plus *offset*.
- If *whence* is 2, the pointer is set to the size of the file plus *offset*.

Upon successful completion, the resulting pointer location, as measured in bytes from the beginning of the file, is returned. Note that if *fildes* is a remote file descriptor and *offset* is negative, *lseek* will return the file pointer even if it is negative.

lseek will fail and the file pointer will remain unchanged if one or more of the following are true:

[EBADF]	*fildes* is not an open file descriptor.
[ESPIPE]	*fildes* is associated with a pipe or fifo.
[EINVAL and SIGSYS signal]	*whence* is not 0, 1, or 2.
[EINVAL]	*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.

SEE ALSO

creat(2), dup(2), fcntl(2), open(2).

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.

NAME

mkdir - make a directory

SYNOPSIS

int mkdir (path, mode)
char *path;
int mode;

DESCRIPTION

The *mkdir* call creates a new directory with the name *path*. The mode of the new directory is initialized from the *mode*. The protection part of the *mode* argument is modified by the process's mode mask [see *umask*(2)].

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 the ''dot'' (.) and ''dot dot' (..) directories. The *mkdir* call fails and no directory is created if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.
[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.
[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.
[ENOENT]	The path is longer than the maximum allowed.
[EEXIST]	The named file already exists.
[EROFS]	The path prefix resides on a read-only file system.
[EFAULT]	*Path* points outside the allocated address space of the process.
[EMLINK]	The maximum number of links to the parent directory would be exceeded.
[EIO]	An I/O error has occurred while accessing the 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.

NAME

mknod - make a directory, or a special or ordinary file

SYNOPSIS

int mknod (path, mode, dev)
char *path;
int mode, dev;

DESCRIPTION

The *mknod* call creates a new file named by the path name pointed to by *path*. The mode of the new file is initialized from *mode*, where the value of *mode* is interpreted as follows:

0170000	file type; one of the following:
	0010000 fifo special
	0020000 character special
	0040000 directory
	0060000 block special
	0100000 or 0000000 ordinary file
0004000	set user ID on execution
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

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*(2)]. If *mode* indicates a block or character special file, *dev* is a configuration-dependent specification of a character or block I/O device. If *mode* does not indicate a block special or character special device, *dev* is ignored.

The *mknod* call can be invoked only by the super-user for file types other than FIFO special.

The call fails and the new file is not created if one or more of the following are true:

[EPERM]	The effective user ID of the process is not super-user.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.
[EROFS]	The directory in which the file is to be created is located on a read-only file system.
[EEXIST]	The named file exists.
[EFAULT]	*Path* points outside the allocated address space of the process.
[ENOSPC]	No space is available.
[EINTR]	A signal was caught during the *mknod* 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.

SEE ALSO

mkdir(1), chmod(2), exec(2), umask(2), fs(4).

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, the major and minor device numbers are interpreted by the server.

NAME

mount - mount a file system

SYNOPSIS

```
#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 file system 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 file system type number. The *sysfs*(2) system call can be used to determine the file system type number. Note that if both the MS_DATA and MS_FSS flag bits of *mflag* are off, the file system type will default to the root file system type. Only if either flag is on will *fstyp* be used to indicate the file system 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 file system type. A particular file system type may not require this data, in which case *dataptr* and *datalen* should both be zero. Note that MS_FSS is obsolete and will be 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* will refer to the root directory on the mounted file system.

The low-order bit of *mflag* is used to control write permission on the mounted file system; if 1, writing is forbidden, otherwise writing is permitted according to individual file accessibility.

mount may be invoked only by the super-user. It is intended for use only by the *mount*(1M) utility.

mount will fail if one or more of the following are true:

[EPERM]	The effective user ID is not super-user.
[ENOENT]	Any of the named files does not exist.

[ENOTDIR]	A component of a path prefix is not a directory.
[EREMOTE]	*Spec* is remote and cannot be mounted.
[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.
[ENOTBLK]	*Spec* is not a block special device.
[ENXIO]	The device associated with *spec* does not exist.
[ENOTDIR]	*Dir* is not a directory.
[EFAULT]	*Spec* or *dir* points outside the allocated address space of the process.
[EBUSY]	*Dir* is currently mounted on, is someone's current working directory, or is otherwise busy.
[EBUSY]	The device associated with *spec* is currently mounted.
[EBUSY]	There are no more mount table entries.
[EROFS]	*Spec* is write protected and *mflag* requests write permission.
[ENOSPC]	The file system state in the super-block is not FsOKAY and *mflag* requests write permission.
[EINVAL]	The super block has an invalid magic number or the *fstyp* is invalid or *mflag* is not valid.
[EBADFS]	An attempt to mount a bit-mapped file system failed due to the dirty flag being set for that file system.

SEE ALSO

mount(1M), sysfs(2), umount(2), fs(4).

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.

NAME

msgctl - message control operations

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl (msqid, cmd, buf)
int msqid, cmd;
struct msqid_ds *buf;

DESCRIPTION

The *msgctl* call provides a variety of message control operations as specified by *cmd*. The following *cmd*s are available:

IPC_STAT
: Place the current value of each member of the data structure associated with *msqid* into the structure pointed to by *buf*. The contents of this structure are defined in *intro*(2). {READ}

IPC_SET
: Set the value of the following members of the data structure associated with *msqid* to the corresponding value found in the structure pointed to by *buf*:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode /* only low 9 bits */
msg_qbytes
```

This *cmd* can be executed only 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_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 be executed only 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*.

NAME

msgget - get message queue

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget (key, msgflg)
key_t key;
int msgflg;

DESCRIPTION

The *msgget* call returns the message queue identifier associated with *key*.

A message queue identifier and associated message queue and data structure [see *intro*(2)] are created for *key* if one of the following are true:

- *Key* is equal to IPC_PRIVATE.
- *Key* does not already have a message queue identifier associated with it, and (*msgflg* & IPC_CREAT) is "true".

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

- **Msg_perm.cuid**, **msg_perm.uid**, **msg_perm.gid**, and **msg_perm.cgid** 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* call fails if one or more of the following are true:

[EACCES]	A message queue identifier exists for *key*, but operation permission [see *intro*(2)] as specified by the low-order 9 bits of *msgflg* would not be granted.
[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.
[EEXIST]	A message queue identifier exists for *key* but [(*msgflg* & IPC_CREAT) & (*msgflg* & IPC_EXCL)] is "true".

SEE ALSO

intro(2), msgctl(2), msgop(2).

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.

NAME

msgop - message operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (msqid, msgp, msgsz, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz, msgflg;

int msgrcv (msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz;
long msgtyp;
int msgflg;
```

DESCRIPTION

The *msgsnd* 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* member is a positive integer that can be used by the receiving process for message selection (see *msgrcv* below). The *mtext* member is any text of length *msgsz* bytes; *msgsz* can range from 0 to a system-imposed maximum.

The *msgflg* parameter specifies the action to be taken if one or more of the following are true:

- The number of bytes already on the queue is equal to **msg_qbytes** [see *intro*(2)].
- The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

- If (*msgflg* & IPC_NOWAIT) is "true", the message 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.
  - *msqid* is removed from the system [see *msgctl*(2)]. When this occurs, *errno* is set equal to EIDRM, and a value of -1 is returned.
  - The calling process receives a signal that is to be caught. In this case the message is not sent and the calling process resumes execution in the manner prescribed in *signal*(2).

The *msgsnd* call fails and no message is sent if one or more of the following are true:

[EINVAL]	*msqid* is not a valid message queue identifier.
[EACCES]	Operation permission is denied to the calling process [see *intro*(2)].
[EINVAL]	*mtype* is less than 1.
[EAGAIN]	The message cannot be sent for one of the reasons cited above and (*msgflg* & IPC_NOWAIT) is "true".
[EINVAL]	*msgsz* is less than zero or greater than the system-imposed limit.
[EFAULT]	*msgp* points to an illegal address.

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid* [see *intro*(2)].

- **msg_qnum** is incremented by 1.
- **msg_lspid** is set equal to the process ID of the calling process.
- **msg_stime** is set equal to the current time.

msgrcv reads a message from the queue associated with the message queue identifier specified by *msqid* and places it in the structure pointed to by *msgp*. {READ} This structure is composed of the following members:

```
long    mtype;      /* message type */
char    mtext[];    /* message text */
```

mtype is the received message's type as specified by the sending process. *mtext* is the text of the message. *msgsz* specifies the size in bytes of *mtext*. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and (*msgflg* & **MSG_NOERROR**) is "true". The truncated part of the message is lost and no indication of the truncation is given to the calling process.

msgtyp specifies the type of message requested as follows:

- If *msgtyp* is equal to 0, the first message on the queue is received.
- If *msgtyp* is greater than 0, the first message of type *msgtyp* is received.
- If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

msgflg specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:

- If (*msgflg* & **IPC_NOWAIT**) is "true", the calling process 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*(2).

The *msgrcv* call fails and no message is received if one or more of the following are true:

[EINVAL]	*msqid* is not a valid message queue identifier.
[EACCES]	Operation permission is denied to the calling process.
[EINVAL]	*msgsz* is less than 0.
[E2BIG]	*mtext* is greater than *msgsz* and (*msgflg* & MSG_NOERROR) is "false".
[ENOMSG]	The queue does not contain a message of the desired type and (*msgtyp* & IPC_NOWAIT) is "true".
[EFAULT]	*msgp* points to an illegal address.

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid* [see *intro*(2)].

- **msg_qnum** is decremented by 1.
- **msg_lrpid** is set equal to the process ID of the calling process.
- **msg_rtime** is set equal to the current time.

SEE ALSO

intro(2), msgctl(2), msgget(2), signal(2).

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 *msqid* from the system, a value of -1 is returned and *errno* is set to EIDRM.

Upon successful completion, the return value is as follows:

- *msgsnd* returns a value of 0.
- *msgrcv* returns a value equal to the number of bytes actually placed into *mtext*.

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

NAME

nfssys - common shared NFS system calls

SYNOPSIS

```
#include <sys/fs/nfs.h>

int nfssys(cmd, argp)
int cmd;
char *argp;

nfs_getfh(fd, fhp)
int fd;
char *fhp;
{
   struct {
      int fdes;
      char *fhp;
   } args;
   extern int nfssys();

   args.fdes = fd;
   args.fhp = fhp;
   return(nfssys(2, &args));
}

nfs_svc(fd)
int fd;
{
   extern int nfssys ();
   return(nfssys(1, fd);
}

async_daemon()
{
   extern int nfssys();
   return(nfssys(3, 0));
}
```

DESCRIPTION

The *nfssys* system call is provided to allow NFS daemons [through *nfs_getfh*(), *nfs_svc*(), and *async_daemon*() routines] to enter the kernel. Note that this call is *not* intended for general purpose use, and is described here only for illustration.

The *cmd* argument to *nfssys* specifies the NFS routine to use:

1	is *nfs_svc*()
2	is *nfs_getfh*()
3	is *async_daemon*().

The *argp* argument is the error return.

The *nfs_getfh* routine, in the *mountd* mount daemon, returns a file handle for the file open as file descriptor *fdes*.

```
nfs_getfh(fd, fhp)
int fd;
char *fhp;
{
  struct {
    int fdes;
    char *fhp;
  } args;
  extern int nfssys();

  args.fdes = fd;
  args.fhp = fhp;
  return(nfssys(2, &args));
}
```

The *nfs_svc*() and *async_daemon*() routines allow kernel processes to have a user context. The *nfs_svc* routine starts the *nfsd* daemon listening on socket *sock*. The socket (in 4.2BSD terminology) must be AF_INET and SOCK_DGRAM (protocol UDP/IP), but this is completely dependent on the local network transport implementation. This system call returns only if the process is killed.

```
nfs_svc(fd)
int fd;
{
  extern int nfssys ();
  return(nfssys(1, fd);
}
```

The *async_daemon* routine implements the NFS *biod* daemon, which handles asynchronous I/O for an NFS client; it never returns.

```
async_daemon()
{
  extern int nfssys();
  return(nfssys(3, 0));
}
```

SEE ALSO

mountd(1M), nfsd(1M).

NAME

nice - change priority of a process

SYNOPSIS

int nice (incr)
int incr;

DESCRIPTION

nice adds the value of *incr* to the nice value of the calling process. A process's *nice value* is a non-negative number for which a more positive value results in lower CPU priority.

The system allows *nice* values only from -8 to 39. The *nice* values -8 to -1 are not accepted unless the **syslocal(SYSLRTNICE)** is executed to enable the mechanism. The *nice* system call grants *nice* values from -8 to -1 only to super-user processes. These negative *nice* values cause the CPU priority of the process to be fixed independently of CPU usage of the process. *nice* values from 0 to 39 allow the system to adjust dynamically the actual CPU priority of the process, temporarily lowering it in proportion to the process's recent level of CPU usage. If a super-user process requires a *nice* value below -8, or if any other process requests a *nice* value below 0, the system imposes a *nice* value of 0. If any process requests a *nice* value above 39, the system imposes a *nice* value of 39.

[EPERM]	*nice* will fail and 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.

SEE ALSO

nice(1), rtpenable(1M), exec(2).

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.

NAME

nice - change priority of a process

SYNOPSIS

int nice (incr)
int incr;

DESCRIPTION

nice adds the value of *incr* to the nice value of the calling process. A process's *nice value* is a non-negative number for which a more positive value results in lower CPU priority.

The system allows *nice* values only from -8 to 39. The *nice* values -8 to -1 are not accepted unless the **syslocal(SYSLRTNICE)** is executed to enable the mechanism. The *nice* system call grants *nice* values from -8 to -1 only to super-user processes. These negative *nice* values cause the CPU priority of the process to be fixed independently of CPU usage of the process. *nice* values from 0 to 39 allow the system to adjust dynamically the actual CPU priority of the process, temporarily lowering it in proportion to the process's recent level of CPU usage. If a super-user process requires a *nice* value below -8, or if any other process requests a *nice* value below 0, the system imposes a *nice* value of 0. If any process requests a *nice* value above 39, the system imposes a *nice* value of 39.

[EPERM]	*nice* will fail and 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.

SEE ALSO

nice(1), rtpenable(1M), exec(2).

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.

NAME

notify, unnotify, evwait, evnowait - manage notifications

SYNOPSIS

#include <notify.h>

int notify(type, arg, tag)
ushort type;
char *arg;
char *tag;

int unnotify(type, arg)
ushort type;
char *arg;

ushort evwait(tag, datum)
char **tag;
char **datum;

ushort evnowait(tag, datum)
char **tag;
char **datum;

DESCRIPTION

The *notify* system call interface allows a user process to record a number of events that it is interested in, and then waits for any one of them. Like *select*(2), it does synchronous I/O multiplexing, but *notify* waits for a wider range of events and thus has greater functionality than *select*.

The *notify* call requests a notification or set of notifications.

The *unnotify* call retracts an earlier request (or set of requests) for notification.

The *evwait* call waits for a notification to be posted to the calling process.

The *evnowait* call returns the first notification if one exists, returning immediately otherwise.

Notifications are posted FIFO (first-in, (frst-out) in the user process, each *evwait* returning the first notification or blocking until one is posted. When a *notify* call is given the user must supply the *type* of notification, a *tag*, and an *argument*. The *tag* is an arbitrary number the size of a (**char ***), which is returned by any *evwait* call triggered by that notification request. The *argument* is type specific and is described below.

The return values of *evwait* and *evnowait* are the *type* of the notification.

It is an error for *notify* to be called with a *type* and *arg* matching a currently active notification.

The *notify* calls support the following *type*s:

N_FDREAD

Queue a notification if the file descriptor *arg* is readable at the time of the *notify* call, and subsequently whenever there is data to be read. A notification is also queued at end-of-file or when the number of writers on a pipe goes to zero. The datum returned from an *evwait* is a count of the number of bytes available to be read, unless the notification is for a terminal device in cooked mode; in this case, the count is actually the number of delimiters encountered (*that is,* the number of *reads* required to get all data). At EOF the datum is -1, and the request is deleted. This type is implemented for sockets, pipes, ttys, and streams.

N_FDWRITE

Queue a notification if the file descriptor *arg* is writable at the time of the *notify* call, and subsequently when the file goes from a non-writable to a writable state (that is, output is not blocked). *Datum* is the number of characters writable. This type is implemented for sockets, pipes, and streams.

N_SIGNAL

Queue a notification on receipt of a signal. This is used in conjunction with regular signal catching [see signal(2)]. When signal notification is in effect, all caught signals queue notifications instead of causing pseudo-interrupts. If multiple instances of a caught signal occur before the process has received the notification, the returned type is **N_LOSTSIG** rather than **N_SIGNAL**. Ignored or defaulted signals are handled normally. Signals are not reset upon notification.

Note that only one call to *notify*

```
notify(N_SIGNAL,ignored,tag)
```

is required to enable notification of all signals that have a signal catching function (use a null function). *Evwait* and *evnowait* return the *tag* and *datum*. *Datum* is a bitwise OR of all queued signals: that is, low-numbered signals are represented as low-order bits (signal *n* sets 2^{n-1}).

N_UMSGREAD, N_UMSGWRITE

Queue a notification if the message queue described by *arg* is or becomes readable or writable, respectively. The datum returned is the number of messages received or the number of characters that can be sent, respectively. When the message queue is removed, *datum* is -1, and the request is deleted.

N_INDIR

If *type* is **N_INDIR**, *arg* is acually a pointer to an array of the following structure (defined in **/usr/include/notify.h**):

```
struct n_request {
        ushort type;
        char *arg;
        char *tag;
}
```

The array should be terminated with an entry having *type* **N_INDIR**. The entire set of notifications is either placed or removed. **N_INDIR** is never returned by *evwait* or *evnowait.*

N_QUERY

Type **N_QUERY** is valid only as an argument to the *notify* call. *arg* is a pointer to an array of **struct n_indir**, and *tag* is a pointer to an **int** containing the number of elements in the array.

On return, the array contains the current active notifications in a form suitable for passing to *notify* or *unnotify* (that is, terminated by **N_INDIR**), and the **int** pointed to by *tag* contains the number of active notifications (even if there was not enough space to copy them all back).

N_SEMOP

Queue a notification if the semaphore described by the **struct n_semop** (below) pointed to by *arg* would not block, is released, or is removed. *Datum* is **semval** unless the semaphore has been removed, in which case it is -1.

```
struct n_semop {
                int semid;      /* semaphore ID */
                short sem_num;  /* semaphore number */
                short sem_op;   /* semaphore operation */
}
```

SEE ALSO

fcntl(2), msgop(2), pipe(2), read(2), select(2), signal(2), socket(2), wait(2), termio(7).

DIAGNOSTICS

All calls return -1 on error, setting *errno* to one of the following:

[EINVAL]	Invalid type was given
[EINVAL]	Caller never did a *notify* (*unnotify, evwait, evnowait*)
[EINVAL]	File is not of a valid type (**N_FDREAD, N_FDWRITE**).
[EBADF]	File is not open (**N_FDREAD, N_FDWRITE**)
[EBADF]	Invalid message queue descriptor (N_UMSG)
[ENOSPC]	No space available to allocate notification queue header
[ENOSPC]	No space available to allocate table entry for this notification
[ENOSPC]	Too many active notification requests for given space (N_QUERY)
[EFAULT]	An address fault was generated by a user-supplied pointer

EXAMPLE

```
#include "sys/types.h"
#include <sys/notify.h>
#include <stdio.h>
#include <signal.h>

int sig_catch();

main()
{
      int tag, datum, i;
      char buf[BUFSIZ];
      ushort rv, evwait();

      setbuf(stdout, NULL);
      if (notify(N_FDREAD, 0, 't') < 0)
            perror("notify for N_FDREAD of stdin failed"), exit(1);

      if (notify(N_SIGNAL, 2, 's') < 0)
            perror("notify failed"), exit(1);

      for (i=0; i<20; i++)
            signal(i, sig_catch);
```

```
        for(;;) {
                /* Wait for an event */
                rv = evwait(&tag, &datum);

                /* Tell the user about it */
                printf("0v: %d tag: %d datum: %d0, rv, tag, datum);

                switch (tag) {
                case 's':
                        break;
                case 't':
                        /* Read the input */
                        gets(buf);
                        printf("read '%s'0, buf);
                        if (*buf == 'q')
                        exit(0);
                        break;
                }
        }
}
sig_catch()
{
}
```

WARNING

The *notify* system call interface is not portable, has little likelihood of becoming so, and may disappear in future releases of CTIX. It is therefore recommended that you use the *poll*(2) system call, and that existing software using *notify* be changed to use poll.

NAME

open - open for reading or writing

SYNOPSIS

#include <fcntl.h>
int open (path, oflag [, mode])
char *path;
int oflag, mode;

DESCRIPTION

The *open* call opens a file descriptor for the file pointed to by *path*, and sets the file status flags according to the value of *oflag*. For non-STREAMS [see *intro*(2)] files, *oflag* values are constructed by or-ing flags from the following list (only one of the first three flags below may be used):

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NDELAY This flag may affect subsequent reads and writes [see *read*(2) and *write*(2)].

When opening a FIFO with O_RDONLY or O_WRONLY set:

If O_NDELAY is set:

An *open* for reading-only will return without delay. An *open* for writing-only will return an error if no process currently has the file open for reading.

If O_NDELAY is clear:

An *open* for reading-only will block until a process opens the file for writing. An *open* for writing-only will block until a process opens the file for reading.

When opening a file associated with a communication line:

If O_NDELAY is set:

The open will return without waiting for carrier.

If O_NDELAY is clear:

The open will block until carrier is present.

O_APPEND If set, the file pointer will be set to the end of the file prior to each write.

O_DIRECT If set, subsequent reads or writes that satisfy the following criteria are moved directly to or from the user space to the physical media:

- The transfer must start on a 1K byte boundary in the file, and it must be in multiples of 1K byte blocks.

O_SYNC When opening a regular file, this flag affects subsequent writes. If set, each *write*(2) will wait for both the file data and file status to be physically updated.

O_CREAT If the file exists, this flag has no effect. Otherwise, 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, and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows [see *creat*(2)]:

- All bits set in the file mode creation mask of the process are cleared [see *umask*(2)].
- The ''save text image after execution bit'' of the mode is cleared [see *chmod*(2)].

O_TRUNC
If the file exists, its length is truncated to 0 and the mode and owner are unchanged.

O_EXCL
If O_EXCL and O_CREAT are set, *open* will fail if the file exists.

When opening a STREAMS file, *oflag* may be constructed from O_NDELAY 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_NDELAY affects the operation of STREAMS drivers and certain system calls [see *read*(2), *getmsg*(2), *putmsg*(2) and *write*(2)]. For drivers, the implementation of O_NDELAY is device-specific. Each STREAMS device driver may treat this option differently.

Certain flag values can be set following *open* as described in *fcntl*(2).

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is set to remain open across *exec* system calls [see *fcntl*(2)].

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

[EACCES]	A component of the path prefix denies search permission.
[EACCES]	*oflag* permission is denied for the named file.
[EAGAIN]	The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file [see *chmod* (2)].
[EEXIST]	O_CREAT and O_EXCL are set, and the named file exists.
[EFAULT]	*path* points outside the allocated address space of the process.
[EINTR]	A signal was caught during the *open* system call.
[EIO]	A hangup or error occurred during a STREAMS *open*.
[EISDIR]	The named file is a directory and *oflag* is write or read/write.
[EMFILE]	NOFILES file descriptors are currently open.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.
[ENFILE]	The system file table is full.
[ENOENT]	O_CREAT is not set and the named file does not exist.
[ENOLINK]	*path* 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 *stream*.
[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.
[ENXIO]	O_NDELAY is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading.
[ENXIO]	A STREAMS module or driver open routine failed.
[EROFS]	The named file resides on a read-only file system and *oflag* is write or read/write.

[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed and *oflag* is write or read/write.
[EDEADLOCK]	A side effect of the *locking*(2) call, when applying OTRUNC. [See the WARNING on the *locking*(2) manpage.]

SEE ALSO

chmod(2), close(2), creat(2), dup(2), fcntl(2), getmsg(2), intro(2), lseek(2), read(2), putmsg(2), umask(2), write(2).

DIAGNOSTICS

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

NAME

pause - suspend process until signal

SYNOPSIS

pause ()

DESCRIPTION

The *pause* 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*(2)], the calling process resumes execution from the point of suspension; with a return value of -1 from *pause* and *errno* set to EINTR.

SEE ALSO

alarm(2), kill(2), signal(2), wait(2).

NAME

pipe - create an interprocess channel

SYNOPSIS

int pipe (fildes)
int fildes[2];

DESCRIPTION

pipe creates an I/O mechanism called a pipe and returns two file descriptors, *fildes*[0] and *fildes*[1]. *fildes*[0] is opened for reading and *fildes*[1] is opened for writing.

Up to 9,216 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.

pipe will fail if:

[EMFILE]	NOFILES file descriptors are currently open.
[ENFILE]	The system file table is full.

SEE ALSO

sh(1), read(2), write(2).

DIAGNOSTICS

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

NAME

plock - lock process, text, or data in memory

SYNOPSIS

#include <sys/lock.h>

int plock (op)
int op;

DESCRIPTION

plock allows the calling process to lock its text segment (text lock), its data and stack segments (data lock), or 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. For 407 object modules TXTLOCK and DATLOCK are identical.

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

Shared regions (for example, text) may be locked by anyone using the text, but they may be unlocked only if the caller is the last one using the region. Note that sticky-bit text that is not explicitly unlocked will remain locked in core even after the last process using it terminates.

plock will fail and not perform the requested operation if one or more of the following are true:

[EPERM]	The effective user ID of the calling process is not super-user.
[EINVAL]	*op* is equal to **PROCLOCK** and a process lock, a text lock, or a data lock already exists on the calling process.
[EINVAL]	*op* is equal to **TXTLOCK** and a text lock, or a process lock already exists on the calling process.
[EINVAL]	*op* is equal to **DATLOCK** and a data lock, or a process lock already exists on the calling process.
[EINVAL]	*op* is equal to **UNLOCK** and no type of lock exists on the calling process.
[EAGAIN]	Not enough memory.

SEE ALSO

exec(2), exit(2), fork(2).

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.

NAME

poll - STREAMS input/output multiplexing

SYNOPSIS

```
#include <stropts.h>
#include <poll.h>

int poll(fds, nfds, timeout)
struct pollfd fds[];
unsigned long nfds;
int timeout;
```

DESCRIPTION

poll provides users with a mechanism for multiplexing input/output over a set of file descriptors that reference open *streams* [see *intro*(2)]. *poll* 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*(2) or *getmsg*(2) and can send messages using *write*(2) and *putmsg*(2). Certain *ioctl*(2) calls, such as I_RECVFD and I_SENDFD [see *streamio*(7)], 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 bitmasks 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 POLLOUT 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*(2)], *poll* will fail.

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. *poll* is not affected by the O_NDELAY flag.

poll fails if one or more of the following are 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.

SEE ALSO

intro(2), read(2), getmsg(2), putmsg(2), write(2), streamio(7).
UNIX System V Release 3.2 Streams Primer.
UNIX System V Release 3.2 Streams Programmer's Guide.

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.

NAME

profil - execution time profile

SYNOPSIS

void profil (buff, bufsiz, offset, scale)
char *buff;
int bufsiz, offset, scale;

DESCRIPTION

buff points to an area of core whose length (in bytes) is given by *bufsiz*. After this call, the user's program counter (pc) is examined each clock tick. 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 will be turned off if an update in *buff* would cause a memory fault.

SEE ALSO

prof(1), times(2), monitor(3C).

DIAGNOSTICS

Not defined.

NAME

ptrace - process trace

SYNOPSIS

int ptrace (request, pid, addr, data);
int request, pid, addr, data;

DESCRIPTION

ptrace provides a means by which a parent process may control the execution of a child process. Its primary use is for the implementation of breakpoint debugging [see *sdb*(1)]. The child process behaves normally until it encounters a signal [see *signal*(2) for the list], at which time it enters a stopped state and its parent is notified via *wait*(2). When the child is in the stopped state, its parent can examine and modify its ''core image'' using *ptrace*. Also, the parent can cause the child either to terminate or continue, with the possibility of ignoring the signal that caused it to stop.

The *request* argument determines the precise action to be taken by *ptrace* and is one of the following:

0 This request must be issued by the child process if it is to be traced by its parent. It turns on the child's trace flag that stipulates that the child should be left in a stopped state upon receipt of a signal rather than the state specified by *func* [see *signal*(2)]. The *pid*, *addr*, and *data* arguments are ignored, and a return value is not defined for this request. Peculiar results will ensue if the parent does not expect to trace the child.

The remainder of the requests can be used only 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. These two requests will fail if *addr* is not the start address of a word, in which case a value of -1 is returned to the parent process and the parent's *errno* is set to EIO.

3 With this request, the word at location *addr* in the child's USER area in the system's address space (see **<sys/user.h>**) is returned to the parent process. Addresses in this area range from 0 to ctob (USIZE) on Convergent Technologies 680x0-family processors. The *data*

argument is ignored. This request will fail if *addr* is not the start address of a word or is outside the USER area, in which case a value of -1 is returned to the parent process and the parent's *errno* is set to EIO.

4, 5 With these requests, the value given by the *data* argument is written into the address space of the child at location *addr*. If I and D space are separated (as on PDP-11), 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 (as on Convergent Technologies 680x0-family processors) 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 will fail if *addr* is not the start address of a word. Upon failure a value of -1 is returned to the parent process and the parent's *errno* is set to EIO.

6 With this request, a few entries in the child's USER area can be written. *Data* gives the value that is to be written and *addr* is the location of the entry. The few entries that can be written are:

- the general registers (that is, registers 0 to 15 on Convergent Technologies 680x0-family processors)
- the condition codes of the Processor Status Word.

7 This request causes the child to resume execution. If the *data* argument is 0, all pending signals including the one that caused the child to stop are canceled before it resumes execution. If the *data* argument is a valid signal number, the child resumes execution as if it had incurred that signal, and any other pending signals are canceled. The *addr* argument must be equal to 1 for this request. Upon successful completion, the value of *data* is returned to the parent. This request will fail if *data* is not 0 or a valid signal number, in which case a value of -1 is returned to the parent process and the parent's *errno* is set to EIO.

8 This request causes the child to terminate with the same consequences as *exit*(2).

9 This request sets the trace bit in the Processor Status Word of the child and then executes the same steps as listed above for request 7. The trace bit causes an interrupt upon completion of one machine instruction. This effectively allows single stepping of the child.

To forestall possible fraud, *ptrace* inhibits the set-user-ID facility on subsequent *exec*(2) calls. If a traced process calls *exec*, it will stop before executing the first instruction of the new image showing signal **SIGTRAP**.

General Errors

ptrace will in general fail if one or more of the following are true:

[EIO]	*request* is an illegal number.
[ESRCH]	*pid* identifies a child that does not exist or has not executed a *ptrace* with request **0**.

SEE ALSO

sdb(1), exec(2), signal(2), wait(2).

NAME

putmsg - send a message on a stream

SYNOPSIS

```
#include <stropts.h>

int putmsg (fd, ctlptr, dataptr, flags)
int fd;
struct strbuf *ctlptr;
struct strbuf *dataptr;
int flags;
```

DESCRIPTION

The *putmsg* call creates a message [see *intro*(2)] from user specified buffer(s) and sends the message to a STREAMS file. The message can 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*(2)]. 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 will be 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* will block 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_NDELAY is set. Instead, it fails and sets *errno* to EAGAIN.

putmsg 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_NDELAY has been specified. No partial message is sent.

putmsg fails if one or more of the following are true:

[EAGAIN]	A non-priority message was specified, the O_NDELAY flag is set and the *stream* write queue is full due to internal flow control conditions.
[EAGAIN]	Buffers could not be allocated for the message that was to be created.
[EBADF]	*fd* is not a valid file descriptor open for writing.
[EFAULT]	*ctlptr* or *dataptr* points outside the allocated address space.
[EINTR]	A signal was caught during the *putmsg* system call.
[EINVAL]	An undefined value was specified in *flags*, or *flags* is set to RS_HIPRI and no control part was supplied.
[EINVAL]	The *stream* referenced by *fd* is linked below a multiplexor.
[ENOSTR]	A *stream* is not associated with *fd*.
[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.

SEE ALSO

intro(2), read(2), getmsg(2), poll(2), write(2).
UNIX System V Release 3.2 Streams Programmer's Guide.
UNIX System V Release 3.2 Streams Primer.

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.

NAME

read - read from file

SYNOPSIS

int read (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

DESCRIPTION

The *fildes* argument is a file descriptor obtained from a *creat*(2), *open*(2), *dup*(2), *fcntl*(2), or *pipe*(2) system call.

The *read* 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*(2) and *termio*(7)], or if the number of bytes left in the file is less than *nbyte* bytes. A value of 0 is returned when an end-of-file has been reached.

A *read* from a STREAMS [see *intro*(2)] 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*(7)], 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 there is no more data to 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*(2) 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*(2)], and there is a blocking (owned by another process) write lock on the segment of the file to be read:

- If O_NDELAY is set, the read returns a -1 and set errno to EAGAIN.
- If O_NDELAY is clear, the read sleeps until the blocking record lock is removed.

When attempting to read from an empty pipe (or FIFO):

- If O_NDELAY is set, the read returns a 0.
- If O_NDELAY 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_NDELAY is set, the read returns 0.
- If O_NDELAY 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_NDELAY is set, the read returns a -1 and set errno to EAGAIN.
- If O_NDELAY 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* 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* call fails if one or more of the following are true:

[EAGAIN]	Mandatory file/record locking was set, O_NDELAY 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 *stream* and O_NDELAY flag set.
[EBADF]	*fildes* is not a valid file descriptor open for reading.
[EBADMSG]	Message waiting to be read on a *stream* is not a data message.
[EDEADLK]	The read was going to go to sleep and cause a deadlock situation to occur.
[EFAULT]	*buf* points outside the allocated address space.
[EINTR]	A signal was caught during the *read* system call.
[EINVAL]	Attempted to read from a *stream* linked to a multiplexor.
[ENOLCK]	The system record lock table was full, so the read 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.
[EDEADLOCK]	A side effect of the *locking*(2) call. (See the WARNING on the *locking*(2) manpage.)

A *read* from a STREAMS file also fails if an error message is received at the *stream head*. In this case, *errno* is set to the value returned in the error message. If a hangup occurs on the *stream* being read, *read* continues to operate normally until the *stream head* read queue is empty. Thereafter, it returns 0.

SEE ALSO

creat(2), dup(2), fcntl(2), getmsg(2), ioctl(2), intro(2), locking(2), open(2), pipe(2), streamio(7), termio(7).

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.

NAME

recv, recvfrom - receive a message from a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int recv(s, buf, len, flags)
int s;
char *buf;
int len, flags;

int recvfrom(s, buf, len, flags, from, fromlen)
int s;
char *buf;
int len, flags;
struct sockaddr *from;
int *fromlen;
```

DESCRIPTION

The *recv* and *recvfrom* calls are used to receive messages from a socket.

The *recv* call can be used only on a *connected* socket [see *connect*(2)], while *recvfrom* can be used to receive data on a socket whether it is in a connected state or not.

If *from* is non-zero, the source address of the message is filled in. *fromlen* is a value-result parameter, initialized to the size of the buffer associated with *from*, and modified on return to indicate the actual size of the address stored there. The length of the message is returned in *cc*. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from; see *socket*(2).

If no messages are available at the socket, the receive call waits for a message to arrive.

The *flags* argument to a send call is formed by *or*'ing one or more of the values:

```
#define  MSG_PEEK   0x1    /* peek at incoming message */
#define  MSG_OOB    0x2    /* process out-of-band data */
```

RETURN VALUE

These calls return the number of bytes received, or -1 if an error occurred.

ERRORS

The calls fail if:

[EBADF]	The argument *s* is an invalid descriptor.
[ENOTSOCK]	The argument *s* is not a socket.
[EINTR]	The receive was interrupted by delivery of a signal before any data was available for the receive.
[EFAULT]	The data was specified to be received into a non-existent or protected part of the process address space.

SEE ALSO

connect(2), intro(2), read(2), send(2), socket(2), intro(7).
CTIX Network Programmer's Primer.

NAME

rmdir - remove a directory

SYNOPSIS

int rmdir (path)
char *path;

DESCRIPTION

The *rmdir* call removes the directory named by the path name pointed to by *path*. The directory must not have any entries other than the dot (.) and dot dot (..) files.

The named directory is removed unless one or more of the following are true:

[EINVAL]	The current directory should not be removed.
[EINVAL]	The dot (.) entry of a directory should not be removed.
[EEXIST]	The directory contains entries other than those for dot (.) and dot dot (..).
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named directory does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	Write permission is denied on the directory containing the directory to be removed.
[EACCES]	The parent directory has the sticky bit set and: the parent directory is not owned by the user; and the directory is not owned by the user; and the directory is not writable by the user; and the user is not super-user.
[EBUSY]	The directory to be removed is the mount point for a mounted file system.
[EROFS]	The directory entry to be removed is part of a read-only file system.
[EFAULT]	*Path* points outside the process's allocated address space.
[EIO]	An I/O error occurred while accessing the file system.
[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.

SEE ALSO

mkdir(1), rm(1), rmdir(1), mkdir(2).

DIAGNOSTICS

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

NAME

select - synchronous I/O multiplexing

SYNOPSIS

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

nfound = select(nfds, readfds, writefds, exceptfds, timeout)
int nfound, nfds;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;

FD_SET(fd, &fdset)
FD_CLR(fd, &fdset)
FD_ISSET(fd, &fdset)
FD_ZERO(&fdset)
int fd;
fd_set fdset;
```

DESCRIPTION

The *select* call 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: *D_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*, otherwiseit is zero. 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-zero pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a zero pointer, the select blocks indefinitely. To affect a poll, the *timeout* argument should be non-zero, pointing to a zero-valued timeval structure.

Any of *readfds*, *writefds*, and *exceptfds* may be given as zero pointers if no descriptors are of interest.

SEE ALSO

accept(2), connect(2), getdtablesize(2), read(2), recv(2), send(2), write(2).
CTIX Network Programmer's Primer.

RETURN VALUE

The *select* call 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 will be unmodified.

ERRORS

Returned error codes from *select* are as follows:

[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.

BUGS

Although the provision of *getdtablesize* (2) was intended to allow user programs to be written independent of the kernel limit on the number of open files, the dimension of a sufficiently large bit field for select remains a problem. The default size FD_SETSIZE (currently 256) is somewhat larger than the current kernel limit to the number of open files. However, in order to accommodate programs that might potentially use a larger number of open files with select, you can to increase this size within a program by providing a larger definition of FD_SETSIZE before the inclusion of <sys/types.h>.

The *select* call 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 will be unmodified by the *select* call.

NAME

semctl - semaphore control operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (semid, semnum, cmd, arg)
int semid, cmd;
int semnum;
union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
} arg;
```

DESCRIPTION

The *semctl* 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* [see *intro*(2)]. {READ}
SETVAL	Set the value of *semval* to *arg.val*. {ALTER} When this cmd is successfully executed, the semadj value corresponding to the specified semaphore in all processes is cleared.
GETPID	Return the value of *sempid*. {READ}
GETNCNT	Return the value of *semncnt*. {READ}
GETZCNT	Return the value of *semzcnt*. {READ}

The following *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 *cmd*s are also available:

IPC_STAT
Place the current value of each member of the data structure associated with *semid* into the structure pointed to by *arg.buf*. The contents of this structure are defined in *intro*(2). {READ}

IPC_SET
Set the value of the following members of the data structure associated with *semid* to the corresponding value found in the structure pointed to by *arg.buf*:

```
sem_perm.uid
sem_perm.gid
sem_perm.mode  /* only low 9 bits */
```

This cmd can be executed only 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* call fails if one or more of the following are true:

[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.
[EACCES]	Operation permission is denied to the calling process [see *intro*(2)].
[ERANGE]	*cmd* is **SETVAL** or **SETALL** and the value to which *semval* is to be set is greater than the system imposed maximum.
[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*.
[EFAULT]	*arg.buf* points to an illegal address.

SEE ALSO

intro(2), semget(2), semop(2).

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.

NAME

semget - get set of semaphores

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget (key, nsems, semflg)
key_t key;
int nsems, semflg;
```

DESCRIPTION

The *semget* call returns the semaphore identifier associated with *key*.

A semaphore identifier and associated data structure and set containing *nsems* semaphores [see *intro*(2)] are created for *key* if one of the following 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.gid**, and **sem_perm.cgid** 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 *semget* call fails if one or more of the following are true:

[EINVAL]	*nsems* is either less than or equal to zero or greater than the system-imposed limit.
[EACCES]	A semaphore identifier exists for *key*, but operation permission [see *intro*(2)] as specified by the low-order 9 bits of *semflg* would not be granted.
[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.
[EEXIST]	A semaphore identifier exists for *key* but [(*semflg* & IPC_CREAT) and (*semflg* & IPC_EXCL)] is "true".

SEE ALSO

intro(2), semctl(2), semop(2).

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.

NAME

semop - semaphore operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop (semid, sops, nsops)
int semid;
struct sembuf **sops;
unsigned nsops;
```

DESCRIPTION

The *semop* call is used to automatically perform an array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by *semid*. The *sops* parameter is a pointer to the array of semaphore-operation structures; *nsops* is the number of such structures in the array. The contents of each structure includes the following members:

```
short   sem_num;  /* semaphore number */
short   sem_op;   /* semaphore operation */
short   sem_flg;  /* operation flags */
```

Each semaphore operation specified by *sem_op* is performed on the corresponding semaphore specified by *semid* and *sem_num*.

The *sem_op* member specifies one of three semaphore operations as follows:

- If *sem_op* is a negative integer, one of the following occurs {ALTER}:
  - If *semval* [see *intro*(2)] is greater than or equal to the absolute value of *sem_op*, the absolute value of *sem_op* is subtracted from semval. Also, if (*sem_flg* & SEM_UNDO) is ''true'', the absolute value of *sem_op* is added to the calling process's *semadj* value [see *exit*(2)] for the specified semaphore.
  - If *semval* is less than the absolute value of *sem_op* and (*sem_flg* & IPC_NOWAIT) is ''true'', *semop* returns immediately.
  - If *semval* is less than the absolute value of *sem_op* and (*sem_flg* & IPC_NOWAIT) is ''false'', *semop* increments the *semncnt* associated with the specified semaphore and suspends execution of the calling process until one of the following conditions occur.

        + *Semval* becomes greater than or equal to the absolute value of *sem_op*. When this occurs, the value of *semncnt* associated with the specified semaphore is decremented, the absolute value of *sem_op* is subtracted from *semval* and, if (*sem_flg* & SEM_UNDO) is "true", the absolute value of *sem_op* is added to the calling process's *semadj* value for the specified semaphore.

        + The *semid* for which the calling process is awaiting action is removed from the system [see *semctl*(2)]. When this occurs, *errno* is set equal to EIDRM, and a value of -1 is returned.

        + The calling process receives a signal that is to be caught. When this occurs, the value of *semncnt* associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in *signal*(2).

- If *sem_op* is a positive integer, the value of *sem_op* is added to *semval* and, if (*sem_flg* & SEM_UNDO) is "true", the value of *sem_op* is subtracted from the calling process's *semadj* value for the specified semaphore. {ALTER}

- If *sem_op* is zero, one of the following 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*(2).

The *semop* call fails if one or more of the following are true for any of the semaphore operations specified by *sops*:

[EINVAL]	*semid* is not a valid semaphore identifier.
[EFBIG]	*sem_num* is less than zero or greater than or equal to the number of semaphores in the set associated with *semid*.
[E2BIG]	*nsops* is greater than the system-imposed maximum.
[EACCES]	Operation permission is denied to the calling process [see *intro*(2)]
[EAGAIN]	The operation would result in suspension of the calling process but (*sem_flg* & **IPC_NOWAIT**) is "true".
[ENOSPC]	The limit on the number of individual processes requesting an **SEM_UNDO** would be exceeded.
[EINVAL]	The number of individual semaphores for which the calling process requests a **SEM_UNDO** would exceed the limit.
[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.
[EFAULT]	*sops* points to an illegal address.

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.

SEE ALSO

exec(2), exit(2), fork(2), intro(2), semctl(2), semget(2).

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.

NAME

send, sendto - send a message to a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int send(s, msg, len, flags)
int s;
char *msg;
int len, flags;

int sendto(s, msg, len, flags, to, tolen)
int s;
char *msg;
int len, flags;
struct sockaddr *to;
int tolen;
```

DESCRIPTION

The *send* and *sendto* calls are used to transmit a message to another socket (*s*). The *send* call can be used only when the socket is in a *connected* state, while *sendto* can be used at any time.

The address of the target is given by *to* with *tolen* specifying its size. The length of the message is given by *len*. If the message is too long to pass atomically through the underlying protocol, then the error EMSGSIZE is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a *send*. Return values of -1 indicate some locally detected errors.

If no message space is available at the socket to hold the message to be transmitted, *send* blocks.

The *flags* parameter can be set to SOF_OOB to send "out-of-band" data on sockets which support this notion (SOCK_STREAM).

SEE ALSO

intro(2), recv(2), socket(2), intro(7).
CTIX Network Programmer's Primer.

RETURN VALUE

The call returns the number of characters sent, or -1 if an error occurred.

ERRORS

[EBADF]	An invalid descriptor was specified.
[ENOTSOCK]	The argument *s* is not a socket.
[EFAULT]	An invalid user space address was specified for a parameter.
[EMSGSIZE]	The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.

NAME

setpgrp - set process group ID

SYNOPSIS

int setpgrp ()

DESCRIPTION

The *setpgrp* call sets the process group ID of the calling process to the process ID of the calling process and returns the new process group ID.

SEE ALSO

exec(2), fork(2), getpid(2), intro(2), kill(2), signal(2), termio(7).

DIAGNOSTICS

The *setpgrp* call returns the value of the new process group ID.

NAME

setuid, setgid - set user and group IDs

SYNOPSIS

int setuid (uid)
int uid;

int setgid (gid)
int gid;

DESCRIPTION

The *setuid* (*setgid*) call is used to set the real user (group) ID and effective user (group) ID of the calling process.

If the effective user ID of the calling process is super-user, the real user (group) ID and effective user (group) ID are set to *uid* (*gid*).

If the effective user ID of the calling process is not super-user, but its real user (group) ID is equal to *uid* (*gid*), the effective user (group) ID is set to *uid* (*gid*).

If the effective user ID of the calling process is not super-user, but the saved set-user (group) ID from *exec*(2) is equal to *uid* (*gid*), the effective user (group) ID is set to *uid* (*gid*).

The *setuid* (*setgid*) call fails if any of the following conditions are true:

[EPERM]
The real user (group) ID of the calling process is not equal to *uid* (*gid*) and its effective user ID is not super-user.

[EINVAL]
The *uid* is out of range.

SEE ALSO

getuid(2), intro(2).

DIAGNOSTICS

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

NAME

shmctl - shared memory control operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl (shmid, cmd, buf)
int shmid, cmd;
struct shmid_ds *buf;
```

DESCRIPTION

The *shmctl* call provides a variety of shared memory control operations as specified by *cmd*. The following *cmds* are available:

IPC_STAT
: Place the current value of each member of the data structure associated with *shmid* into the structure pointed to by *buf*. The contents of this structure are defined in *intro*(2). {READ}

IPC_SET
: Set the value of the following members of the data structure associated with *shmid* to the corresponding value found in the structure pointed to by *buf*:

```
shm_perm.uid
shm_perm.gid
shm_perm.mode   /* only low 9 bits */
```

This *cmd* can be executed only 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 *shmid*.

IPC_RMID
: Remove the shared memory identifier specified by *shmid* from the system and destroy the shared memory segment and data structure associated with it. This *cmd* can be executed only 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 *shmid*.

SHM_LOCK
: Lock the shared memory segment specified by *shmid* in memory. This *cmd* can be executed only by a process that has an effective user ID equal to super-user.

SHM_UNLOCK	Unlock the shared memory segment specified by *shmid*. This *cmd* can be executed only by a process that has an effective user ID equal to super user.

The *shmctl* call fails if one or more of the following are true:

[EINVAL]	*shmid* is not a valid shared memory identifier.
[EINVAL]	*cmd* is not a valid command.
[EACCES]	*cmd* is equal to **IPC_STAT** and {READ} operation permission is denied to the calling process [see *intro*(2)].
[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.
[EFAULT]	*buf* points to an illegal address.
[ENOMEM]	*cmd* is equal to **SHM_LOCK** and there is not enough memory.

SEE ALSO

intro(2), shmget(2), shmop(2).

DIAGNOSTICS

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

NOTES

The user must explicitly remove shared memory segments after the last reference to them has been removed.

NAME

shmget - get shared memory segment identifier

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget (key, size, shmflg)
key_t key;
int size, shmflg;

DESCRIPTION

The *shmget* 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 [see *intro*(2)] are created for *key* if one of the following are true:

- *key* is equal to **IPC_PRIVATE.**
- *key* does not already have a shared memory identifier associated with it, and (*shmflg* & **IPC_CREAT**) is "true".

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

- **shm_perm.cuid**, **shm_perm.uid**, **shm_perm.gid**, and **shm_perm.cgid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order nine bits of **shm_perm.mode** are set equal to the low-order nine 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 zero.
- **shm_ctime** is set equal to the current time.

The *shmget* call fails if one or more of the following are true:

[EINVAL]	*size* is less than the system-imposed minimum or greater than the system-imposed maximum.
[EACCES]	A shared memory identifier exists for *key* but operation permission [see *intro*(2)] as specified by the low-order nine bits of *shmflg* would not be granted.

[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".
[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.
[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.
[EEXIST]	A shared memory identifier exists for *key* but [(*shmflg* & IPC_CREAT) and (*shmflg* & IPC_EXCL)] is "true".

SEE ALSO

intro(2), shmctl(2), shmop(2).

DIAGNOSTICS

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.

NOTES

The user must explicitly remove shared memory segments after the last reference to them has been removed.

NAME

shmop - shared memory operations

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

char *shmat (shmid, shmaddr, shmflg)
int shmid;
char *shmaddr;
int shmflg;

int shmdt (shmaddr)
char *shmaddr;

DESCRIPTION

shmat attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the data segment of the calling process. The segment is attached at the address specified by one of the following criteria:

- If *shmaddr* is equal to zero, the segment is attached at the first available address as selected by the system.
- If *shmaddr* is not equal to zero and (*shmflg* & SHM_RND) is "true", the segment is attached at the address given by [*shmaddr* - (*shmaddr* modulus SHMLBA)].
- If *shmaddr* is not equal to zero and (*shmflg* & SHM_RND) is "false", the segment is attached at the address given by *shmaddr*.

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 will fail and not attach the shared memory segment if one or more of the following are true:

[EINVAL]	*shmid* is not a valid shared memory identifier.
[EACCES]	Operation permission is denied to the calling process [see *intro*(2)].
[ENOMEM]	The available data space is not large enough to accommodate the shared memory segment.

[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.
[EMFILE]	The number of shared memory segments attached to the calling process would exceed the system-imposed limit.
[EINVAL]	*shmdt* will fail and not detach the shared memory segment if *shmaddr* is not the data segment start address of a shared memory segment.

NAME

shutdown - shut down part of a full-duplex connection

SYNOPSIS

int shutdown(s, how)
int s, how;

DESCRIPTION

The *shutdown* call causes all or part of a full-duplex connection on the socket associated with *s* to be shut down. If *how* is 0, further receives are disallowed. If *how* is 1, further sends are disallowed. If *how* is 2, further sends and receives are disallowed.

SEE ALSO

connect(2), socket(2).
CTIX Network Programmer's Primer.

DIAGNOSTICS

A 0 is returned if the call succeeds, -1 if it fails.

ERRORS

The call succeeds unless one of the following is true:

[EBADF]	*S* is not a valid descriptor.
[ENOTSOCK]	*S* is a file, not a socket.
[ENOTCONN]	The specified socket is not connected.

NAME

signal - specify what to do upon receipt of a signal

SYNOPSIS

#include <signal.h>

void (*signal (sig, func))()
int sig;
void (*func)();

DESCRIPTION

signal allows the calling process to choose one of three ways in which it is possible to handle the receipt of a specific signal. *sig* specifies the signal and *func* specifies the choice.

sig can be assigned any one of the following except **SIGKILL:**

SIGHUP	01	hangup
SIGINT	02	interrupt
SIGQUIT	03[1]	quit
SIGILL	04[1]	illegal instruction (not reset when caught)
SIGTRAP	05[1]	trace trap (not reset when caught)
SIGIOT	06[1]	IOT instruction
SIGEMT	07[1]	EMT instruction
SIGFPE	08[1]	floating point exception
SIGKILL	09	kill (cannot be caught or ignored)
SIGBUS	10[1]	bus error
SIGSEGV	11[1]	segmentation violation
SIGSYS	12[1]	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[2]	death of a child
SIGPWR	19[2]	power fail
SIGWIND	20	reserved
SIGPHONE	21	reserved
SIGPOLL	22[3]	selectable event pending

func is assigned one of three values: **SIG_DFL, SIG_IGN,** or a *function address*. SIG_DFL, and SIG_IGN, are defined in the include file *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.

The actions prescribed by the values of *func* are as follows:

SIG_DFL - terminate process upon receipt of a signal
: Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit*(2). See NOTE [1] below.

SIG_IGN - ignore signal
: The signal *sig* is to be ignored.

  Note that the signal **SIGKILL** cannot be ignored.

function address - catch signal
: Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function pointed to by *func*. The signal number *sig* will be passed as the only argument to the signal-catching function. 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 will be set to **SIG_DFL** unless the signal is **SIGILL**, **SIGTRAP**, or **SIGPWR**.

  Upon return from the signal-catching function, the receiving process will resume execution at the point it was interrupted.

  When a signal that is to be caught occurs during a *read*(2), a *write*(2), an *open*(2), or an *ioctl*(2) system call on a slow device (like a terminal; but not a file), during a *pause*(2) system call, or during a *wait*(2) system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal catching function will be executed and then the interrupted system call may return a -1 to the calling process with *errno* set to EINTR.

  signal will 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 signal **SIGKILL** cannot be caught.

A call to *signal* cancels a pending signal *sig* except for a pending **SIGKILL** signal.

signal will fail if *sig* is an illegal signal number, including **SIGKILL**. [EINVAL]

NOTES

[1] If **SIG_DFL** is assigned for these signals, in addition to the process being terminated, a "core image" will be 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 will have the following properties:
  - — a mode of 0666 modified by the file creation mask [see *umask*(2)]
  - — a file owner ID that is the same as the effective user ID of the receiving process.
  - — a file group ID that is the same as the effective group ID of the receiving process.

[2] For the signals **SIGCLD** and **SIGPWR** *func* is 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 will not create zombie processes when they terminate [see *exit*(2)].

function address - catch signal
If the signal is **SIGPWR**, the action to be taken is the same as that described above for *func* equal to *function address*. The same is true if the signal is **SIGCLD** with one exception: while the process is executing the signal-catching function, any received **SIGCLD** signals will be ignored. (This is the default action.)

In addition, **SIGCLD** affects the *wait*, 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* will block until all of the calling process's child processes terminate; it will then return a value of -1 with *errno* set to ECHILD.

exit If in the exiting process's parent process the *func* value of **SIGCLD** is set to **SIG_IGN**, the exiting process will not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the proceeding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set **SIGCLD** to be caught.

[3] **SIGPOLL** is issued when a file descriptor corresponding to a STREAMS [see *intro*(2)] 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 will never receive **SIGPOLL**.

SEE ALSO

kill(1), intro(2), kill(2), pause(2), ptrace(2), wait(2), setjmp(3C), sigset(2).

DIAGNOSTICS

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*.

NAME

sigset, sighold, sigrelse, sigignore, sigpause - signal management

SYNOPSIS

```
#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

These functions provide signal management for application processes. *sigset* 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. 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.

SIGHUP	hangup
SIGINT	interrupt
SIGQUIT*	quit
SIGILL*	illegal instruction (not held when caught)
SIGTRAP*	trace trap (not held when caught)
SIGABRT*	abort
SIGFPE*	floating point exception
SIGKILL	kill (can not be caught or ignored)
SIGSYS*	bad argument to system call
SIGPIPE	write on a pipe with no one to read it
SIGALRM	alarm clock
SIGTERM	software termination signal
SIGUSR1	user-defined signal 1
SIGUSR2	user-defined signal 2

SIGCLD	death of a child (see WARNING below)
SIGPWR	power fail (see WARNING below)
SIGPOLL	selectable event pending (see NOTES below)

See below under SIG_DFL regarding asterisks (*) in the above list.

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*(2). In addition a "core image" will be made in the current working directory of the receiving process if *sig* is one for which an asterisk appears in the above list *and* the following conditions are met:

- The effective user ID and the real user ID of the receiving process are equal.
- An ordinary file named *core* exists and is writable or can be created. If the file must be created, it will have the following properties:
  - A mode of 0666 modified by the file creation mask [see *umask* (2)].
  - A file owner ID that is the same as the effective user ID of the receiving process.
  - A file group ID that is the same as the effective group ID of the receiving process.

SIG_IGN - ignore signal

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 will call 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* will be passed as the only argument to the signal-catching handler. Before calling the signal-catching handler, the system signal action will be 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 *sigrelse* must be called to restore the system signal action and release any held signal of this type.

In general, upon return from the signal-catching handler, the receiving process will resume execution at the point it was interrupted. However, when a signal is caught during a *read*(2), a *write*(2), an *open*(2), or an *ioctl*(2) system call during a *sigpause* system call, or during a *wait*(2) system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal-catching handler will be executed and then the interrupted system call may return a -1 to the calling process with *errno* set to EINTR.

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*(2). 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* will fail if one or more of the following are 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*.

SEE ALSO

kill(2), pause(2), signal(2), wait(2), setjmp(3C).

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

SIGPOLL is issued when a file descriptor corresponding to a STREAMS [see *intro*(2)] file has a selectable event pending. A process must specifically request that this signal be sent using the I_SETSIG *ioctl*(2) call [see *streamio*(7)]. Otherwise, the process will never receive **SIGPOLL**.

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** can not be changed from the default system action.

Specific implementations 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. If an implementation can detect this condition, this signal type should be used.

The other signal management functions, *signal*(2) and *pause*(2), should not be used in conjunction with these routines for a particular signal type.

WARNING

Two signals that behave differently than the signals described above exist in this release of the system:

SIGCLD	death of a child (reset when caught)
SIGPWR	power fail (not reset when caught)

For these signals, *func* is assigned one of three values: SIG_DFL, SIG_IGN, or a *function address*. The actions prescribed by these values are as follows:

SIG_DFL - ignore signal
    The signal is to be ignored.

SIG_IGN - ignore signal
    The signal is to be ignored. Also, if *sig* is **SIGCLD**, the calling process's child processes will not create zombie processes when they terminate [see *exit*(2)].

function address - catch signal
    If the signal is **SIGPWR**, the action to be taken is the same as that described above for *func* equal to *function address*. The same is true if the signal is **SIGCLD** with one exception: while the process is executing the signal-catching function, any received **SIGCLD** signals will be ignored. (This is the default action.)

The **SIGCLD** affects two other system calls [*wait*(2), and *exit*(2)] in the following ways:

wait If the *func* value of **SIGCLD** is set to SIG_IGN and a *wait* is executed, the *wait* will block until all of the calling process's child processes terminate; it will then return a value of -1 with *errno* set to ECHILD.

exit If in the exiting process's parent process the *func* value of **SIGCLD** is set to SIG_IGN , the exiting process will not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the proceeding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set **SIGCLD** to be caught.

NAME

socket - create an endpoint for communication

SYNOPSIS

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

s = socket(domain, type, protocol)
int s, domain, type, protocol;

DESCRIPTION

The *socket* call creates an endpoint for communication and returns a descriptor; *s* is a file descriptor returned by the *socket* system call.

The *domain* parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. The protocol family generally is the same as the address family for the addresses supplied in later operations on the socket. These families are defined in the include file *<sys/socket.h>*. The only currently supported format is PF_INET (ARPA Internet protocols).

The socket has the indicated *type*, which specifies the semantics of communication. Currently defined types include:

SOCK_STREAM
SOCK_DGRAM
SOCK_RAW

Note that not all types are supported by all protocol families.

A SOCK_STREAM type provides sequenced, reliable, two-way connection-based byte streams with an out-of-band data transmission mechanism. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed maximum length).

SOCK_RAW sockets provide access to internal network protocols and interfaces. This type is available only to the super-user.

The *protocol* specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the "communication domain" in which communication is to take place; see *protocols*(4).

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. A stream socket must be in a *connected* state before any data may be sent or received on it. A connection to another socket is created with a *connect*(2) call.

Once connected, data may be transferred using *read*(2) and *write*(2) calls or some variant of the *send*(2) and *recv*(2) calls. When a session has been completed a *close*(2) may be performed. Out-of-band data may also be transmitted as described in *send*(2) and received as described in *recv*(2).

The communications protocols used to implement a SOCK_STREAM ensure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with -1 returns and with ETIMEDOUT as the specific code in the global variable *errno*. The protocols optionally keep sockets "warm" by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (e.g. 5 minutes). A SIGPIPE signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit.

SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in *send*(2) calls. Datagrams are generally received with *recv*(2), which returns the next datagram with its return address.

An *fcntl*(2) call can be used to specify a process group to receive a SIGUSR1 signal when the out-of-band data arrives.

The operation of sockets is controlled by socket level *options*. These options are defined in the file **<sys/socket.h>**. *setsockopt* and *getsockopt* [see *getsockopt*(2)] are used to set and get options, respectively. The following options are recognized at the socket level:

SO_DEBUG	Toggle recording of debugging information.
SO_REUSEADDR	Toggle on/off local address reuse.
SO_KEEPALIVE	Toggle keep connections alive.
SO_DONTROUTE	Toggle routing bypass for outgoing messages.
SO_LINGER	Linger on close if data present.
SO_BROADCAST	Toggle permission to transmit broadcast messages.
SO_OOBINLINE	Toggle reception of out-of-band data in band.
SO_SNDBUF	Set buffer size for output.
SO_RCVBUF	Set buffer size for input.

SO_TYPE	Get the type of the socket (get only).
SO_ERROR	Get and clear error on the socket (get only).

The options work as follows:

SO_DEBUG Enables debugging in the underlying protocol modules.

SO_REUSEADDR indicates that the rules used in validating addresses supplied in a *bind*(2) call should allow reuse of local addresses.

SO_KEEPALIVE enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and processes using the socket are notified via a SIGPIPE signal.

SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER controls the action taken when unsent messags are queued on socket and a *close*(2) is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system blocks the process on the *close* attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the *setsockopt* call when SO_LINGER is requested). If SO_LINGER is disabled and a *close* is issued, the system processes the close in a manner that allows the process to continue as quickly as possible.

SO_BROADCAST requests permission to send broadcast datagrams on the socket. Broadcast was a privileged operation in earlier versions of the system.

With protocols that support out-of-band data, SO_OOBINLINE requests that out-of-band data be placed in the normal data input queue as received; it is then accessible with *recv* or *read* calls without the MSG_OOB flag.

SO_SNDBUF and SO_RCVBUF are options to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. The system places an absolute limit on these values.

SO_TYPE and SO_ERROR are options used only with *setsockopt*. SO_TYPE returns the type of the socket, such as SOCK_STREAM; it is useful for servers that inherit sockets on startup. SO_ERROR returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

SEE ALSO

accept(2), bind(2), connect(2), getsockname(2), getsockopt(2), intro(2), ioctl(2), listen(2), read(2), recv(2), select(2), send(2), shutdown(2), write(2), inet(7), intro(7).
CTIX Network Programmer's Primer.

RETURN VALUE

A -1 is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

ERRORS

The *socket* call fails if:

[EPROTONOSUPPORT]	The protocol type or the specified protocol is not supported within this communication domain.
[EMFILE]	The per-process descriptor table is full.
[ENFILE]	The system file table is full.
[EACCESS]	Permission to create a socket of the specified type and/or protocol is denied.
[ENOSR]	Insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.

NAME

stat, fstat - get file status

SYNOPSIS

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

int stat (path, buf)
char *path;
struct stat *buf;

int fstat (fildes, buf)
int fildes;
struct stat *buf;
```

DESCRIPTION

Path points to a path name naming a file. Read, write, or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable. *stat* obtains information about the named file.

Note that in a Remote File Sharing environment, the information returned by *stat* depends upon the user/group mapping set up between the local and remote computers. [See *idload*(1M).]

fstat obtains information about an open file known by the file descriptor *fildes*, obtained from a successful *open*, *creat*, *dup*, *fcntl*, or *pipe* system call.

buf is a pointer to a *stat* structure into which information is placed concerning the file.

The contents of the structure pointed to by *buf* include the following members:

```
ushort  st_mode;    /* File mode [see mknod (2)] */
ino_t   st_ino;     /* Inode number */
dev_t   st_dev;     /* ID of device containing a directory entry for this file */
dev_t   st_rdev;    /* ID of device.  Defined only for character */
                    /* special or block special files */
short   st_nlink;   /* Number of links */
ushort  st_uid;     /* User ID of the file's owner */
ushort  st_gid;     /* Group ID of the file's group */
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*(2) 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*(2) 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*(2), *mknod*(2), *pipe*(2), *utime*(2), and *read*(2).
st_mtime	Time when data was last modified. Changed by the following system calls: *creat*(2), *mknod*(2), *pipe*(2), *utime*(2), and *write*(2).
st_ctime	Time when file status was last changed. Changed by the following system calls: *chmod*(2), *chown*(2), *creat*(2), *link*(2), *mknod*(2), *pipe*(2), *unlink*(2), *utime*(2), and *write*(2).

stat will fail if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EFAULT]	*buf* or *path* points to an invalid address.
[EINTR]	A signal was caught during the *stat* 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.

fstat will fail if one or more of the following are true:

[EBADF]	*fildes* is not a valid open file descriptor.
[EFAULT]	*buf* points to an invalid address.
[ENOLINK]	*fildes* points to a remote machine and the link to that machine is no longer active.

SEE ALSO

chmod(2), chown(2), creat(2), link(2), mknod(2), pipe(2), read(2), syslocal(2), time(2), unlink(2), utime(2), write(2).

DIAGNOSTICS

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

NAME

statfs, fstatfs - get file system information

SYNOPSIS

```
#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

The *statfs* call returns a ''generic super-block'' 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) to be filled by the system call, and *len* is the number of bytes of information the system should return in the structure. The value of *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 */
short   f_bsize;    /* Block size */
short   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* call is similar to the *statfs* call, except that the file named by *path* in *statfs* is instead identified by an open file descriptor *fildes* obtained from a successful *open*(2), *creat*(2), *dup*(2), *fcntl*(2), or *pipe*(2) system call.

The *statfs* call obsoletes *ustat*(2) and should be used in preference to it in new programs.

The *statfs* and *fstatfs* calls fail if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EFAULT]	*buf* or *path* points to an invalid address.
[EBADF]	*fildes* is not a valid open file descriptor.
[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)**.
[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

chmod(2), chown(2), creat(2), link(2), mknod(2), pipe(2), read(2), time(2), unlink(2), utime(2), write(2), fs(4).

NAME

stime - set time

SYNOPSIS

int stime (tp)
long *tp;

DESCRIPTION

stime sets the system's idea of the time and date. *tp* points to the value of time as measured in seconds from 00:00:00 GMT January 1, 1970.

[EPERM]	*stime* will fail if the effective user ID of the calling process is not super-user.

SEE ALSO

time(2).

DIAGNOSTICS

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

NAME

swrite - synchronous write on a file

SYNOPSIS

int swrite (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

DESCRIPTION

swrite has the same purpose and conventions as *write* (2). The two differ solely in their handling of disk input/output. *swrite*, unlike *write*, does not give a normal return before physical output is complete. A program that executes an *swrite* can assume that the data is on the disk, not waiting in a buffer pool.

SEE ALSO

creat(2), dup(2), lseek(2), open(2), pipe(2), ulimit(2).

NAME

sync - update super block

SYNOPSIS

void sync ( )

DESCRIPTION

The *sync* call causes all information in memory that should be on disk to be written out, including modified super blocks, modified i-nodes, and delayed block I/O.

The *sync* call should be used by programs that examine a file system: for example *fsck* and *df*. A call to *sync* is mandatory before a reboot.

The writing, although scheduled, is not necessarily complete upon return from *sync*.

NAME

sysfs - get file system type information

SYNOPSIS

```
#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

sysfs 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:

GETFSIND	translates *fsname*, a null-terminated file-system identifier, into a file-system type index.
GETFSTYP	translates *fs_index*, a file-system type index, into a null-terminated file-system identifier and writes it into the buffer pointed to by *buf*; this buffer must be at least of size **FSTYPSZ** as defined in <*sys/fstyp.h*>.
GETNFSTYP	returns the total number of file system types configured in the system.

sysfs will fail if one or more of the following are true:

[EINVAL]	*fsname* points to an invalid file-system identifier; *fs_index* is zero, or invalid; *opcode* is invalid.
[EFAULT]	*buf* or *fsname* point to an invalid user address.

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 **GETNFSTYP**. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

NAME

syslocal - special system requests

SYNOPSIS

#include <syslocal.h>
int syslocal (cmd [, arg] ...)
int cmd;

DESCRIPTION

The *syslocal* routine executes certain special system calls. The specific call is indicated by the first argument. See the **<sys/syslocal.h>** file for complete documentation of parameters.

System Type

int syslocal(SYSL_SYSTEM)

Return SYSL_MITI for S/Series.

Family Member

int syslocal(SYSL_FAMILYMEMBER)

Returns a value identifying the specific system: SYSLFMITI1 for S/120, S/221-2, or S/320; SYSLFMITI2 for S/480 or S/640; SYSLFS80 for S/80; and SYSLFS280 for S/280.

Superblock Resynchronization

int syslocal(SYSL_RESYNC, devnum)
short devnum

Reread contents of superblock from disk. *devnum* specifies the file system: the high order byte contains the major device number of the character special device; the low order byte contains the minor device number. Only the super-user can reread the contents of the superblock from disk.

Maximum Number of Users

int syslocal(SYSL_MAXUSERS)

Returns maximum number of users this system is configured for.

Kernel Addresses

syslocal(SYSL_KADDR, arg)

Returns certain addresses of kernel data structures. This allows certain programs (*ps*, *killall*) to run properly, even if **/unix** is not the currently running operating system.

arg is one of the following:

SLA_V	Return address of var structure (sys/var.h).
SLA_PROC	Return address of proc structure (sys/proc.h).
SLA_ERR	Return address of err structure (sys/err.h).
SCA_TIME	Return address of int time.
SLA_CDT	Return address of crash dump table (CDT) = (sys/hardware.h).
SLA_GDUTAB	Return address of gdutab (sys/iobuf.h).
SLA_USRSTK	Return highest address of user stack.
SLA_USIGN	Return signature of running UNIX (may be compared with that of /unix to see if they are identical).
SLA_MEM	Return number of bytes of physical memory.
SLA_BDEVCNT	Return the number of slots in struct bdevsw (sys/conf.h).
SLA_CDEVCNT	Return the number of slots in struct cdevsw (sys/conf.h).
SLA_PRELD	Return the address of the preloaded driver table.

Object Module Type

syslocal(SYSL_0413MAGIC)

Returns 1 if the kernel can support the **-F** option of *ld*().

Read Real-Time Clock

syslocal(SYSL_RDRTC, arg)

Read current state of real-time (battery supported) clock. *arg* is a pointer to struct **rtc** (sys/rtc.h)

Write Real-Time Clock

syslocal(SYSL_WTRTC, arg)

Write new state of real-time clock. *arg* is a pointer to a struct **rtc** (sys/rtc.h). EIO is returned if any of the values are illegal. Only the super-user can write the real-time clock.

Reboot System

syslocal(SYSL_REBOOT)

Force a software reset. Only the superuser may reset. Obsolete: retained for compatibility. Use *uadmin*(2) instead.

Allocate or Bind a Loadable Driver

syslocal(SYSL_ALLOCDRV, option, arg)

syslocal(SYSL_BINDDRV, option, arg)

These two functions implement the loadable driver functions of CTIX. They both require super-user privilege.

Loading drivers consists of two phases: allocation of virtual space, device numbers, and device IDs; and binding. Fully relocating a driver into memory, allocating physical space, plugging the device switch tables, calling initialization routines, and unloading require the same two phases in reverse. For information on the arguments, see **/usr/include/sys/drv.h**.

Determine Processor Type

syslocal(SYSL_PROCESSOR)

Returns a value that can be used to determine what kind of processor (68010 or 68020) is running and whether floating-point hardware (68881) is available.

Enable Fixed Priority Range

syslocal(SYSL_RTNICE,flag)

Enables/disables the fixed priority range [see *nice*(2)]. *flag* is 1 for enable, 2 for disable. Only the super-user can execute this call, which affects every process.

S/Series Hardware Configuration

syslocal(SYSL_MITICFIG)

Returns a bit mask of the hardware that is present. Values can be found in **syslocal.h.** A more convenient way to get this information is by using *hinv*(1M).

SEE ALSO

fsck(1M), nice(2).

DIAGNOSTICS

Note that *syslocal* fails if one of the following is true:

[EINVAL]	*cmd* or any suboption is illegal.
[EFAULT]	An *arg* points outside the process's space.

NAME

time - get time

SYNOPSIS

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

time_t time (tloc)
long *tloc;
```

DESCRIPTION

The *time* call returns the value of time in seconds since 00:00:00 GMT, January 1, 1970.

If *tloc* is non-zero, the return value is also stored in the location to which *tloc* points.

[EFAULT] *time* fails if *tloc* points to an illegal address.

SEE ALSO

stime(2).

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.

NAME

times - get process and child process times

SYNOPSIS

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

long times (buffer)
struct tms *buffer;

DESCRIPTION

The *times* call fills the structure pointed to by *buffer* with time-accounting information. The following are the contents of this structure:

```
struct  tms {
        time_t  tms_utime;
        time_t  tms_stime;
        time_t  tms_cutime;
        time_t  tms_cstime;

};
```

This information comes from the calling process and each of its terminated child processes for which it has executed a *wait*. All times are reported in clock ticks per second. Clock ticks are a system-dependent parameter. The specific value for an implementation is defined by the variable HZ, found in the include file param.h.

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_utime* s and *tms_cutime* s of the child processes.

tms_cstime
: Is the sum of the *tms_stime* s and *tms_cstime* s of the child processes.

The *times* call fails if the following is true:

[EFAULT]	*buffer* points to an illegal address.

SEE ALSO

exec(2), fork(2), time(2), wait(2).

DIAGNOSTICS

Upon successful completion, *times* returns the elapsed real time, in clock ticks per second, from an arbitrary point in the past (such as 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.

NAME

uadmin - administrative control

SYNOPSIS

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

int uadmin (cmd, fcn, mdep)
int cmd, fcn, mdep;
```

DESCRIPTION

Uadmin 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 *mdep* argument is provided for machine-dependent use and is not defined here.

As specified by *cmd*, the following commands are available:

A_SHUTDOWN
: Shut down the system. All user processes are killed, the buffer cache is flushed, and the root file system is unmounted. The *fcn* function specifies the action to be taken after the system is shut down. The functions are generic; the hardware capabilities vary on specific machines.

AD_HALT	Halt the processor so it is safe to turn off the power.
AD_BOOT	Reboot the system.

A_REBOOT
: Reboot the system immediately, without further processing.

A_REMOUNT
: The root file system is mounted again after having been fixed. This should be used only during the startup process.

A_HALT
: The system stops immediately.

uadmin fails if any of the following are true:

[EPERM]	The effective user ID is not super-user.

SEE ALSO

syslocal(2).

DIAGNOSTICS

Upon successful completion, the value returned depends on *cmd* as follows:

A_SHUTDOWN	Never returns.
A_REBOOT	Never returns.
A_REMOUNT	0
A_HALT	Never returns.

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

NAME

ulimit - get and set user limits

SYNOPSIS

long ulimit (cmd, newlimit)
int cmd;
long newlimit;

DESCRIPTION

This function provides for control over process limits. The *cmd* values follow:

1 Get the regular file size limit of the process. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.

2 Set the regular file size limit of the process to the value of *newlimit*. Any process may decrease this limit, but only a process with an effective user ID of super-user may increase the limit. *ulimit* fails and the limit is unchanged if a process with an effective user ID other than super-user attempts to increase its regular file size limit. [EPERM]

3 Get the maximum possible break value [see *brk*(2)].

4 Get the current value of the maximum number of open files per process configured in the system.

SEE ALSO

brk(2), write(2).

DIAGNOSTICS

Upon successful completion, a non-negative value is returned; otherwise, a value of -1 is returned and *errno* is set to indicate the error.

WARNING

The *ulimit* call is effective in limiting the growth of regular files. Pipes are currently limited to 9,216 bytes (this is the maximum atomic write size).

NAME

umask - set and get file creation mask

SYNOPSIS

int umask (cmask)
int cmask;

DESCRIPTION

The *umask* call sets the process's file mode creation mask to *cmask* and returns the previous value of the mask. Only the low-order 9 bits of *cmask* and the file mode creation mask are used.

SEE ALSO

mkdir(1), sh(1), chmod(2), creat(2), mknod(2), open(2).

DIAGNOSTICS

The previous value of the file mode creation mask is returned.

NAME

umount - unmount a file system

SYNOPSIS

int umount (file)
char *file;

DESCRIPTION

The *umount* call requests that a previously mounted file system contained on the block special device or directory identified by *file* be unmounted. The *file* parameter 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* call can be invoked only by the super-user.

The *umount* call fails if one or more of the following are true:

[EPERM]	The process's effective user ID is not super-user.
[EINVAL]	*file* does not exist.
[ENOTBLK]	*file* is not a block special device.
[EINVAL]	*file* is not mounted.
[EBUSY]	A file on *file* is busy.
[EFAULT]	*file* points to an illegal address.
[EREMOTE]	*file* is remote.
[ENOLINK]	*file* is on a remote machine, and the link to that machine is no longer active.
[EMULTIHOP]	Components of the path pointed to by *file* require hopping to multiple remote machines.

SEE ALSO

mount(2).

DIAGNOSTICS

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

NAME

uname - get name of current CTIX system

SYNOPSIS

#include <sys/utsname.h>

int uname (name)
struct utsname *name;

DESCRIPTION

The *uname* call stores information identifying the current CTIX system in the structure pointed to by *name*.

The call uses the structure defined in **<sys/utsname.h>**, whose members follow:

```
char    sysname[9];
char    nodename[9];
char    release[9];
char    version[9];
char    machine[9];
```

The *uname* call returns a null-terminated character string naming the current CTIX system in the character array *sysname*. Similarly, *nodename* contains the name that the system is known by on a communications network. [Note the equivalence of *nodename* and the left-most qualifier in a full Internet name; see *hostname*(1).] The *release* and *version* members further identify the operating system; *machine* contains a standard name that identifies the hardware that the CTIX system is running on.

[EFAULT]	*uname* fails if *name* points to an invalid address.

SEE ALSO

hostname(1), setuname(1M), uname(1), sethostname (2).

DIAGNOSTICS

Upon successful completion, a non-negative value is returned; otherwise, a value of -1 is returned and *errno* is set to indicate the error.

NAME

unlink - remove directory entry

SYNOPSIS

int unlink (path)
char *path;

DESCRIPTION

unlink removes the directory entry named by the path name pointed to by *path*.

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

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	Write permission is denied on the directory containing the link to be removed.
[EACCES]	The parent directory has the sticky bit set and: the parent directory is not owned by the user; and the directory is not owned by the user; and the directory is not writable by the user; and the user is not super-user.
[EPERM]	The named file is a directory and the effective user ID of the process is not super-user.
[EBUSY]	The entry to be unlinked is the mount point for a mounted file system.
[ETXTBSY]	The entry to be unlinked is the last link to a pure procedure (shared text) file that is being executed.
[EROFS]	The directory entry to be unlinked is part of a read-only file system.
[EFAULT]	*Path* points outside the process's allocated address space.
[EINTR]	A signal was caught during the *unlink* 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.

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.

SEE ALSO

rm(1), close(2), link(2), open(2).

DIAGNOSTICS

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

NAME

ustat - get file system statistics

SYNOPSIS

```
#include <sys/types.h>
#include <ustat.h>

int ustat (dev, buf)
dev_t dev;
struct ustat *buf;
```

DESCRIPTION

The *ustat* 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 *ustat* call fails if one or more of the following are true:

[EINVAL]	*dev* is not the device number of a device containing a mounted file system.
[EFAULT]	*buf* points outside the process's allocated address space.
[EINTR]	A signal was caught during a *ustat* system call.
[ENOLINK]	*dev* is on a remote machine and the link to that machine is no longer active.
[ECOMM]	*dev* is on a remote machine and the link to that machine is no longer active.

SEE ALSO

stat(2), statfs(2), fs(4).

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.

NOTE

The *statfs* call obsoletes *ustat*(2) and should be used in preference to *ustat*(2) in new programs.

NAME

utime - set file access and modification times

SYNOPSIS

```
#include <sys/types.h>
int utime (path, times)
char *path;
struct utimbuf *times;
```

DESCRIPTION

The *path* parameter points to a path name naming a file. The *utime* call sets the access and modification times of the named file.

If *times* is NULL, the access and modification times of the file are set to the current time. A process must be the owner of the file or have write permission to use *utime* in this manner.

If *times* is not NULL, *times* is interpreted as a pointer to a *utimbuf* structure and the access and modification times are set to the values contained in the designated structure. Only the owner of the file or the super-user can use *utime* this way.

The times in the following structure are measured in seconds since 00:00:00 GMT, Jan. 1, 1970.

```
struct    utimbuf  {
          time_t   actime;      /* access time */
          time_t   modtime;     /* modification time */
};
```

The *utime* call fails if one or more of the following are true:

[ENOENT]	The named file does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[EACCES]	Search permission is denied by a component of the path prefix.
[EPERM]	The effective user ID is not super-user and not the owner of the file and *times* is not NULL.
[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.
[EROFS]	The file system containing the file is mounted read-only.
[EFAULT]	*times* is not NULL and points outside the process's allocated address space.

[EFAULT]	*path* points outside the process's allocated address space.
[EINTR]	A signal was caught during the *utime* 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.

SEE ALSO

stat(2).

DIAGNOSTICS

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

NAME

wait - wait for child process to stop or terminate

SYNOPSIS

int wait (stat_loc)
int *stat_loc;

DESCRIPTION

The *wait* call suspends the calling process until until one of the immediate children terminates or until a child that is being traced stops, because it has hit a break point. The *wait* system call returns prematurely if a signal is received and if a child process stopped or terminated prior to the call on *wait*, return is immediate.

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*(2)].
- 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 (bit 200) is set, a "core image" will have been produced [see *signal*(2)].

If a parent process terminates without waiting for its child processes to terminate, the parent process ID of each child process is set to 1. This means the initialization process inherits the child processes [see *intro*(2)].

The *wait* call fails and returns immediately if the following is true:

[ECHILD]	The calling process has no existing unwaited-for child processes.

SEE ALSO

exec(2), exit(2), fork(2), intro(2), pause(2), ptrace(2), signal(2).

DIAGNOSTICS

If *wait* returns due to the receipt of a signal, a value of -1 is returned to the calling process and *errno* is set to EINTR. If *wait* returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process; otherwise, a value of -1 is returned and *errno* is set to indicate the error.

WARNING

The *wait* call fails and its actions are undefined if *stat_loc* points to an invalid address.

See *WARNING* in *signal*(2).

NAME

write - write on a file

SYNOPSIS

int write (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

DESCRIPTION

The *fildes* argument is a file descriptor obtained from a *creat*(2), *open*(2), *dup*(2), *fcntl*(2), or *pipe*(2) system call.

The *write* call attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the *fildes*.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from *write*, the file pointer is incremented by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

If the O_APPEND flag of the file status flags is set, the file pointer will be set to the end of the file prior to each write.

For regular files, if the O_SYNC flag of the file status flags is set, the write will not return until both the file data and file status have been physically updated. This function is for special applications that require extra reliability at the cost of performance. For block special files, if O_SYNC is set, the write will not return until the data has been physically updated.

A write to a regular file will be blocked if mandatory file/record locking is set [see *chmod*(2)], and there is a record lock owned by another process on the segment of the file to be written. If O_NDELAY is not set, the write sleeps until the blocking record lock is removed.

For STREAMS [see *intro*(2)] files, the operation of *write* is determined by the values of the minimum and maximum *nbyte* range (packet size) accepted by the *stream*. These values are contained in the topmost *stream* module. Unless the user pushes [see I_PUSH in *streamio*(7)] the topmost module, these values can not be set or tested from user level. If *nbyte* falls within the packet size range, *nbyte* bytes will be written. If *nbyte* does not fall within the range and the minimum packet size value is zero, *write* will break the buffer into maximum packet size segments prior to sending the data downstream (the last segment may contain less than the maximum packet size). If *nbyte* does not fall within the range and the minimum value is non-zero, *write* will fail with *errno* set to ERANGE. Writing a zero-length buffer (*nbyte* is zero) sends zero bytes with zero returned.

For STREAMS files, if O_NDELAY is not set and the *stream* can not accept data (the *stream* write queue is full due to internal flow control conditions), *write* will block until data can be accepted. O_NDELAY will prevent a process from blocking due to flow control conditions. If O_NDELAY is set and the *stream* can not accept data, *write* will fail. If O_NDELAY is set and part of the buffer has been written when a condition in which the *stream* can not accept additional data occurs, *write* will terminate and return the number of bytes written.

write will fail and the file pointer will remain unchanged if one or more of the following are true:

[EAGAIN]	Mandatory file/record locking was set, O_NDELAY was set, and there was a blocking record lock.
[EAGAIN]	Total amount of system memory available when reading via raw I/O is temporarily insufficient.
[EAGAIN]	Attempt to write to a *stream* that can not accept data with the O_NDELAY 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.
[EFAULT]	*buf* points outside the process's allocated address space.
[EFBIG]	An attempt was made to write a file that exceeds the process's file size limit or the maximum file size [see *ulimit*(2)].
[EINTR]	A signal was caught during the *write* system call.
[EINVAL]	Attempt to write to a *stream* linked below a multiplexor.
[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.

[ENXIO]	A hangup occurred on the *stream* being written to.
[EPIPE and SIGPIPE signal]	An attempt is made to write to a pipe that is 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.
[EDEADLOCK]	A side effect of the *locking*(2) call. (See the WARNING on the *locking*(2) manpage.)

If a *write* requests that more bytes be written than there is room for (e.g., the *ulimit* [see *ulimit*(2)] or the physical end of a medium), only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512-bytes will return 20. The next write of a non-zero number of bytes will give a failure return (except as noted below).

If the file being written is a pipe (or FIFO) and the O_NDELAY flag of the file flag word is set, then write to a full pipe (or FIFO) will return a count of 0. Otherwise (O_NDELAY clear), writes to a full pipe (or FIFO) will block until space becomes available.

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.

SEE ALSO

creat(2), dup(2), fcntl(2), intro(2), lseek(2), open(2), pipe(2), ulimit(2).

DIAGNOSTICS

Upon successful completion the number of bytes actually written is returned. Otherwise, -1 is returned and *errno* is set to indicate the error.

NAME

intro - introduction to functions and libraries

DESCRIPTION

This section describes functions found in various libraries, other than those functions that directly invoke CTIX system primitives, which are described in Section 2 of this volume. Certain major collections are identified by a letter after the section number:

(3C) These functions, together with those of Section 2 and those marked (3S), constitute the Standard C Library *libc*, which is automatically loaded by the C compiler, *cc*(1). The link editor *ld*(1) searches this library under the **-lc** option. A ''shared library'' version of *libc* can be searched using the **-lc_s** option, resulting in smaller *a.outs*. Declarations for some of these functions may be obtained from **#include** files indicated on the appropriate pages.

(3S) These functions constitute the ''standard I/O package'' [see *stdio*(3S)]. These functions are in the library *libc*, already mentioned. Declarations for these functions may be obtained from the **#include** file **<stdio.h>**.

(3M) These functions constitute the Math Library, *libm*. They are not automatically loaded by the C compiler, *cc*(1); however, the link editor searches this library under the **-lm** option. Declarations for these functions may be obtained from the **#include** file **<math.h>**. Several generally useful mathematical constants are also defined there [see *math*(5)].

(3N) This contains sets of functions constituting the Network Services library. These sets provide protocol-independent interfaces to networking services based on the service definitions of the OSI (Open Systems Interconnection) reference model. Application developers access the function sets that provide services at a particular level.

This library contains the functions of the TRANSPORT INTERFACE (TI) - provide the services of the OSI Transport Layer. These services provide reliable end-to-end data transmission using the services of an underlying network. Applications written using the TI functions are independent of the underlying protocols. Declarations for these functions may be obtained from the **#include** file **<tiuser.h>**. The link editor *ld*(1) searches this library under the **-lnsl_s** option.

(3X) Various specialized libraries. The files in which these libraries are found are given on the appropriate pages.

DEFINITIONS

A *character* is any bit pattern able to fit into a byte on the machine. The *null character* is a character with value 0, represented in the C language as '\0'. A *character array* is a sequence of characters. A *null-terminated character array* is a sequence of characters, the last of which is the *null character*. A *string* is a designation for a *null-terminated character array*. The *null string* is a character array containing only the null character. A NULL pointer is the value that is obtained by casting **0** into a pointer. The C language guarantees that this value will not match that of any legitimate pointer, so many functions that return pointers return it to indicate an error. NULL is defined as **0** in **<stdio.h>**; the user can include an appropriate definition if not using **<stdio.h>**.

Netbuf

In the Network Services library, *netbuf* is a structure used in various Transport Interface (TI) functions to send and receive data and information. It contains the following members:

```
unsigned int maxlen;
unsigned int len;
char      *buf;
```

buf points to a user input and/or output buffer. *len* generally specifies the number of bytes contained in the buffer. If the structure is used for both input and output, the function will replace the user value of *len* on return.

maxlen generally has significance only when *buf* is used to receive output from the TI function. In this case, it specifies the physical size of the buffer, the maximum value of *len* that can be set by the function. If *maxlen* is not large enough to hold the returned information, an TBUFOVFLW error will generally result. However, certain functions may return part of the data and not generate an error.

Note that a *struct sockaddr* goes in all ''addr'' TLI *netbufs*.

FILES

/lib
/lib/libc.a
/lib/libc_s.a
/lib/libm.a
/shlib/libc1sw_s
/shlib/libc2sw_s
/shlib/libc2fp_s
/shlib/libnsl1sw_s
/shlib/libnsl2sw_s
/shlib/libnsl2fp_s
/usr/lib/libnsl_s.a

SEE ALSO

ar(1), cc(1), ld(1), lint(1), nm(1), intro(2), stdio(3S), math(5).

DIAGNOSTICS

Functions in the C and Math Libraries (3C and 3M) may return the conventional values **0** or ±**HUGE** (the largest-magnitude single-precision floating-point numbers; **HUGE** is defined in the **<math.h>** header file) when the function is undefined for the given arguments or when the value is not representable. In these cases, the external variable *errno* [see *intro*(2)] is set to the value EDOM or ERANGE.

WARNING

Many of the functions in the libraries call and/or refer to other functions and external variables described in this section and in Section 2 (*System Calls*). If a program inadvertently defines a function or external variable with the same name, the presumed library version of the function or external variable may not be loaded. The *lint*(1) program checker reports name conflicts of this kind as ''multiple declarations'' of the names in question. Definitions for Sections 2, 3C, and 3S are checked automatically. Other definitions can be included by using the **-l** option. (For example, **-l***m* includes definitions for Section 3M, the Math Library.) Use of *lint* is highly recommended.

NAME

a64l, l64a - convert between long integer and base-64 ASCII string

SYNOPSIS

```
long a64l (s)
char *s;

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

DESCRIPTION

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

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

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

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

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

CAVEAT

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

NAME

abort - generate a SIGABRT

SYNOPSIS

int abort ( )

DESCRIPTION

The *abort* routine does the work of *exit*(2), but instead of just exiting, *abort* causes **SIGABRT** to be sent to the calling process. If **SIGABRT** is neither caught nor ignored, all *stdio*(3S) streams are flushed prior to the signal being sent, and a core dump results.

The *abort* routine returns the value of the *kill*(2) system call.

SEE ALSO

sdb(1), exit(2), kill(2), signal(2).

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.

NAME

abs - return integer absolute value

SYNOPSIS

int abs (i)
int i;

DESCRIPTION

The *abs* routine returns the absolute value of its integer operand.

SEE ALSO

floor(3M).

CAVEAT

In two's-complement representation, the absolute value of the negative integer with largest magnitude is undefined. Some implementations trap this error, but others simply ignore it.

NAME

assert - verify program assertion

SYNOPSIS

#include <assert.h>

assert (expression)
int expression;

DESCRIPTION

This macro is useful for putting diagnostics into programs. When it is executed, if *expression* is false (zero), *assert* prints

Assertion failed: *expression*, file *xyz*, line *nnn*

on the standard error output and aborts. In the error message, *xyz* is the name of the source file and *nnn* the source line number of the *assert* statement.

Compiling with the preprocessor option **-DNDEBUG** [see *cpp* (1)], or with the preprocessor control statement **#define NDEBUG** ahead of the **#include <assert.h>** statement, will stop assertions from being compiled into the program.

SEE ALSO

cpp(1), abort(3C).

CAVEAT

Since *assert* is implemented as a macro, the *expression* may not contain any string literals.

NAME

bessel: j0, j1, jn, y0, y1, yn - Bessel functions

SYNOPSIS

```
#include <math.h>

double j0 (x)
double x;

double j1 (x)
double x;

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

double y0 (x)
double x;

double y1 (x)
double x;

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

DESCRIPTION

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

y0 and *y1* return 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.

SEE ALSO

matherr(3M).

DIAGNOSTICS

Non-positive arguments cause *y0*, *y1* and *yn* to return the value **-HUGE** 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 can be changed with the function *matherr*(3M).

NAME

bsearch - binary search a sorted table

SYNOPSIS

#include <search.h>

char *bsearch ((char *) key, (char *) base, nel, sizeof (*key), compar)
unsigned nel;
int (*compar)( );

DESCRIPTION

The *bsearch* routine is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table indicating where a datum can 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 as accordingly the first argument is to be considered less than, equal to, or greater than the second.

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 the string and its length, or prints an error message.

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

#define    TABSIZE         1000

struct node {                  /* these are stored in the table */
           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));
}
```

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(3C), lsearch(3C), qsort(3C), tsearch(3C).

DIAGNOSTICS

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

NAME

bcopy, bcmp, bzero - bit and byte string operations

SYNOPSIS

```
int bcopy(src, dst, length)
char *src, *dst;
int length;

int bcmp(b1, b2, length)
char *b1, *b2;
int length;

int bzero(b, length)
char *b;
int length;
```

DESCRIPTION

The functions *bcopy*, *bcmp*, and *bzero* operate on variable length strings of bytes. They do not check for null bytes as the routines in *string*(3) do.

The *bcopy* routine copies *length* bytes from string *src* to the string *dst*.

The *bcmp* routine compares byte string *b1* against byte string *b2*, returning zero if they are identical, non-zero otherwise. Both strings are assumed to be *length* bytes long.

bzero places *length* 0 bytes in the string *b1*.

WARNING

The *bcopy* routine take parameters backwards from *strcpy*.

NAME

htonl, htons, ntohl, ntohs - convert values between host and network byte order

SYNOPSIS

```
#include <sys/types.h>
#include <sys/in.h>

netlong = htonl(hostlong);
unsigned long netlong, hostlong;

netshort = htons(hostshort);
ushort netshort, hostshort;

hostlong = ntohl(netlong);
unsigned long hostlong, netlong;

hostshort = ntohs(netshort);
ushort hostshort, netshort;
```

DESCRIPTION

These routines convert 16- and 32-bit quantities between network byte order and host byte order. These routines are defined as null macros in the include file <*sys/in.h*> (that is, network byte order is native 680x0 order).

These routines are most often used in conjunction with Internet addresses and ports as returned by *gethostent*(3) and *getservent*(3).

SEE ALSO

gethostbyname(3), getservent(3).
CTIX Network Programmer's Primer.

NAME

clock - report CPU time used

SYNOPSIS

long clock ( )

DESCRIPTION

The *clock* routine returns the amount of CPU time (in microseconds) used since the first call to *clock*. The time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed *wait*(2), *pclose*(3S), or *system*(3S).

The resolution of the clock is 1/HZ seconds on CTIX processors (HZ is defined in <*sys/param.h*>).

SEE ALSO

times(2), wait(2), popen(3S), system(3S).

BUGS

The value returned by *clock* is defined in microseconds for compatibility with systems that have CPU clocks with much higher resolution. Because of this, the value returned wraps around after accumulating only 2147 seconds of CPU time (about 36 minutes). The value then accumulates to -2148 and finally wrap around again to 0.

NAME

conv: toupper, tolower, _toupper, _tolower, toascii - translate characters

SYNOPSIS

#include <ctype.h>

int toupper (c)
int c;

int tolower (c)
int c;

int _toupper (c)
int c;

int _tolower (c)
int c;

int toascii (c)
int c;

DESCRIPTION

The *toupper* and *tolower* routines have as domain the range of *getc*(3S): the integers from -1 through 255. 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 lower-case letter. All other arguments in the domain are returned unchanged.

The macros *_toupper* and *_tolower*, accomplish the same thing as *toupper* and *tolower* but have restricted domains and are faster. *_toupper* requires a lowercase letter as its argument; its result is the corresponding uppercase letter. The macro *_tolower* requires an uppercase letter as its argument; its result is the corresponding lower-case letter. Arguments outside the domain cause undefined results.

The *toascii* routine yields its argument with all bits turned off that are not part of a standard ASCII character; it is intended for compatibility with other systems.

SEE ALSO

ctype(3C), getc(3S).

NAME

crypt, setkey, encrypt - generate hashing encryption

SYNOPSIS

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

void setkey (key)
char *key;

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

DESCRIPTION

The *crypt* routine performs password encryption. 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.

The *key* argument is a user's typed password. The *salt* argument is a two-character string chosen from the set [**a-zA-Z0-9./**]; this string is used to perturb the hashing algorithm in one of 4,096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password. The first two characters are the salt itself.

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

The argument to the *encrypt* entry is a character array of length 64 containing only the characters with numerical value 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the hashing algorithm using the key set by *setkey*. *Ignored* is unused by *encrypt* but it must be present.

SEE ALSO

login(1), passwd(1), getpass(3C), passwd(4).

CAVEAT

The return value points to static data that are overwritten by each call.

WARNING

The standard CTIX distribution is the international version, which does not support encryption.

NAME

crypt - password and file encryption functions

SYNOPSIS

```
cc [flag ...] file ... -lcrypt

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

void setkey (key)
char *key;

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

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

void des_setkey (key)
char *key;

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

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];
```

DESCRIPTION

The *des_crypt* routine performs password encryption. 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.

The *key* argument is a user's typed password. The *salt* argument is a two-character string chosen from the set **[a-zA-Z0-9./]**; 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 *flag* is zero, the argument is encrypted; if non-zero, it is decrypted.

Note that decryption is not provided in the international version of *crypt*(3X). If decryption is attempted with the international version of *des_encrypt*, an error message is printed.

The *crypt*, *setkey*, and *encrypt* routines 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*(1) and *vi*(1)] that must be compatible with the *crypt*(1) user-level utility. *run_setkey* establishes a two-way pipe connection with *crypt*(1), using *key* as the password argument. The *run_crypt* routine takes a block of characters and transforms the cleartext or ciphertext into their ciphertext or cleartext using *crypt*(1). The *offset* argument is the relative byte position from the beginning of the file that the block of text provided in *block* is coming from; *count* is the number of characters in *block*, and *connection* is an array containing indices to a table of input and output file streams. When encryption is finished, *crypt_close* is used to terminate the connection with *crypt*(1).

The *run_setkey* routine returns -1 if a connection with *crypt*(1) cannot be established; this occurs on international versions of CTIX where *crypt*(1) is not available. If a null key is passed to *run_setkey*, 0 is returned; otherwise, 1 is returned. The *run_crypt* routine returns -1 if it cannot write output or read input from the pipe attached to *crypt*; otherwise, it returns 0.

SEE ALSO

crypt(1), login(1), passwd(1), getpass(3C), passwd(4).

DIAGNOSTICS

In the international version of *crypt*(3X), a flag argument of 1 to *des_encrypt* is not accepted, and an error message is printed.

CAVEAT

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

WARNING

The standard CTIX distribution is the international version, which does not support encryption.

NAME

ctermid - generate file name for terminal

SYNOPSIS

#include <stdio.h>
char *ctermid (s)
char *s;

DESCRIPTION

The *ctermid* routine 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.

SEE ALSO

ttyname(3C).

NOTES

The difference between *ctermid* and *ttyname*(3C) is that *ttyname* must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while *ctermid* returns a string (**/dev/tty**) that 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.

NAME

ctime, localtime, gmtime, asctime, cftime, ascftime, tzset - convert date and time to string

SYNOPSIS

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

char *ctime (clock)
time_t *clock;

struct tm *localtime (clock)
time_t *clock;

struct tm *gmtime (clock)
time_t *clock;

char *asctime (tm)
struct tm *tm;

int cftime(buf, fmt, clock)
char *buf, *fmt;
time_t *clock;

int ascftime (buf, fmt, tm)
char *buf, *fmt;
struct tm *tm;

extern long timezone, altzone;

extern int daylight;

extern char *tzname[2];

void tzset ( )
```

DESCRIPTION

The *ctime*, *localtime*, and *gmtime* routines accept arguments of type **time_t** (declared in **<sys/types.h>**), pointed to by *clock*, representing the time in seconds since 00:00:00 GMT, January 1, 1970. *ctime* returns a pointer to a 26-character string in the following form. All the fields have constant width.

```
Fri Sep 13 00:00:00 1986\n\0
```

The *localtime* and *gmtime* routines 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 Greenwich Mean Time (GMT), which is the time the UNIX system uses.

The *asctime* routine converts a **tm** structure to a 26-character string, as shown in the above example, and returns a pointer to the string.

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

```
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 */
};
```

If the alternate time zone is in effect, *tm_isdst* is non-zero.

The *cftime* and *ascftime* routines provide the capabilities of *ctime* and *asctime*, respectively, as well as additional ones. The *cftime* routine takes an integer of type *time_t* pointed to by *clock* and converts it to a character string; *ascftime* takes a pointer to a **tm** structure and converts it to a character string. In both functions, the characters are placed into the array pointed to by *buf* (plus a terminating \0) and the value returned is the number of such characters (not counting the terminating \0). The *fmt* argument controls the format of the resulting string; it is a character string that consists of field descriptors and text characters, reminiscent of *printf*(3S). Each field descriptor consists of a % character followed by another character which specifies the replacement for the field descriptor. All other characters are copied from *fmt* into the result. The following field descriptors are supported:

%%	same as %
%a	abbreviated weekday name
%A	full weekday name
%b	abbreviated month name
%B	full month name
%d	day of month (01 - 31)
%D	date as %m/%d/%y
%e	day of month (1-31; single digits are preceded by a blank)
%h	abbreviated month name
%H	hour (00 - 23)

%I	hour (00 - 12)
%j	day number of year (001 - 366)
%m	month number (01 - 12)
%M	minute (00 - 59)
%n	same as \n
%p	ante meridian or post meridian
%r	time as %I:%M:%S %p
%R	time as %H:%M
%S	seconds (00 - 59)
%t	insert a tab
%T	time as %H:%M:%S
%U	week number of year (01 - 52), Sunday is the first day of week
%w	weekday number (Sunday = 0)
%W	week number of year (01 - 52), Monday is the first day of week
%x	Local specific date format
%X	Local specific time format
%y	year within century (00 - 99)
%Y	year as ccyy (for example, 1986)
%Z	time zone name

The difference between **%U** and **%W** lies in which day is counted as the first of the week. Week number 01 is the first week with four or more January days in it.

The example below shows what the values in the **tm** structure would look like for Thursday, August 28, 1986 at 12:44:36 in New Jersey.

```
ascftime (buf, "%A %h %d %j", tm)
```

This example would result in the buffer containing Thursday Aug 28 240.

If *fmt* is (char *)0, the value of the environment variable CFTIME is used. If CFTIME is undefined or empty, a default format is used. The default format string is taken from the file that contains the date and time strings associated with the then current language [see below for details on changing the current language and *cftime*(4) for a description of the structure of these files].

The user can request that the output of *cftime* and *ascftime* be in a specific language by setting the environment variable LANGUAGE to the desired language. If LANGUAGE is empty, unset or set to an unsupported language, the last language requested will be used (the default is the **usa_english** strings).

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 as follows:

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

The functions know about the peculiarities of this conversion for various time periods for the U.S.A (specifically, the years 1974, 1975, and 1987). The functions will handle the new daylight savings time starting with the first Sunday in April, 1987.

The *tzset* routine uses the contents of the environment variable **TZ** to override the value of the different external variables. The syntax of **TZ** can be described as follows:

```
TZ → zone
        | zone signed_time
        | zone signed_time zone
        | zone signed_time zone dst
zone → letter letter letter
signed_time → sign time
        | time
time → hour
        | hour : minute
        | hour : minute :
dst → signed_time
        | signed_time ; dst_date ,
        | ; dst_date , dst_date
dst_date → julian
        | julian / time
letter      →        a | A | b | B | ... | z | Z
hour        →        00 | 01 | ... | 23
minute      →        00 | 01 | ... | 59
second      →        00 | 01 | ... | 59
julian      →        001 | 002 | ...| 366
sign        →        − | +
```

tzset scans the contents of the environment variable and assigns the different fields to the respective variable. For example, the setting for New Jersey in 1986 could be either of the following:

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

HBEST5EDT

A southern hemisphere setting such as the Cook Islands could be the following:

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

When the longer format is used, the variable must be surrounded by double quotes as shown. For more details, see *timezone*(4) and *environ*(5). In the longer version of the New Jersey example of TZ, *tzname[0]* is EST, *timezone* will be set to 5*60*60, *tzname[1]* is EDT, *altzone* will be 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* will be set to non-zero. Starting and ending times are relative to the alternate time zone. If the alternate time zone start and end dates and the time are not provided, the days for the United States that year will be used and the time will be 2 AM. If the start and end dates are provided but the time is not provided, the time will be midnight. 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 can also be called explicitly by the user.

FILES

/lib/cftime - directory that contains the language specific printable files

SEE ALSO

time(2), getenv(3C), printf(3S), putenv(3C), cftime(4), timezone(4), environ(5).

CAVEAT

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.

NAME

isdigit, isxdigit, islower, isupper, isalpha, isalnum, isspace, iscntrl, ispunct, isprint, isgraph, isascii, tolower, toupper, toascci, _tolower, _toupper, setchrclass - character handling

SYNOPSIS

#include <ctype.h>

int isdigit (c);
int c;

. . .

tolower(c)
int c;

. . .

int setchrclass (chrclass)
char *chrclass;

DESCRIPTION

The character classification macros listed below return nonzero for true, zero for false. *isascii* is defined on all integer values; the rest are defined on valid members of the character set and on the single value **EOF** [see *stdio*(3S)] (guaranteed not to be a character set member).

isdigit	Tests for the digits 0 through 9.
isxdigit	Tests for any character for which *isdigit* is true or for the letters *a* through *f* or *A* through *F*.
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 or space.

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 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, *toupper* returns the 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.

The *setchrclass* routine itializes the table used by these functions and macros to a specific character classification set. It uses the value of its argument or the value of the environment variable **CHRCLASS** as the name of the datafile containing the information for the desired character set. These datafiles are searched for in the special directory /lib/chrclass.

If *chrclass* is (char *)0, the value of the environment variable **CHRCLASS** is used. If **CHRCLASS** is not set or is undefined, the table retains its current value, which at initialization time is **ascii**.

FILES

/lib/chrclass - directory containing the datafiles for *setchrclass*

SEE ALSO

ascii(5), chrtbl(1), environ(5), stdio(3S).

DIAGNOSTICS

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

If *setchrclass* does not successfully fill the table, the table will not change (initially ``ascii'') and -1 is returned. If everything works, *setchrclass* returns 0.

NAME

curses - terminal screen handling and optimization package

OVERVIEW

The *curses* manual page is organized as follows:

SYNOPSIS:

- Compiling information
- Summary of parameters used by *curses* routines
- Alphabetical list of *curses* routines, showing parameters

DESCRIPTION:

An overview of how *curses* routines should be used

ROUTINES (descriptions of each *curses* routine are grouped under the appropriate topics):

- 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**

ATTRIBUTES

FUNCTION-KEYS

LINE GRAPHICS

SYNOPSIS

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

#include <curses.h> (automatically includes **<stdio.h>**, **<termio.h>**, and **<unctrl.h>**).

The parameters in the following list are not global variables, but rather this is a summary of the parameters used by the *curses* library routines. All routines return the **int** values **ERR** or **OK** unless otherwise noted. Routines that return pointers always return **NULL** on error. (**ERR**, **OK**, and **NULL** are all defined in **<curses.h>**.) Routines that return integers are not listed in the parameter list below.

bool bf

char **area,*boolnames[ ], *boolcodes[ ], *boolfnames[ ], *bp
char *cap, *capname, codename[2], erasechar, *filename, *fmt
char *keyname, killchar, *label, *longname
char *name, *numnames[ ], *numcodes[ ], *numfnames[ ]
char *slk_label, *str, *strnames[ ], *strcodes[ ], *strfnames[ ]
char *term, *tgetstr, *tigetstr, *tgoto, *tparm, *type

chtype attrs, ch, horch, vertch

FILE *infd, *outfd

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

SCREEN *new, *newterm, *set_term

TERMINAL *cur_term, *nterm, *oterm

va_list varglist

WINDOW *curscr, *dstwin, *initscr, *newpad, *newwin, *orig
WINDOW *pad, *srcwin, *stdscr, *subpad, *subwin, *win

addch(ch)
addstr(str)

attroff(attrs)
attron(attrs)
attrset(attrs)
baudrate( )
beep( )
box(win, vertch, horch)
cbreak( )
clear( )
clearok(win, bf)
clrtobot( )
clrtoeol( )
copywin(srcwin, dstwin, sminrow, smincol, dminrow, dmincol, dmaxrow,
  dmaxcol, overlay)
curs_set(visibility)
def_prog_mode( )
def_shell_mode( )
del_curterm(oterm)
delay_output(ms)
delch( )
deleteln( )
delwin(win)
doupdate( )
draino(ms)
echo( )
echochar(ch)
endwin( )
erase( )
erasechar( )
filter( )
flash( )
flushinp( )
garbagedlines(win, begline, numlines)
getbegyx(win, y, x)
getch( )
getmaxyx(win, y, x)
getstr(str)
getsyx(y, x)
getyx(win, y, x)
halfdelay(tenths)
has_colors( )
has_ic( )

has_il( )
idlok(win, bf)
inch( )
init_color(color, r, g, b)
init_pair(pair, f, b)
initscr( )
insch(ch)
insertln( )
intrflush(win, bf)
isendwin( )
keyname(c)
keypad(win, bf)
killchar( )
leaveok(win, bf)
longname( )
meta(win, bf)
move(y, x)
mvaddch(y, x, ch)
mvaddstr(y, x, str)
mvcur(oldrow, oldcol, newrow, newcol)
mvdelch(y, x)
mvgetch(y, x)
mvgetstr(y, x, str)
mvinch(y, x)
mvinsch(y, x, ch)
mvprintw(y, x, fmt [, arg . . .])
mvscanw(y, x, fmt [, arg . . .])
mvwaddch(win, y, x, ch)
mvwaddstr(win, y, x, str)
mvwdelch(win, y, x)
mvwgetch(win, y, x)
mvwgetstr(win, y, x, str)
mvwin(win, y, x)
mvwinch(win, y, x)
mvwinsch(win, y, x, ch)
mvwprintw(win, y, x, fmt [, arg . . .])
mvwscanw(win, y, x, fmt [, arg . . .])
napms(ms)
newpad(nlines, ncols)
newterm(type, outfd, infd)
newwin(nlines, ncols, begin_y, begin_x)

nl( )
nocbreak( )
nodelay(win, bf)
noecho( )
nonl( )
noraw( )
notimeout(win, bf)
overlay(srcwin, dstwin)
overwrite(srcwin, dstwin)
pair_content(pair, &f, &b)
pechochar(pad, ch)
pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
printw(fmt [, arg . . .])
putp(str)
raw( )
refresh( )
reset_prog_mode( )
reset_shell_mode( )
resetty( )
restartterm(term, fildes, errret)
ripoffline(line, init)
savetty( )
scanw(fmt [, arg . . .])
scr_dump(filename)
scr_init(filename)
scr_restore(filename)
scroll(win)
scrollok(win, bf)
set_curterm(nterm)
set_term(new)
setscrreg(top, bot)
setsyx(y, x)
setupterm(term, fildes, errret)
set_attron(attrs)
set_attrset(attrs)
set_attroff(attrs)
slk_clear( )
slk_init(fmt)
slk_label(labnum)
slk_noutrefresh( )

slk_refresh( )
slk_restore( )
slk_set(labnum, label, fmt)
slk_touch( )
standend( )
standout( )
start_color( )
subpad(orig, nlines, ncols, begin_y, begin_x)
subwin(orig, nlines, ncols, begin_y, begin_x)
tgetent(bp, name)
tgetflag(codename)
tgetnum(codename)
tgetstr(codename, area)
tgoto(cap, col, row)
tigetflag(capname)
tigetnum(capname)
tigetstr(capname)
touchline(win, start, count)
touchwin(win)
tparm(str, p1, p2, ..., p9)
tputs(str, count, putc)
typeahead(fildes)
unctrl(c)
ungetch(c)
vidattr(attrs)
vidputs(attrs, putc)
vwprintw(win, fmt, varglist)
vwscanw(win, fmt, varglist)
waddch(win, ch)
waddstr(win, str)
wattroff(win, attrs)
wattron(win, attrs)
wattrset(win, attrs)
wclear(win)
wclrtobot(win)
wclrtoeol(win)
wdelch(win)
wdeleteln(win)
wechochar(win, ch)
werase(win)
wgetch(win)

wgetstr(win, str)
winch(win)
winsch(win, ch)
winsertln(win)
wmove(win, y, x)
wnoutrefresh(win)
wprintw(win, fmt [, arg...])
wrefresh(win)
wscanw(win, fmt [, arg...])
wsetscrreg(win, top, bot)
wstandend(win)
wstandout(win)

DESCRIPTION

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

In order to initialize the routines, **# include <curses.h>** must be included at the beginning of files 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. This can be done by executing the **tput init** command after the shell environment variable **TERM** has been exported. For further details, see *profile*(4), *tput*(1), and the "Tabs and Initialization" subsection of *terminfo*(4).

The *curses* library contains routines that manipulate data structures called *windows* that can be thought of as two-dimensional arrays of characters representing all or part of a terminal screen. A default window called **stdscr** is supplied, which is the size of the terminal screen. Others can 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 can also be stored with each character.

Special windows called *pads* can 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 can be included which cause the characters to show up in modes such as underlined or in reverse video on terminals that support such display enhancements. Line drawing characters can 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.**

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.

The environment variables **LINES** and **COLUMNS** can be set to override **terminfo**'s idea of how large a screen is. These can be used in an AT&T Teletype 5620 layer, for example, where the size of a screen is changeable.

If the environment variable **TERMINFO** is defined, any program using *curses* checks for a local terminal definition before checking in the standard place. For example, if the environment variable **TERM** is set to **act4**, then the compiled terminal definition is found in */usr/lib/terminfo/a/act4*. (The **a** is copied from the first letter of **act4** to avoid creation of huge directories.) However, if **TERMINFO** is set to *$HOME/myterms*, *curses* first checks *$HOME/myterms/a/act4*, and, if that fails, then checks */usr/lib/terminfo/a/act4*. 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 is 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 SYNOPSIS for a summary of what types all variables are.

All routines return either the integer **ERR** or the integer **OK**, unless otherwise noted. Routines that return pointers always return NULL on error.

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.

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*(3S) call, for example. This routine restores *tty*(7) 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( )

This routine 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 can continue to run in a line-oriented mode if the terminal cannot support a screen-oriented program, must also use this routine. **newterm**( ) should be called once for each terminal. It returns a variable of type **SCREEN*** that should be saved as a reference to that terminal. The arguments are the *type* of the terminal to be used in place of the environment variable **TERM**; *outfd*, a *stdio*(3S) file pointer for output to the terminal; and *infd*, another file pointer for input from the terminal. When it is done running, the program must also call **endwin**( ) for each terminal being used. If **newterm**( ) is called more than once for the same terminal, the first terminal referred to must be the last one for which **endwin**( ) is called.

SCREEN ***set_term**(new)

This routine is used to switch between different terminals. The screen reference *new* becomes the new current terminal. A pointer to the screen of the previous terminal is returned by the routine. This is the only routine which manipulates SCREEN pointers; all other routines affect only the current terminal.

Window and Pad Manipulation

refresh( )

wrefresh (win)

These routines [or **prefresh**( ), **pnoutrefresh**( ), **wnoutrefresh**( ), or **doupdate**( )] must be called to write output to the terminal, as most other routines merely manipulate data structures. **wrefresh**( ) copies the named window to the physical terminal screen, taking into account what is already there in order to minimize the amount of information that's sent to the terminal (called optimization). **refresh**( ) does the same thing, except it uses **stdscr** as a default window. Unless **leaveok**( ) has been enabled, the physical cursor of the terminal is left at the location of the window's cursor. The number of characters output to the terminal is returned.

Note that **refresh**( ) is a macro.

wnoutrefresh(win)

doupdate( )

These two routines allow multiple updates to the physical terminal screen with more efficiency than **wrefresh**( ) alone. How this is accomplished is described in the next paragraph.

curses keeps two data structures representing the terminal screen: a *physical* terminal screen, describing what is actually on the screen, and a *virtual* terminal screen, describing what the programmer wants to have on the screen. **wrefresh**( ) works by first calling **wnoutrefresh**( ), which copys the named window to the virtual screen, and then by calling **doupdate**( ), which compares the virtual screen to the physical screen and does the actual update. If the programmer wishes to output several windows at once, a series of calls to **wrefresh**( ) results 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)

This routine creates and returns 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)

This routine noves the window to position the upper-left corner at (*y*, *x*). If the move would cause any portion of the window to be off the screen, it is an error and the window is not moved.

WINDOW ***subwin**(orig, nlines, ncols, begin_y, begin_x)

This routine creates and returns 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)

This routine deletes the named window, freeing up all memory associated with it. If you try to delete a main window before all of its subwindows are deleted, ERR is returned.

WINDOW ***newpad**(nlines, ncols)

This routine creates and returns 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)

This routine creates and returns 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, smaxcol)
pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)

These routines are analogous to **wrefresh**( ) and **wnoutrefresh**( ) except that pads, instead of windows, are involved. The additional parameters are needed to indicate what part of the pad and screen are involved. *pminrow* and *pmincol* specify the upper-left corner, in the pad, of the rectangle to be displayed. *sminrow*, *smincol*, *smaxrow*, and *smaxcol* specify the edges, on the screen, of the rectangle to be displayed in. The lower-right corner in the pad of the rectangle to be displayed is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of *pminrow*, *pmincol*, *sminrow*, or *smincol* are treated as if they were zero.

Output

These routines are used to 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*(3S)]. 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 Control 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 ORing 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.

Note that **addch( )**, **mvaddch( )**, and **mvwaddch( )**, are macros.

echochar(ch)
wechochar(win, ch)
pechochar(pad, ch)

These routines are functionally equivalent to a call to **addch**(ch) followed by a call to **refresh( )**, a call to **waddch**(win, ch) followed by a call to **wrefresh**(win), or a call to **waddch**(pad, ch) followed by a call to **prefresh**(pad). The knowledge that only a single character is being output is taken into consideration and, for non-control characters, a considerable performance gain can be seen by using these routines instead of their equivalents. In the case of **pechochar( )**, the last location of the pad on the screen is reused for the arguments to **prefresh( )**.

Note that *ch* is actually of type **chtype**, not a character.

Note that **echochar( )** is a macro.

addstr(str)
waddstr(win, str)
mvwaddstr(win, y, x, str)
mvaddstr(y, x, str)

These routines write all the characters of the null-terminated character string *str* on the given window. This is equivalent to calling **waddch( )** once for each character in the string.

Note that **addstr( )**, **mvaddstr( )**, and **mvwaddstr( )** are macros.

attroff(attrs)
wattroff(win, attrs)
attron(attrs)
wattron(win, attrs)
attrset(attrs)
wattrset(win, attrs)
standend( )
wstandend(win)
standout( )
wstandout(win)

These routines manipulate the current attributes of the named window. These attributes can be any combination of **A_STANDOUT**, **A_REVERSE**, **A_BOLD**, **A_DIM**, **A_BLINK**, **A_UNDERLINE**, and **A_ALTCHARSET**, as well as the macro **COLOR_PAIR**( ). These constants are defined in **<curses.h>** and can be combined with the C logical OR (|) operator.

The current attributes of a window are applied to all characters that are written into the window with **waddch**( ). Attributes are a property of the character, and move with the character through any scrolling and insert/delete line/character operations. To the extent possible on the particular terminal, they 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.

clrtobot( )
wclrtobot(win)
All lines below the cursor in this window are erased. Also, the current line to the right of the cursor, inclusive, is erased.

Note that **clrtobot**( ) is a macro.

clrtoeol( )
wclrtoeol(win)
The current line to the right of the cursor, inclusive, is erased.

Note that **clrtoeol**( ) is a macro.

delay_output(ms)
Insert a *ms* millisecond pause in the output. It is not recommended that this routine be used extensively, because padding characters are used rather than a processor pause.

delch( )
wdelch(win)
mvdelch(y, x)
mvwdelch(win, y, x)

The character under the cursor in the window is deleted. All characters to the right on the same line are moved to the left one position and the last character on the line is filled with a blank. The cursor position does not change [after moving to (*y*, *x*), if specified]. (This does not imply use of the hardware ''delete-character'' feature.)

Note that **delch**( ), **mvdelch**( ), and **mvwdelch**( ) are macros.

deleteln( )
wdeleteln(win)

The line under the cursor in the window is deleted. All lines below the current line are moved up one line. The bottom line of the window is cleared. The cursor position does not change. (This does not imply use of the hardware ''delete-line'' feature.)

Note that **deleteln**( ) is a macro.

getyx(win, y, x)

The cursor position of the window is placed in the two integer variables *y* and *x*.

Note that **getyx**( ) is a macro, so no ampersand (&) 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*.

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.

Note that **insch**( ), **mvinsch**( ), and **mvwinsch**( ) are macros.

insertln( )
winsertln(win)

A blank line is inserted above the current line and the bottom line is lost. (This does not imply use of the hardware ''insert-line'' feature.)

Note that **insertln**( ) is a macro.

move(y, x)
wmove(win, y, x)

The cursor associated with the window is moved to line (row) *y*, column *x*. This does not move the physical cursor of the terminal until **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 *srcwin* on top of text from *dstwin* 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 a finer grain of control over the **overlay**( ) and **overwrite**( ) routines. As in the **prefresh**( ) routine, a rectangle is specified in the destination window, (*dminrow*, *dmincol*) and (*dmaxrow*, *dmaxcol*), and the upper-left-corner coordinates of the source window, (*sminrow*, *smincol*). If the argument *overlay* is true, then copying is non-destructive, as in **overlay**( ).

printw(fmt [, arg . . .])
wprintw(win, fmt [, arg . . .])
mvprintw(y, x, fmt [, arg . . .])
mvwprintw(win, y, x, fmt [, arg . . .])

These routines are analogous to **printf**(3). The string which would be output by **printf**(3) is instead output using **waddstr**( ) on the given window.

vwprintw(win, fmt, varglist)

This routine corresponds to *vfprintf*(3S). It performs a **wprintw**( ) using a variable argument list. The third argument is a *va_list*, a pointer to a list of arguments, as defined in **<varargs.h>**. See the *vprintf*(3S) and *varargs*(5) manual pages for a detailed description on how to use variable argument lists.

scroll(win)

The window is scrolled up one line. This involves moving the lines in the window data structure. As an optimization, if the window is **stdscr** and the scrolling region is the entire window, the physical screen is scrolled at the same time.

touchwin(win)
touchline(win, start, count)

Throw away all optimization information about which parts of the window have been touched, by pretending that the entire window has been drawn on. This is sometimes necessary when using overlapping windows, since a change to one window 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*(7) driver when each character is typed, the program may produce undesirable results. If **wgetch**( ) encounters a Control-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 **ControlD**).

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

flushinp( )

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

ungetch(c)

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

inch( )
winch(win)
mvinch(y, x)
mvwinch(win, y, x)

The character, of type **chtype**, at the current position in the named window is returned. If any attributes are set for that position, their values are ORed into the value returned. The predefined constants **A_CHARTEXT** and **A_ATTRIBUTES**, defined in **<curses.h>**, can be used with the C logical AND operator (&) to extract the character or attributes alone.

Note that **inch**( ), **winch**( ), **mvinch**( ), and **mvwinch**( ) are macros.

scanw(fmt [, arg . . .])
wscanw(win, fmt [, arg . . .])
mvscanw(y, x, fmt [, arg . . .])
mvwscanw(win, y, x, fmt [, arg . . .])

These routines correspond to *scanf*(3S), as do their arguments and return values. **wgetstr**( ) is called on the window, and the resulting line is used as input for the scan. 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 than the *stdio* routines (especially **ControlD).**

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*(3S) and *varargs*(5) manual pages for a detailed description on how to use variable argument lists.

Output Options Setting

These routines set options within *curses* that deal with output. All options are initially FALSE, unless otherwise stated. It is not necessary to turn these options off before calling **endwin**( ).

clearok(win, bf)

If enabled (*bf* is **TRUE**), the next call to **wrefresh**( ) with this window 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* uses the hardware ''insert/delete-line'' feature of terminals so equipped; if disabled (*bf* is FALSE), *curses* very seldom uses the hardware ''insert/delete-line'' 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 line 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 used by the output routines.]

Note that **setscrreg**( ) and wsetscrreg( ) are macros.

scrollok(win, bf)

This option controls what happens when the cursor of a window is moved off the edge of the window or scrolling region, either from a newline on the bottom line, or typing the last character of the last line. If disabled (*bf* is **FALSE**), the cursor is left on the bottom line at the location where the offending character was entered. If enabled (*bf* is **TRUE**), **wrefresh**( ) is called on the window, and then the physical terminal and window are scrolled up one line. [Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call **idlok**( ).] Note that **scrollok**( ) always returns **OK**.

Input Options Setting

These routines set options within *curses* that deal with input. The options involve using *ioctl*(2) and therefore interact with *curses* routines. It is not necessary to turn these options off before calling **endwin**( ).

For more information on these options, see *UNIX System V Release 3.2 Programmer's Guide.*

cbreak( )
nocbreak( )

These two routines put the terminal into and out of CBREAK mode, respectively. In CBREAK mode, characters typed by the user are immediately available to the program and erase/kill character processing is not performed. When in NOCBREAK mode, the tty driver buffers characters typed until a newline or carriage return is typed. Interrupt and flow-control characters are unaffected by this mode [see *termio*(7)]. Initially, the terminal may or may not be in CBREAK mode, as it is inherited, therefore, a program should call **cbreak**( ) or **nocbreak**( ) explicitly. Most interactive programs using *curses* sets 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*(7) 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**) enables it.

meta(win, bf)
: Initially, whether the terminal returns seven or eight significant bits on input depends on the control mode of the tty driver [see *termio*(7)]. To force eight bits to be returned, invoke **meta** (*win*, **TRUE**); to force seven bits to be returned, invoke **meta** (*win*, **FALSE**). The window argument, *win*, is always ignored. If the *terminfo*(4) capabilities **smm** (meta_on) and **rmm** (meta_off) are defined for the terminal, **smm** is sent to the terminal when **meta** (*win*, TRUE) is called and **rmm** 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. The differences are that in RAW mode, the interrupt, quit, suspend, and flow control characters are passed through uninterpreted, instead of generating a signal. The behavior of the BREAK key depends on other bits in the tty driver that are not set by *curses* [see *termio*(7)].

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 performed.

Note that *fildes* is a file descriptor, not a **<stdio.h>** FILE pointer.

Environment Queries

baudrate( )

Returns the output speed of the terminal. The number returned is in bits per second, for example, 9600, and is an integer.

char erasechar( )

The user's current erase character is returned.

has_ic( )

True if the terminal has insert- and delete-character capabilities.

has_il( )

True if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This might be used to check to see if it would be appropriate to turn on physical scrolling using **scrollok**( ).

char killchar( )

The user's current line-kill character is returned.

char *longname( )

This routine returns a pointer to a static area containing a verbose description of the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to **initscr**( ) or **newterm**( ). The area is overwritten by each call to **newterm**( ) and is not restored by **set_term**( ), so the value should be saved between calls to **newterm**( ) if **longname**( ) is going to be used with multiple terminals.

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 are 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 eight labels of eight 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 eight 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**( ) is likely to follow soon.

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.

slk_attron(attrs)
slk_attrset(attrs)
slk_attrof(attrs)
These routines correspond to **attron**( ), **attrset**( ), and **attrof**( ). The have effect only if soft labels are simulated at the bottom of the screen.

Low-Level *curses* Access

The following routines give low-level access to various *curses* functionality. These routines typically would be used inside of library routines.

def_prog_mode( )
def_shell_mode( )

Save the current terminal modes as the "program" (in **curses**) or "shell" (not in **curses**) state for use by the **reset_prog_mode( )** and **reset_shell_mode( )** routines. This is done automatically by **initscr( )**.

reset_prog_mode( )
reset_shell_mode( )

Restore the terminal to "program" (in **curses**) or "shell" (out of *curses*) state. These are done automatically by **endwin( )** and **doupdate( )** after an **endwin( )**, so they normally would not be called.

resetty( )
savetty( )

These routines save and restore the state of the terminal modes. **savetty( )** saves the current state of the terminal in a buffer and **resetty( )** restores the state to what it was at the last call to **savetty( )**.

getsyx(y, x)

The current coordinates of the virtual screen cursor are returned in *y* and *x*. If **leaveok( )** is currently **TRUE**, then **-1,-1** is returned. If lines have been removed from the top of the screen using **ripoffline( )**, *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( )**.

ripoffline(line, init)

This routine provides access to the same facility that **slk_init( )** uses to reduce the size of the screen. **ripoffline( )** must be called before **initscr( )** or **newterm( )** is called. If *line* is positive, a line 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 one-line window that has been allocated and an integer with the number of columns in the window. Inside this initialization routine, the integer variables **LINES** and **COLS** (defined in **<curses.h>**) are not guaranteed to be accurate and **wrefresh**( ) or **doupdate**( ) must not be called. It is allowable to call **wnoutrefresh**( ) during the initialization routine. **ripoffline**( ) can be called up to five times before calling **initscr**( ) or **newterm**( ).

scr_dump(filename)
The current contents of the virtual screen are written to the file *filename*.

scr_restore(filename)
The virtual screen is set to the contents of *filename*, which must have been written using **scr_dump**( ). **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*(3S) 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 time-stamp of the tty is old or the *terminfo*(4) capability **nrrmc** is true. 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 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.

draino(ms)
Wait until the output has drained enough that it takes only *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*(4) database to handle certain terminal capabilities, such as programming function keys. For all other functionality, *curses* routines are more suitable and their use is recommended.

Initially, **setupterm**( ) should be called. [Note that **setupterm**( ) is automatically called by **initscr**( ) and **newterm**( ).] This defines the set of terminal-dependent variables defined in the *terminfo*(4) database. The *terminfo*(4) variables **lines** and **columns** [see *terminfo*(4)] are initialized by **setupterm**( ) as follows: if the environment variables **LINES** and **COLUMNS** exist, their values are used. If the above environment variables do not exist, the values for **lines** and **columns** specified in the *terminfo*(4) database are used.

The header files **<curses.h>** and **<term.h>** should be included, in this order, to get the definitions for these strings, numbers, and flags. Parameterized strings should be passed through **tparm**( ) to instantiate them. All *terminfo*(4) strings [including the output of **tparm**( )] should be printed with **tputs**( ) or **putp**( ). Before exiting, **reset_shell_mode**( ) should be called to restore the tty modes. Programs that use cursor addressing should output **enter_ca_mode** upon startup and should output **exit_ca_mode** before exiting [see *terminfo*(4)]. Programs that use shell escapes should call **reset_shell_mode**( ) and output **exit_ca_mode** before the shell is called and should output **enter_ca_mode** and call **reset_prog_mode**( ) after returning from the shell. Note that this is different from the *curses* routines [see **endwin**( )].

setupterm(term, fildes, errret)

Reads in the *terminfo*(4) database, initializing the *terminfo*(4) structures, but does not set up the output virtualization structures used by *curses*. The terminal type is in the character string *term*; if *term* is NULL, the environment variable **TERM** is used. All output is to the file descriptor *fildes*. If *errret* is not NULL, then **setupterm**( ) returns

OK or **ERR** and store a status value in the integer pointed to by *errret*. A status of **1** in *errret* is normal, **0** means that the terminal could not be found, and **-1** means that the *terminfo*(4) database could not be found. If *errret* is **NULL, setupterm**( ) prints an error message upon finding an error and exit. Thus, the simplest call is **setupterm** [(**char** *)**0, 1,** (**int** *)**0**], which uses all the defaults.

The *terminfo*(4) boolean, numeric and string variables are stored in a structure of type **TERMINAL**. After **setupterm**( ) returns successfully, the variable **cur_term** (of type **TERMINAL** *) is initialized with all of the information that the *terminfo*(4) boolean, numeric and string variables refer to. The pointer can be saved before calling **setupterm**( ) again. Further calls to **setupterm**( ) allocates new space rather than reuse the space pointed to by **cur_term**.

set_curterm(nterm)
nterm is of type TERMINAL *. **set_curterm**( ) sets the variable **cur_term** to *nterm*, and makes all of the *terminfo*(4) boolean, numeric and string variables use the values from *nterm*.

del_curterm(oterm)
oterm is of type **TERMINAL** *. **del_curterm**( ) frees the space pointed to by *oterm* and makes it available for further use. If *oterm* is the same as **cur_term**, then references to any of the *terminfo*(4) boolean, numeric and string variables thereafter may refer to invalid memory locations until another **setupterm**( ) has been called.

restartterm(term, fildes, errret)
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, $p_1, p_2, \ldots, p_9$)
Instantiate the string *str* with parms p_i. A pointer is returned to the result of *str* with the parameters applied.

tputs(str, count, putc)
Apply padding to the string *str* and output it. *str* must be a *terminfo*(4) string variable or the return value from **tparm**( ), **tgetstr**( ), **tigetstr**( ) or **tgoto**( ). *count* is the number of lines affected, or **1** if not applicable. *putc*( ) is a *putchar*(3S)-like routine to which the characters are passed, one at a time.

putp(str)
A routine that calls **tputs** [*str*, **1**, **putchar**( )].

vidputs(attrs, putc)
Output a string that puts the terminal in the video attribute mode *attrs*, which is any combination of the attributes listed below. The characters are passed to the *putchar*(3S)-like routine *putc*( ).

vidattr(attrs)
Like **vidputs**( ), except that it outputs through *putchar*(3S).

The following routines return the value of the capability corresponding to the character string containing the *terminfo*(4) *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*(4) variables.

Termcap Emulation

These routines are included as a conversion aid for programs that use the *termcap* library. Their parameters are the same and the routines are emulated using the *terminfo*(4) database.

tgetent(bp, name)
Look up *termcap* entry for *name*. The emulation ignores the buffer pointer *bp*.

tgetflag(codename)
Get the boolean entry for *codename*.

tgetnum(codes)
Get numeric entry for *codename*.

char *tgetstr(codename, area)
Return the string entry for *codename*. If *area* is not **NULL**, then also store it in the buffer pointed to by *area* and advance *area*. **tputs**( ) should be used to output the returned string.

char *tgoto(cap, col, row)
Instantiate the parameters into the given capability. The output from this routine is to be passed to **tputs**( ).

tputs(str, affcnt, putc)
See **tputs**( ) above, in this section.

Miscellaneous

unctrl(c)
This macro expands to a character string which is a printable representation of the character *c*. Control characters are displayed in the Control X notation. Printing characters are displayed as is.

unctrl( ) is a macro, defined in **<unctrl.h>**, which is automatically included by **<curses.h>**.

char *keyname(c)
A character string corresponding to the key *c* is returned.

filter( ) This routine is one of the few that is to be called before **initscr**( ) or **newterm**( ) is called. It arranges things so that *curses* thinks that there is a one-line screen. *curses* 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 **cbreak**( ).
fixterm( )	Replaced by **reset_prog_mode**( ).
gettmode( )	A no-op.
nocrmode( )	Replaced by **nocbreak**( ).
resetterm( )	Replaced by **reset_shell_mode**( ).
saveterm( )	Replaced by **def_prog_mode**( ).
setterm( )	Replaced by **setupterm**( ).

ATTRIBUTES

The following video attributes, defined in **<curses.h>**, can be passed to the routines **wattron**( ), **wattroff**( ), and **wattrset**( ), or ORed 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	Color-pair defined in n (note that this is a macro)

A_CHARTEXT	Bit-mask to extract character [described under **winch**( )]
A_ATTRIBUTES	Bit-mask to extract attributes [described under **winch**( )]
A_NORMAL	Bit mask to reset all attributes off [for example: **attrset (A_NORMAL)**]
A_COLOR	Extract color-pair field information

FUNCTION-KEYS

The following function keys, defined in **<curses.h>**, might be returned by **getch**( ) if **keypad**( ) has been enabled. Note that not all of these can be supported on a particular terminal if the terminal does not transmit a unique code when the key is pressed or the definition for the key is not present in the *terminfo* (4) database.

Name	*Value*	*Key name*
KEY_BREAK	0401	Break key (unreliable)
KEY_DOWN	0402	The four arrow keys ...
KEY_UP	0403	
KEY_LEFT	0404	
KEY_RIGHT	0405	...
KEY_HOME	0406	Home key (upward+left arrow)
KEY_BACKSPACE	0407	Backspace (unreliable)
KEY_F0	0410	Function keys. Space for 64 keys is reserved.
KEY_F(n)	[KEY_F0 +(n)]	Formula for f_n.
KEY_DL	0510	Delete line
KEY_IL	0511	Insert line
KEY_DC	0512	Delete character
KEY_IC	0513	Insert char or enter insert mode
KEY_EIC	0514	Exit insert char mode
KEY_CLEAR	0515	Clear screen
KEY_EOS	0516	Clear to end of screen
KEY_EOL	0517	Clear to end of line
KEY_SF	0520	Scroll one line forward
KEY_SR	0521	Scroll one line backwards (reverse)
KEY_NPAGE	0522	Next page
KEY_PPAGE	0523	Previous page
KEY_STAB	0524	Set tab
KEY_CTAB	0525	Clear tab
KEY_CATAB	0526	Clear all tabs
KEY_ENTER	0527	Enter or send
KEY_SRESET	0530	Soft (partial) reset
KEY_RESET	0531	Reset or hard reset

KEY_PRINT	0532	Print or copy
KEY_LL	0533	Home down or bottom (lower-left). Keypad is arranged like this: A1 up A3 / left B2 right / C1 down C3
KEY_A1	0534	Upper left of keypad
KEY_A3	0535	Upper right of keypad
KEY_B2	0536	Center of keypad
KEY_C1	0537	Lower left of keypad
KEY_C3	0540	Lower right of keypad
KEY_BTAB	0541	Back tab key
KEY_BEG	0542	Beg(inning) key
KEY_CANCEL	0543	Cancel key
KEY_CLOSE	0544	Close key
KEY_COMMAND	0545	Cmd (command) key
KEY_COPY	0546	Copy key
KEY_CREATE	0547	Create key
KEY_END	0550	End key
KEY_EXIT	0551	Exit key
KEY_FIND	0552	Find key
KEY_HELP	0553	Help key
KEY_MARK	0554	Mark key
KEY_MESSAGE	0555	Message key
KEY_MOVE	0556	Move key
KEY_NEXT	0557	Next object key
KEY_OPEN	0560	Open key
KEY_OPTIONS	0561	Options key
KEY_PREVIOUS	0562	Previous object key
KEY_REDO	0563	Redo key
KEY_REFERENCE	0564	Ref(erence) key
KEY_REFRESH	0565	Refresh key
KEY_REPLACE	0566	Replace key
KEY_RESTART	0567	Restart key
KEY_RESUME	0570	Resume key

KEY_SAVE	0571	Save key
KEY_SBEG	0572	Shifted beginning key
KEY_SCANCEL	0573	Shifted cancel key
KEY_SCOMMAND	0574	Shifted command key
KEY_SCOPY	0575	Shifted copy key
KEY_SCREATE	0576	Shifted create key
KEY_SDC	0577	Shifted delete char key
KEY_SDL	0600	Shifted delete line key
KEY_SELECT	0601	Select key
KEY_SEND	0602	Shifted end key
KEY_SEOL	0603	Shifted clear line key
KEY_SEXIT	0604	Shifted exit key
KEY_SFIND	0605	Shifted find key
KEY_SHELP	0606	Shifted help key
KEY_SHOME	0607	Shifted home key
KEY_SIC	0610	Shifted input key
KEY_SLEFT	0611	Shifted left arrow key
KEY_SMESSAGE	0612	Shifted message key
KEY_SMOVE	0613	Shifted move key
KEY_SNEXT	0614	Shifted next key
KEY_SOPTIONS	0615	Shifted options key
KEY_SPREVIOUS	0616	Shifted prev key
KEY_SPRINT	0617	Shifted print key
KEY_SREDO	0620	Shifted redo key
KEY_SREPLACE	0621	Shifted replace key
KEY_SRIGHT	0622	Shifted right arrow
KEY_SRSUME	0623	Shifted resume key
KEY_SSAVE	0624	Shifted save key
KEY_SSUSPEND	0625	Shifted suspend key
KEY_SUNDO	0626	Shifted undo key
KEY_SUSPEND	0627	Suspend key
KEY_UNDO	0630	Undo key

LINE GRAPHICS

The following variables can be used to add line-drawing characters to the screen with **waddch**( ). When defined for the terminal, the variable has the **A_ALTCHARSET** bit enabled; otherwise, the default charcter 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

RETURN VALUES

All routines return the integer **OK** upon successful completion and the integer **ERR** upon failure, unless otherwise noted in the preceding routine descriptions.

All macros return the value of their **w** version, except **getsyx**( ), **getyx**( ), **getbegyx**( ), **getmaxyx**( ). For these macros, no useful value is returned.

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

SEE ALSO

cc(1), ld(1), ioctl(2), plot(3X), putc(3S), scanf(3S), stdio(3S), system(3S), vprintf(3S), profile(4), term(4), terminfo(4), varargs(5), termio(7).
UNIX System V Release 3.2 Programmer's Guide.

WARNINGS

To use the new *curses* features, use the version of *curses* on CTIX Releases 6.1 and higher. All programs that ran *curses* under CTIX releases prior to 6.1 will run with CTIX Release 6.1. You can link applications with object files based on the previous *curses/terminfo* with the CTIX 6.1 *libcurses.a* library. You can link applications with object files based on the CTIX 6.1 *curses/terminfo* with previous CTIX releases' *libcurses.a* libraries, so long as the application does not use the new features in the CTIX 6.1 *curses/terminfo*.

The plotting library *plot*(3X) and the curses library *curses*(3X) both use the names **erase**( ) and **move**( ). The *curses* versions are macros. If you need both libraries, put the *plot*(3X) code in a different source file than the *curses*(3X) code, and/or **#undef move**( ) and **erase**( ) in the *plot*(3X) code.

Between the time a call to **initscr**( ) and **endwin**( ) has been issued, use only the routines in the *curses* library to generate output. Using system calls or the "standard I/O package" [see *stdio*(3S)] for output during that time can cause unpredictable results.

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).

BUGS

Currently typeahead checking is done using a nodelay read followed by an **ungetch**( ) of any character that may have been read. Typeahead checking is done only if **wgetch**( ) has been called at least once. This will be changed when proper kernel support is available. Programs which use a mixture of their own input routines with *curses* input routines may wish to call **typeahead(-1)** to turn off typeahead checking. The argument to **napms**( ) is currently rounded up to the nearest second.

draino (ms) only works for *ms* equal to **0**.

NAME

cuserid - get character login name of the user

SYNOPSIS

#include <stdio.h>

char *cuserid (s)
char *s;

DESCRIPTION

The *cuserid* routine gets the user's login name as found in **/etc/utmp**. If the login name cannot be found, *cuserid* gets the login name corresponding to the user ID of the current process. If *s* is a NULL pointer, this representation is generated in an internal static area, the address of which is returned. Otherwise, *s* is assumed to point to an array of at least **L_cuserid** characters; the representation is left in this array. The constant **L_cuserid** is defined in the **<stdio.h>** header file.

SEE ALSO

getlogin(3C), getpwent(3C).

DIAGNOSTICS

If the login name cannot be found, *cuserid* returns a NULL pointer; if *s* is not a NULL pointer, a null character (\0) will be placed at *s[0]*.

NAME

dbminit, fetch, store, delete, firstkey, nextkey - database subroutines

SYNOPSIS

```
#include <dbm.h>

typedef struct {
        char *dptr;
        int dsize;
} datum;

dbminit(file)
char *file;

datum fetch(key)
datum key;

store(key, content)
datum key, content;

delete(key)
datum key;

datum firstkey()

datum nextkey(key)
datum key;
```

DESCRIPTION

The *dbm* 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**.

*key*s and *content*s 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 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 can 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))
```

SEE ALSO

ndbm(3X).

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*.

NOTE

The *dbm* library has been superseded by *ndbm*(3), and is now implemented using *ndbm*. *dbm*(3X) is included for compatibility with existing programs that invoke *dbm*(3X). When writing new programs, use *ndbm*(3X) instead.

WARNINGS

The **.pag** file contains holes so that its apparent size is about four times its actual content. Older UNIX systems can 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, not on anything interesting.

NAME

dial - establish an out-going terminal line connection

SYNOPSIS

```
#include <dial.h>

int dial (call)
CALL call;

void undial (fd)
int fd;
```

DESCRIPTION

The *dial* routine 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 as follows:

```
typedef struct {
    struct termio  *attr;       /* pointer to termio attribute
                                   struct */
    int            baud;        /* transmission data rate */
    int            speed;       /* 212A modem: low=300,
                                   high=1200 (unused) */
    char           *line;       /* device name for out-going line */
    char           *telno;      /* pointer to tel-no digits string */
    int            modem;       /* specify modem control for direct
                                   lines */
    char           *device;     /* Will hold the name of the device
                                   used to make a connection
                                   (unused) */
    int            dev_len;     /* The length of the device used to
                                   make connection (unused) */
} CALL;
```

The CALL element *baud* is for the desired transmission baud rate. The rate must be one of those supported by the operating system (134.5 is rounded to 134). If the *baud* is less than 300, the line will be dialed at 300 baud then switched to the desired rate (unless *attr* is non-null; see below).

If a particular terminal line is desired, 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 **/usr/lib/uucp/Devices**. In this case, if *baud* is 0, the speed used will be determined by the line in the **Devices** file for the terminal device.

The *telno* element is for a pointer to a character string representing the telephone number to be dialed. Numbers consist of the following symbols:

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

On a smart modem, these symbols are translated to modem commands using the modem description in **/usr/lib/uucp/Dialers**.

If *telno* is specified, an ACU entry in the **Devices** file will be used. If it is NULL, a Direct entry will be used.

The CALL element *modem* is used to specify modem control for direct lines. This element should be non-zero if modem control is required.

The CALL element *attr* is a pointer to a *termio* structure, as defined in the *termio.h* header file. A NULL value for this pointer element may be passed to the *dial* function, but if such a structure is included, the elements specified in it will be set for the outgoing terminal line before the connection is established. This is often important for certain attributes such as parity and baud-rate. Values in this structure override the *baud* and *modem* entries.

Information on 801 type dialing units is obtained from the **Devices** file; thus the *speed*, *device* and *dev_len* elements are no longer used.

FILES

/usr/lib/uucp/Devices
/usr/lib/uucp/Dialers
/usr/spool/locks/LCK..*tty-device*

SEE ALSO

uucp(1C), alarm(2), read(2), write(2), Devices(5), Dialers(5), termio(7).

DIAGNOSTICS

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

```
INTRPT   -1   /* interrupt occurred */
D_HUNG   -2   /* dialer hung (no return from write) */
NO_ANS   -3   /* no answer within 10 seconds */
```

```
ILL_BD    -4     /* illegal baud-rate */
A_PROB    -5     /* acu problem (open() failure) */
L_PROB    -6     /* line problem (open() failure) */
NO_Ldv    -7     /* can't open LDEVS file */
DV_NT_A   -8     /* requested device not available */
DV_NT_K   -9     /* requested device not known */
NO_BD_A   -10    /* no device available at requested baud */
NO_BD_K   -11    /* no device known at requested baud */
```

WARNINGS

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

The above routine uses **<stdio.h>**, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

BUGS

An *alarm*(2) system call for 3600 seconds is made (and caught) within the *dial* module for the purpose of "touching" the *LCK..* file and constitutes the device allocation semaphore for the terminal device. Otherwise, *uucp*(1C) may simply delete the *LCK..* entry on its 90-minute clean-up rounds. The alarm may go off while the user program is in a *read*(2) or *write*(2) system call, causing an apparent error return. If the user program expects to be around for an hour or more, error returns from *read*s should be checked for **(errno==EINTR)**, and the *read* possibly reissued.

NAME

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

SYNOPSIS

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

DIR *opendir (filename)
char *filename;

struct dirent *readdir (dirp)
DIR *dirp;

long telldir (dirp)
DIR *dirp;

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

void rewinddir (dirp)
DIR *dirp;

void closedir(dirp)
DIR *dirp;
```

DESCRIPTION

The *opendir* routine opens the directory named by *filename* and associates a *directory stream* with it. The *opendir* routine 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*(3X) enough memory to hold a DIR structure or a buffer for the directory entries.

The *readdir* routine 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.

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

The *seekdir* routine sets the position of the next *readdir* operation on the *directory stream.* The new position reverts to the one associated with the *directory stream* when the *telldir* operation from which *loc* was obtained 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.

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

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

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

opendir:

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

readdir:

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

telldir, *seekdir*, and *closedir*:

[EBADF]	The file descriptor determined by the DIR stream is no longer valid. This results if the DIR stream has been closed.

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;
```

SEE ALSO

getdents(2), dirent(4).

WARNINGS

The *rewinddir* routine is implemented as a macro, so its function address cannot be taken.

NAME

drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48 - generate uniformly distributed pseudo-random numbers

SYNOPSIS

double drand48 ( )

double erand48 (xsubi)
unsigned short xsubi[3];

long lrand48 ( )

long nrand48 (xsubi)
unsigned short xsubi[3];

long mrand48 ( )

long jrand48 (xsubi)
unsigned short xsubi[3];

void srand48 (seedval)
long seedval;

unsigned short *seed48 (seed16v)
unsigned short seed16v[3];

void lcong48 (param)
unsigned short param[7];

DESCRIPTION

This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

Functions *drand48* and *erand48* return non-negative double-precision floating-point values uniformly distributed over the interval [0.0, 1.0).

Functions *lrand48* and *nrand48* return non-negative long integers uniformly distributed over the interval $[0, 2^{31})$.

Functions *mrand48* and *jrand48* return signed long integers uniformly distributed over the interval $[-2^{31}, 2^{31})$.

Functions *srand48, seed48* and *lcong48* are initialization entry points, one of which should be invoked before either *drand48, lrand48* or *mrand48* is called. (Although it is not recommended practice, constant default initializer values will be supplied automatically if *drand48, lrand48* or *mrand48* is called without a prior call to an initialization entry point.) Functions *erand48, nrand48* and *jrand48* do not require an initialization entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values, X_i, according to the linear congruential formula:

$$X_{n+1} = (aX_n + c)_{\text{mod } m} \qquad n \geq 0.$$

The parameter $m = 2^{48}$; hence 48-bit integer arithmetic is performed. Unless *lcong48* has been invoked, the multiplier value a and the addend value c are given by:

$$a = \text{5DEECE66D}_{16} = 273673163155_8$$
$$c = \text{B}_{16} = 13_8.$$

The value returned by any of the functions *drand48, erand48, lrand48, nrand48, mrand48* or *jrand48* is computed by first generating the next 48-bit X_i in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (leftmost) bits of X_i and transformed into the returned value.

The functions *drand48, lrand48* and *mrand48* store the last 48-bit X_i generated in an internal buffer, 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 will *not* depend upon how many times the routines have been called to generate numbers for the other streams.

The initializer function *srand48* sets the high-order 32 bits of X_i to the 32 bits contained in its argument. The low-order 16 bits of X_i are set to the arbitrary value 330E_{16}.

The initializer function *seed48* sets the value of X_i to the 48-bit value specified in the argument array. In addition, the previous value of X_i is copied into a 48-bit internal buffer, used only by *seed48*, and a pointer to this buffer is the value returned by *seed48*. This returned pointer, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time — use the pointer to get at and store the last X_i value, and then use this value to reinitialize via *seed48* when the program is restarted.

The initialization function *lcong48* allows the user to specify the initial X_i, the multiplier value a, and the addend value c. Argument array elements *param[0-2]* specify X_i, *param[3-5]* specify the multiplier a, and *param[6]* specifies the 16-bit addend c. After *lcong48* has been called, a subsequent call to either *srand48* or *seed48* will restore the ''standard'' multiplier and addend values, *a* and *c*, specified on the previous page.

SEE ALSO

rand(3C).

NAME

dup2 - duplicate an open file descriptor

SYNOPSIS

int dup2 (fildes, fildes2)
int fildes, fildes2;

DESCRIPTION

The *fildes* argument is a file descriptor referring to an open file; *fildes2* is a non-negative integer less than NOFILES. (NOFILES is a system-imposed maximum per process [see *creat*(2)].) The *dup2* routine causes *fildes2* to refer to the same file as *fildes*. If *fildes2* already referred to an open file, it is closed first.

Note that *dup2* fails if one or more of the following are true:

[EBADF]	*fildes* is not a valid open file descriptor.
[EMFILE]	NOFILES file descriptors are currently open.

SEE ALSO

creat(2), close(2), exec(2), fcntl(2), open(2), pipe(2), lockf(3C).

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.

NAME

ecvt, fcvt, gcvt - convert floating-point number to string

SYNOPSIS

char *ecvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt (value, ndigit, buf)
double value;
int ndigit;
char *buf;

DESCRIPTION

The *ecvt* routine converts *value* to a null-terminated string of *ndigit* digits and returns a pointer to that string. The high-order digit is non-zero, unless the value is zero. The low-order digit is rounded. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). The decimal point is not included in the returned string. If the sign of the result is negative, the word pointed to by *sign* is non-zero; otherwise, it is zero.

The *fcvt* routine 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*.

The *gcvt* routine 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 in E-format, ready for printing. A minus sign if any, or a decimal point, will be included as part of the returned string. Trailing zeros are suppressed.

SEE ALSO

printf(3S).

BUGS

The values returned by *ecvt* and *fcvt* point to a single static data array whose content is overwritten by each call.

NAME

end, etext, edata - last locations in program

SYNOPSIS

extern end;
extern etext;
extern edata;

DESCRIPTION

These names refer neither to routines nor to locations with interesting contents. The address of *etext* is the first address above the program text, *edata* above the initialized data region, and *end* above the uninitialized data region.

When execution begins, the program break (the first location beyond the data) coincides with *end*, but the program break may be reset by the routines of *brk*(2), *malloc*(3C), standard input/output [*stdio*(3S)], the profile (**-p**) option of *cc*(1), and so on. Thus, the current value of the program break should be determined by **sbrk [(char *)(0)]** [see *brk*(2)].

SEE ALSO

cc(1), brk(2), malloc(3C), stdio(3S).

NAME

erf, erfc - error function and complementary error function

SYNOPSIS

#include <math.h>

double erf (x)
double x;

double erfc (x)
double x;

DESCRIPTION

The *erf* routine returns the error function of x, defined as follows:

$$\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

The *erfc* routine, which returns 1.0 - *erf(x)*, is provided because of the extreme loss of relative accuracy if *erf(x)* is called for large x and the result subtracted from 1.0 (e.g., for $x = 5$, 12 places are lost).

SEE ALSO

exp(3M).

NAME

exp, log, log10, pow, sqrt - exponential, logarithm, power, square root functions

SYNOPSIS

#include <math.h>

double exp (x)
double x;

double log (x)
double x;

double log10 (x)
double x;

double pow (x, y)
double x, y;

double sqrt (x)
double x;

DESCRIPTION

The *exp* routine returns e^x.

The *log* routine returns the natural logarithm of x. The value of x must be positive.

The *log10* routine returns the logarithm base ten of x. The value of x must be positive.

The *pow* routine returns x^y. If x is zero, y must be positive. If x is negative, y must be an integer.

The *sqrt* routine returns the non-negative square root of x. The value of x may not be negative.

SEE ALSO

hypot(3M), matherr(3M), sinh(3M).

DIAGNOSTICS

The *exp* routine 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* routines 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* routine 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 ±**HUGE** or 0, respectively, and sets *errno* to **ERANGE.**

The *sqrt* routine returns 0 and sets *errno* to **EDOM** when x is negative. A message indicating DOMAIN error is printed on the standard error output.

These error-handling procedures may be changed with the function *matherr*(3M).

NAME

fclose, fflush - close or flush a stream

SYNOPSIS

```
#include <stdio.h>

int fclose (stream)
FILE *stream;

int fflush (stream)
FILE *stream;
```

DESCRIPTION

The *fclose* routine causes any buffered data for the named *stream* to be written out, and the *stream* to be closed.

The *fclose* routine is performed automatically for all open files upon calling *exit*(2).

The *fflush* routine causes any buffered data for the named *stream* to be written to that file. The *stream* remains open.

SEE ALSO

close(2), exit(2), fopen(3S), setbuf(3S), stdio(3S).

DIAGNOSTICS

These functions return 0 for success and **EOF** if any error was detected (such as trying to write to a file that has not been opened for writing).

NAME

ferror, feof, clearerr, fileno - stream status inquiries

SYNOPSIS

```
#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* routine returns non-zero when an I/O error has previously occurred reading from or writing to the named *stream*, otherwise zero.

The *feof* routine returns non-zero when **EOF** has previously been detected reading the named input *stream*, otherwise zero.

The *clearerr* routine resets the error indicator and **EOF** indicator to zero on the named *stream*.

The *fileno* routine returns the integer file descriptor associated with the named *stream*; see *open*(2).

SEE ALSO

open(2), fopen(3S), stdio(3S).

NOTES

All the functions are implemented as macros; they cannot be declared or redeclared.

NAME

floor, ceil, fmod, fabs - floor, ceiling, remainder, absolute value functions

SYNOPSIS

```
#include <math.h>

double floor (x)
double x;

double ceil (x)
double x;

double fmod (x, y)
double x, y;

double fabs (x)
double x;
```

DESCRIPTION

The *floor* routine returns the largest integer (as a double-precision number) not greater than x.

The *ceil* routine returns the smallest integer not less than x.

The *fmod* routine 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 some integer i, and $|f| < |y|$.

The *fabs* routine returns the absolute value of x, $|x|$.

SEE ALSO

abs(3C).

NAME

fopen, freopen, fdopen - open a stream

SYNOPSIS

#include <stdio.h>

FILE *fopen (filename, type)
char *filename, *type;

FILE *freopen (filename, type, stream)
char *filename, *type;
FILE *stream;

FILE *fdopen (fildes, type)
int fildes;
char *type;

DESCRIPTION

The *fopen* routine opens the file named by *filename* and associates a *stream* with it. The routine returns a pointer to the FILE structure associated with the *stream*.

The *filename* argument points to a character string that contains the name of the file to be opened.

The *type* argument is a character string having one of the following values:

r	open for reading
w	truncate or create for writing
a	append; open for writing at end of file, or create for writing
r+	open for update (reading and writing)
w+	truncate or create for update
a+	append; open or create for update at end-of-file

The *freopen* routine substitutes the named file in place of the open *stream*. The original *stream* is closed, regardless of whether the open ultimately succeeds. The routine returns a pointer to the FILE structure associated with *stream*.

The *freopen* routine is typically used to attach the preopened *streams* associated with **stdin, stdout** and **stderr** to other files.

The *fdopen* routine associates a *stream* with a file descriptor. File descriptors are obtained from *open*, *dup*, *creat*, or *pipe*(2), which open files but do not return pointers to a FILE structure *stream*. Streams are necessary input for many of the Section 3S library routines. The *type* of *stream* must agree with the mode of the open file.

When a file is opened for update, both input and output can be done on the resulting *stream*. However, output cannot be followed directly by input without an intervening *fseek* or *rewind*, and input cannot be followed directly by output without an intervening *fseek*, *rewind*, or an input operation which 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. *fseek* can be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output. If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written by the other. The output from the two processes will be intermixed in the file in the order in which it is written.

SEE ALSO

creat(2), dup(2), open(2), pipe(2), fclose(3S), fseek(3S), stdio(3S).

DIAGNOSTICS

The *fopen*, *fdopen*, and *freopen* routines return a NULL pointer on failure.

NAME

fpgetround, fpsetround, fpgetmask, fpsetmask, fpgetsticky, fpsetsticky - IEEE floating point environment control

SYNOPSIS

```
#include <ieeefp.h>

typedef enum {
        FP_RN=0,        /* round to nearest */
        FP_RZ=0x10,     /* round to zero (truncate) */
        FP_RM=0x20,     /* round to minus */
        FP_RP=0x30,     /* round to plus */
        } fp_rnd;

fp_rnd fpgetround( );

fp_rnd fpsetround(rnd_dir)
fp_rnd rnd_dir;

#define    fp_except    int
#define    FP_X_INV     0x80   /* invalid operation */
                               /* exception */
#define    FP_X_OFL     0x40   /* overflow */
                               /* exception */
#define    FP_X_UFL     0x20   /* underflow */
                               /* exception */
#define    FP_X_DZ      0x10   /* divide-by-zero */
                               /* exception */
#define    FP_X_IMP     0x08   /* imprecise (loss */
                               /* of precision) */

fp_except fpgetmask( );

fp_except fpsetmask(mask);
fp_except mask;

fp_except fpgetsticky( );

fp_except fpsetsticky(sticky);
fp_except sticky;
```

DESCRIPTION

These routines let the user change the behavior on occurrence of any of five floating point exceptions: divide-by-zero, overflow, underflow, imprecise (inexact) result, and invalid operation. The routines also change the rounding mode for floating point operations. When a floating point exception occurs, the corresponding sticky bit is set (1), and if the mask bit is enabled (1), the trap takes place. The routines are valid only on systems that are equipped with floating point accelerator hardware; otherwise, floating point operations are compiled differently and handled in software.

The *fpgetround*( ) routine returns the current rounding mode.

The *fpsetround*( ) routine sets the rounding mode and returns the previous rounding mode.

The *fpgetmask*( ) routine returns the current exception masks.

The *fpsetmask*( ) routine sets the exception masks and returns the previous setting.

The *fpgetsticky*( ) routine returns the current exception sticky flags.

The *fpsetsticky*( ) routine sets (clears) the exception sticky flags and returns the previous setting.

The environment for Convergent computers that combine the MC68020 CPU with the MC68881 or MC68882 floating point processor follows:

- Rounding mode set to nearest(FP_RN),
- Divide-by-zero,
- Floating point overflow, and
- Invalid operation traps enabled.

SEE ALSO

isnan(3C).

CAVEATS

The utilities described in this manual page are applicable only for computers that are equipped with both the MC68020 microprocessor for the CPU and the MC68881 or MC68882 microprocessor for a hardware floating point accelerator. Programs that invoke these utilities that are run on computers without the floating point hardware result in no operation and no returned error message for the particular function.

One must clear the sticky bit to recover from the trap and to proceed. If the sticky bit is not cleared before the next trap occurs, a wrong exception type may be signaled.

For the same reason, when calling *fpsetmask*( ) the user should make sure that the sticky bit corresponding to the exception being enabled is cleared.

WARNINGS

The *fpsetsticky*() routine 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.

NAME

fread, fwrite - binary input/output

SYNOPSIS

```
#include <stdio.h>
#include <sys/types.h>

int fread (ptr, size, nitems, stream)
char *ptr;
int nitems;
size_t size;
FILE *stream;

int fwrite (ptr, size, nitems, stream)
char *ptr;
int nitems;
size_t size;
FILE *stream;
```

DESCRIPTION

The *fread* routine copies, into an array pointed to by *ptr*, *nitems* items of data from the named input *stream*, where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. The routine stops appending bytes if an end-of-file or error condition is encountered while reading *stream*, or if *nitems* items have been read. The *fread* routine leaves the file pointer in *stream*, if defined, pointing to the byte following the last byte read if there is one. It does not change the contents of *stream*.

The *fwrite* routine appends at most *nitems* items of data from the array pointed to by *ptr* to the named output *stream*; it stops appending when it has appended *nitems* items of data or if an error condition is encountered on *stream*. The *fwrite* routine 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*.

SEE ALSO

read(2), write(2), fopen(3S), getc(3S), gets(3S), printf(3S), putc(3S), puts(3S), scanf(3S), stdio(3S).

DIAGNOSTICS

The *fread* and *fwrite* routines return the number of items read or written. If *nitems* is non-positive, no characters are read or written and 0 is returned by both *fread* and fwrite.

NAME

frexp, ldexp, modf - manipulate parts of floating-point numbers

SYNOPSIS

double frexp (value, eptr)
double value;
int *eptr;

double ldexp (value, exp)
double value;
int exp;

double modf (value, iptr)
double value, *iptr;

DESCRIPTION

Every non-zero number can be written uniquely as $x * 2^n$, where the "mantissa" (fraction) x is in the range $0.5 \le |x| < 1.0$, 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.

The *ldexp* routine returns the quantity $value * 2^{exp}$.

The *modf* routine 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.**

NAME

fseek, rewind, ftell - reposition a file pointer in a stream

SYNOPSIS

#include <stdio.h>

int fseek (stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;

void rewind (stream)
FILE *stream;

long ftell (stream)
FILE *stream;

DESCRIPTION

The *fseek* routine sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, from the current position, or from the end of the file, according as *ptrname* has the value 0, 1, or 2.

The *rewind*(*stream*) routine is equivalent to *fseek*(*stream*, 0L, 0), except that no value is returned.

The *fseek* and *rewind* routines undo any effects of *ungetc*(3S).

After *fseek* or *rewind*, the next operation on a file opened for update may be either input or output.

The *ftell* routine returns the offset of the current byte relative to the beginning of the file associated with the named *stream*.

SEE ALSO

lseek(2), fopen(3S), popen(3S), stdio(3S), ungetc(3S).

DIAGNOSTICS

The *fseek* routine returns non-zero for improper seeks, otherwise zero. An improper seek can be, for example, an *fseek* done on a file that has not been opened via *fopen*; in particular, *fseek* may not be used on a terminal, or on a file opened via *popen*(3S).

WARNING

Although on the CTIX system and other systems derived from the UNIX system, an offset returned by *ftell* is measured in bytes, and it is permissible to seek to positions relative to that offset, portability to non-UNIX systems requires that an offset be used by *fseek* directly. Arithmetic may not meaningfully be performed on such an offset, which is not necessarily measured in bytes.

NAME

ftw - walk a file tree

SYNOPSIS

```
#include <ftw.h>

int ftw (path, fn, depth)
char *path;
int (*fn) ( );
int depth;
```

DESCRIPTION

The *ftw* routine recursively descends the directory hierarchy rooted in *path*. For each object in the hierarchy, *ftw* calls *fn*, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a **stat** structure [see *stat*(2)] containing information about the object, and an integer. Possible values of the integer, defined in the <ftw.h> header file, are FTW_F for a file, FTW_D for a directory, FTW_DNR for a directory that cannot be read, and FTW_NS for an object for which *stat* could not be executed successfully. If the integer is FTW_DNR, descendants of that directory will not be processed. If the integer is FTW_NS, the **stat** structure will contain garbage. An example of an object that would cause FTW_NS to be passed to *fn* would be a file in a directory with read but without execute (search) permission.

The *ftw* routine visits a directory before visiting any of its descendants.

The tree traversal continues until the tree is exhausted, an invocation of *fn* returns a non-zero value, or some error is detected within *ftw* (such as an I/O error). If the tree is exhausted, *ftw* returns zero. If *fn* returns a non-zero 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* routine uses one file descriptor for each level in the tree. The *depth* argument limits the number of file descriptors so used. If *depth* is zero or negative, the effect is the same as if it were 1. *Depth* must not be greater than the number of file descriptors currently available for use. The *ftw* routine runs more quickly if *depth* is at least as large as the number of levels in the tree.

SEE ALSO

stat(2), malloc(3C).

CAVEAT

The *ftw* routine uses *malloc*(3C) to allocate dynamic storage during its operation. If *ftw* is forcibly terminated, such as by *longjmp* being executed by *fn* or an interrupt routine, *ftw* will not have a chance to free that storage, so it will remain permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have *fn* return a non-zero value at its next invocation.

BUGS

Because *ftw* is recursive, it is possible for it to terminate with a memory fault when applied to very deep file structures.

NAME

gamma - log gamma function

SYNOPSIS

#include <math.h>

double gamma (x)
double x;

extern int signgam;

DESCRIPTION

The *gamma* routine returns $\ln(|\Gamma(x)|)$, where $\Gamma(x)$ is defined as $\int_0^\infty e^{-t}t^{x-1}\,dt$.

The sign of $\Gamma(x)$ is returned in the external integer *signgam*. The argument x cannot 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*(3M) to return a range error, and is defined in the *<values.h>* header file.

SEE ALSO

exp(3M), matherr(3M), values(5).

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.

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*(3M).

NAME

getc, getchar, fgetc, getw - get character or word from a stream

SYNOPSIS

#include <stdio.h>

int getc (stream)
FILE *stream;

int getchar ()

int fgetc (stream)
FILE *stream;

int getw (stream)
FILE *stream;

DESCRIPTION

The *getc* routine returns the next character (that is, byte) from the named input *stream*, as an integer. It also moves the file pointer, if defined, ahead one character in *stream.* The *getchar* variable is defined as *getc(stdin)*; *getc* and *getchar* are macros.

The *fgetc* routine behaves like *getc*, but it is a function rather than a macro; it 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* routine returns the next word (for example, integer) from the named input *stream.* The routine 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. It assumes no special alignment in the file.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), gets(3S), putc(3S), scanf(3S), stdio(3S).

DIAGNOSTICS

These functions return the constant **EOF** at end-of-file or upon an error. Because **EOF** is a valid integer, *ferror*(3S) should be used to detect *getw* errors.

CAVEATS

Because it is implemented as a macro, *getc* evaluates a *stream* argument more than once. In particular, **getc(*f++)** does not work sensibly; *fgetc* should be used instead.

Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent and cannot be read using *getw* on a different processor.

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 sign extension of a character on widening to integer is machine-dependent.

NAME

getcwd - get path-name of current working directory

SYNOPSIS

```
char *getcwd (buf, size)
char *buf;
int size;
```

DESCRIPTION

The *getcwd* routine returns a pointer to the current directory path name. The value of *size* must be at least two greater than the length of the path-name to be returned.

If *buf* is a NULL pointer, *getcwd* gets *size* bytes of space using *malloc*(3C). In this case, the pointer returned by *getcwd* can be used as the argument in a subsequent call to *free*.

The function is implemented by using *popen*(3S) to pipe the output of the *pwd*(1) command into the specified string space.

EXAMPLE

```
void exit(), perror();
.
.
.
if ((cwd = getcwd((char *)NULL, 64)) == NULL) {
        perror("pwd");
        exit(2);
}
printf("%s\n", cwd);
```

SEE ALSO

pwd(1), malloc(3C), popen(3S).

DIAGNOSTICS

The *getcwd* routine returns NULL with *errno* set to ERANGE if *size* is not large enough, or if an error occurs in a lower-level function.

NAME

getenv - return value for environment name

SYNOPSIS

char *getenv (name)
char *name;

DESCRIPTION

The *getenv* routine searches the environment list [see *environ*(5)] for a string of the form *name=value*, and returns a pointer to the *value* in the current environment if such a string is present; otherwise, it returns a NULL pointer.

SEE ALSO

exec(2), putenv(3C), environ(5).

NAME

getgrent, getgrgid, getgrnam, setgrent, endgrent, fgetgrent - get group file entry

SYNOPSIS

#include <grp.h>

struct group *getgrent ( )

struct group *getgrgid (gid)
int gid;

struct group *getgrnam (name)
char *name;

void setgrent ( )

void endgrent ( )

struct group *fgetgrent (f)
FILE *f;

DESCRIPTION

The *getgrent*, *getgrgid*, and *getgrnam* routines each return a pointer 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 first called, *getgrent* returns a pointer to the first group structure in the file; thereafter, it returns a pointer to the next group structure in the file; so, successive calls can be used to search the entire file. The *getgrgid* routine searches from the beginning of the file until a numerical group ID matching *gid* is found and returns a pointer to the particular structure in which it was found. The *getgrnam* routine searches from the beginning of the file until a group name matching *name* is found and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setgrent* has the effect of rewinding the group file to allow repeated searches. The *endgrent* routine can be called to close the group file when processing is complete.

The *fgetgrent* routine returns a pointer to the next group structure in the stream *f*, which matches the format of **/etc/group**.

FILES

/etc/group

SEE ALSO

getlogin(3C), getpwent(3C), group(4).

DIAGNOSTICS

A **NULL** pointer is returned on **EOF** or error.

CAVEAT

All information is contained in a static area, so it must be copied if it is to be saved.

WARNING

The above routines use **<stdio.h>**, which causes them to increase the size of programs, not otherwise using standard I/O, more than might be expected.

NAME

gethostbyname, gethostbyaddr, gethostent, sethostent, endhostent - get network host entry

SYNOPSIS

```
#include <netdb.h>

extern int h_errno;

struct hostent *gethostbyname(name)
char *name;

struct hostent *gethostbyaddr(addr, len, type)
char *addr; int len, type;

struct hostent *gethostent( )

sethostent(stayopen)
int stayopen;

endhostent( )
```

DESCRIPTION

The *gethostbyname* and *gethostbyaddr* routines each return a pointer to an object with the structure illustrated below. This structure contains either the information obtained from the name server, *named*(1M), or broken-out fields from a line in */etc/hosts,* (if the local name server is not running). *gethostbyname* takes a pointer to the ascii name of the host or one of its aliases. *gethostbyaddr* takes a pointer to address-family specific data; if type is AF_INET the data should be a **struct in_addr.**

```
struct hostent {
        char    *h_name;        /* official name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* host address type */
        int     h_length;       /* length of address */
        char    **h_addr_list;  /* list of addresses from name server */
};
#define         h_addr  h_addr_list[0]          /* address, for backward */
                                                /* compatibility */
```

The members of this structure follow:

h_name	Official name of the host.
h_aliases	A zero terminated array of alternate names for the host.
h_addrtype	The type of address being returned; currently it is always AF_INET.

h_length	The length, in bytes, of the address.
h_addr_list	A zero terminated array of network addresses for the host. Host addresses are returned in network byte order.
h_addr	The first address in h_addr_list; this is for backward compatiblity.

The *sethostent* routine allows a request for the use of a connected socket using TCP for queries. If the *stayopen* flag is non-zero, this sets the option to send all queries to the name server using TCP and to retain the connection after each call to *gethostbyname* or *gethostbyaddr*.

The *endhostent* routine closes the TCP connection.

FILES

/etc/hosts

SEE ALSO

named(1M), inet(3), resolver(3), hosts(4).
CTIX Network Programmer's Primer.

DIAGNOSTICS

Error return or EOF status from *gethostbyname* and *gethostbyaddr* is indicated by return of a null pointer. The external integer *h_errno* can then be checked to see whether this is a temporary failure or an invalid or unknown host.

The *h_errno* member can have the following values:

HOST_NOT_FOUND	No such host is known.
TRY_AGAIN	This is usually a temporary error, and it means that the local server did not receive a response from an authoritative server. A retry at some later time may succeed.
NO_RECOVERY	This is a non-recoverable error.
NO_ADDRESS	The requested name is valid but does not have an IP address; this is not a temporary error. This means another type of request to the name server will result in an answer.

WARNINGS

All information is contained in a static area so it must be copied if it is to be saved.

Only the Internet address format is currently understood.

NAME

getlogin - get login name

SYNOPSIS

char *getlogin ( );

DESCRIPTION

The *getlogin* routine returns a pointer to the login name as found in **/etc/utmp**. It can be used in conjunction with *getpwnam* to locate the correct password file entry when the same user ID is shared by several login names.

If *getlogin* is called within a process that is not attached to a terminal, it returns a NULL pointer. The correct procedure for determining the login name is to call *cuserid*, or to call *getlogin* and if it fails to call *getpwuid*.

FILES

/etc/utmp

SEE ALSO

cuserid(3S), getgrent(3C), getpwent(3C), utmp(4).

DIAGNOSTICS

The *getlogin* routine returns the NULL pointer if *name* is not found.

CAVEAT

The return values point to static data whose content is overwritten by each call.

NAME

getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent - get network entry

SYNOPSIS

```
#include <netdb.h>

struct netent *getnetent ( )

struct netent *getnetbyname (name)
char *name;

struct netent *getnetbyaddr (net, type)
long net;

setnetent (stayopen)
int stayopen

endnetent ( )
```

DESCRIPTION

The *getnetent*, *getnetbyname*, and *getnetbyaddr* routines each return a pointer to an object with the following structure containing the broken-out fields of a line in the network database, /etc/networks.

```
struct netent {
        char    *n_name;        /* official name of net */
        char    **n_aliases;    /* alias list */
        int     n_addrtype;     /* net number type */
        long    n_net;          /* net number */
};
```

The members of this structure are:

n_name	The official name of the network.
n_aliases	A zero-terminated list of alternate names for the network.
n_addrtype	The type of the network number returned; currently only AF_INET.
n_net	The network number. Network numbers are returned in machine byte order.

The *getnetent* routine reads the next line of the file, opening the file if necessary.

The *setnetent* routine opens and rewinds the file. If the *stayopen* flag is non-zero, the network database is not closed after each call to *getnetent* (either directly or indirectly through one of the other getnet calls).

The *endnetent* routine closes the file.

The *getnetbyname* and *getnetbyaddr* routines sequentially search from the beginning of the file until a matching net name or net address is found, or until EOF is encountered. Network numbers are supplied in host order. If type AF_NET is supplied to *gethostbyaddr*, net should be right-justified: that is, net 3 would be a 3 rather than 0x3000000.

FILES

/etc/networks

SEE ALSO

networks(4).
CTIX Network Programmer's Primer.

DIAGNOSTICS

A null pointer (0) is returned on EOF or error.

BUGS

All information is contained in a static area, so it must be copied if it is to be saved.

Only Internet network numbers are currently understood.

NAME

getopt - get option letter from argument vector

SYNOPSIS

int getopt (argc, argv, optstring)
int argc;
char **argv, *opstring;

extern char *optarg;
extern int optind, opterr;

DESCRIPTION

The *getopt* routine returns the next option letter in *argv* that matches a letter in *optstring*. It supports all the rules of the command syntax standard [see *intro*(1)]. So all new commands will adhere to the command syntax standard, they should use *getopts*(1) or *getopt*(3C) to parse positional parameters and check for options that are legal for that command.

The *optstring* argument must contain the option letters the command using *getopt* can recognize; 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.

The **optarg** variable is set to point to the start of the option-argument on return from *getopt*.

The *getopt* routine places in **optind** the *argv* index of the next argument to be processed. The **optind** variable 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 "--" can be used to delimit the end of the options; when it is encountered, -1 will be returned, and "--" will be skipped.

DIAGNOSTICS

The *getopt* routine 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 can be disabled by setting **opterr** to **0**.

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 (2);
          }
          for ( ; optind < argc; optind++) {
                    if (access(argv[optind], 4)) {
          . . .
}
```

SEE ALSO

getopts(1), intro(1).

WARNINGS

Although the following command syntax rule [see *intro*(1)] 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 -aboarg file
```

(Rule 5 violation: options with option-arguments must not be grouped with other options).

```
cmd -ab -oarg file
```

(Rule 6 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.

NAME

getpass - read a password

SYNOPSIS

char *getpass (prompt)
char *prompt;

DESCRIPTION

The *getpass* routine reads up to a newline or **EOF** from the file **/dev/tty**, after prompting on the standard error output with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most 8 characters. If **/dev/tty** cannot be opened, a NULL pointer is returned. An interrupt will terminate input and send an interrupt signal to the calling program before returning.

FILES

/dev/tty

SEE ALSO

crypt(3C).

WARNING

The above routine uses **<stdio.h>**, which causes it to increase the size of programs not otherwise using standard I/O, more than might be expected.

CAVEAT

The return value points to static data whose content is overwritten by each call.

NAME

getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent - get protocol entry

SYNOPSIS

#include <netdb.h>

struct protoent *getprotoent ( )

struct protoent *getprotobyname (name)
char *name;

struct protoent *getprotobynumber (proto)
int proto;

setprotoent (stayopen)
int stayopen

endprotoent ( )

DESCRIPTION

The *getprotoent*, *getprotobyname*, and *getprotobynumber* routines each return a pointer to an object with the following structure containing the broken-out fields of a line in the network protocol database, **/etc/protocols**.

```
struct   protoent {
         char   *p_name;       /* official name of protocol */
         char   **p_aliases;   /* alias list */
         long   p_proto;       /* protocol number */
};
```

The members of this structure follow:

p_name	The official name of the protocol.
p_aliases	A zero-terminated list of alternate names for the protocol.
p_proto	The protocol number.

The *getprotoent* routine reads the next line of the file, opening the file if necessary.

The *setprotoent* routine opens and rewinds the file. If the *stayopen* flag is non-zero, the network database will not be closed after each call to *getprotoent* (either directly or indirectly through one of the other getproto calls).

The *endprotoent* routine closes the file.

The *getprotobyname* and *getprotobynumber* routines sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until EOF is encountered.

FILES

/etc/protocols

SEE ALSO

protocols(4).
CTIX Network Programmer's Primer.

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area, so it must be copied if it is to be saved.

Only the Internet protocols are currently understood.

NAME

getpw - get name from UID

SYNOPSIS

int getpw (uid, buf)
int uid;
char *buf;

DESCRIPTION

The *getpw* routine searches the password file for a user ID number that equals *uid*, copies the line of the password file in which *uid* was found into the array pointed to by *buf*, and returns 0. The routine returns non-zero if *uid* cannot be found.

This routine is included only for compatibility with prior systems and should not be used; see *getpwent*(3C) for routines to use instead.

FILES

/etc/passwd

SEE ALSO

getpwent(3C), passwd(4).

DIAGNOSTICS

The *getpw* returns non-zero on error.

WARNING

If a program not otherwise using standard I/O uses this routine, the size of the program increases more than might be expected.

NAME

getpwent, getpwuid, getpwnam, setpwent, endpwent, fgetpwent - get password file entry

SYNOPSIS

#include <pwd.h>

struct passwd *getpwent ( )

struct passwd *getpwuid (uid)
int uid;

struct passwd *getpwnam (name)
char *name;

void setpwent ( )

void endpwent ( )

struct passwd *fgetpwent (f)
FILE *f;

DESCRIPTION

The *getpwent*, *getpwuid* and *getpwnam* routines each return a pointer to an object with the following structure containing the broken-out fields of a line in the /etc/passwd file. Each line in the file contains a "passwd" structure, declared in the *<pwd.h>* header file:

```
struct passwd {
          char      *pw_name;
          char      *pw_passwd;
          int       pw_uid;
          int       pw_gid;
          char      *pw_age;
          char      *pw_comment;
          char      *pw_gecos;
          char      *pw_dir;
          char      *pw_shell;
};
```

This structure is declared in *<pwd.h>* so you need not redeclare it. The field meanings are described in *passwd*(4).

When first called, *getpwent* returns a pointer to the first passwd structure in the file; thereafter, it returns a pointer to the next passwd structure in the file so successive calls can be used to search the entire file. The *getpwuid* routine searches from the beginning of the file until a numerical user ID matching *uid* is found; the routine then returns a pointer to the structure in which it was found.

The *getpwnam* routine searches from the beginning of the file until a login name matching *name* is found; the routine then returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setpwent* rewinds the password file to allow repeated searches. A call to *endpwent* closes the password file when processing is complete.

The *fgetpwent* routine returns a pointer to the next passwd structure in the stream *f* that matches the format of **/etc/passwd**.

FILES

/etc/passwd

SEE ALSO

getgrent(3C), getlogin(3C), getspent(3X), putpwent(3C), passwd(4).

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

WARNING

If a program not otherwise using standard I/O uses this routine, the size of the program increases more than might be expected.

CAVEAT

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

getrpcent, getrpcbyname, getrpcbynumber - get rpc entry

SYNOPSIS

#include <netdb.h>

struct rpcent *getrpcent()

struct rpcent *getrpcbyname(name)
char *name;

struct rpcent *getrpcbynumber(number)
int number;

setrpcent(stayopen)
int stayopen

endrpcent()

DESCRIPTION

The *getrpcent*, *getrpcbyname*, and *getrpcbynumber* routines each return a pointer to an object with the following structure containing the broken-out fields of a line in the rpc program number database, */etc/rpc*.

```
struct  rpcent {
        char  *r_name;/* name of server for this rpc program */
        char  **r_aliases; /* alias list */
        long  r_number; /* rpc program number */
};
```

The members of this structure follow:

r_name	The name of the server for this rpc program.
r_aliases	A zero terminated list of alternate names for the rpc program.
r_number	The rpc program number for this service.

The *getrpcent* routine reads the next line of the file, opening the file if necessary.

The *setrpcent* routine opens and rewinds the file. If the *stayopen* flag is non-zero, the net database will not be closed after each call to *getrpcent* (either directly or indirectly through one of the other *qgetrpc* calls).

The *endrpcent* routine closes the file.

The *getrpcbyname* and *getrpcbynumber* routines sequentially search from the beginning of the file until a matching *rpc* program name or program number is found, or until EOF is encountered.

FILES

/etc/rpc

SEE ALSO

rpc(4), rpcinfo(1M).

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved.

NAME

getrpcport - get RPC port number

SYNOPSIS

int getrpcport(host, prognum, versnum, proto)
char *host;
int prognum, versnum, proto;

DESCRIPTION

The *getrpcport* routine returns the port number for version *versnum* of the RPC program *prognum* running on *host* and using protocol *proto*. The routine returns 0 if it cannot contact the portmapper, or if *prognum* is not registered; if *prognum* is registered but not with version *versnum*, *getrpcport* returns that port number.

SEE ALSO

portmap(1M), rpcinfo(1M), rpc(4).
CTIX Network Programmer's Guide

NAME

gets, fgets - get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char *gets (s)
char *s;

char *fgets (s, n, stream)
char *s;
int n;
FILE *stream;
```

DESCRIPTION

The *gets* routine reads characters from the standard input stream, *stdin,* into the array pointed to by *s*, until a newline character is read or an end-of-file condition is encountered. The newline character is discarded and the string is terminated with a null character.

The *fgets* routine reads characters from the *stream* into the array pointed to by *s*, until *n*-1 characters are read, or a newline character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null character.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), getc(3S), scanf(3S), stdio(3S).

DIAGNOSTICS

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a NULL pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a NULL pointer is returned. Otherwise, *s* is returned.

NAME

getservent, getservbyport, getservbyname, setservent, endservent - get service entry

SYNOPSIS

#include <netdb.h>

struct servent *getservent ( )

struct servent *getservbyname (name, proto)
char *name, *proto;

struct servent *getservbyport (port, proto)
int port; char *proto;

setservent (stayopen)
int stayopen

endservent ( )

DESCRIPTION

The *getservent*, *getservbyname*, and *getservbyport* routines each return a pointer to an object with the following structure containing the broken-out fields of a line in the network services database, /etc/services.

```
struct    servent {
     char  *s_name;     /* official name of service */
     char  **s_aliases; /* alias list */
     long  s_port;      /* port service resides at */
     char  *s_proto;    /* protocol to use */
};
```

The members of this structure follow:

s_name	The official name of the service.
s_aliases	A zero-terminated list of alternate names for the service.
s_port	The port number at which the service resides. Port numbers are returned in network byte order.
s_proto	The name of the protocol to use when contacting the service.

The *getservent* routine reads the next line of the file, opening the file if necessary.

The *setservent* routine opens and rewinds the file. If the *stayopen* flag is non-zero, the network database is not be closed after each call to *getservent* (either directly or indirectly through one of the other getserv calls).

The *endservent* routine closes the file.

The *getservbyname* and *getservbyport* routines sequentially search from the beginning of the file until a matching protocol name or port number is found, or until EOF is encountered.

If a protocol name is also supplied (non-NULL), searches must also match the protocol.

FILES

/etc/services

SEE ALSO

getprotoent(3), services(4).
CTIX Network Programmer's Primer.

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

getspent, getspnam, setspent, endspent, fgetspent, lckpwdf, ulckpwdf - get shadow password file entry

SYNOPSIS

```
#include <shadow.h>

struct spwd *getspent ()

struct spwd *getspnam (name)
char *name;

int lckpwdf ()
int ulckpwdf ()

void setspent ()

void endspent ()

struct spwd *fgetspent (fp)
FILE *fp;
```

DESCRIPTION

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 (**spwd**), 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; this way, 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 then returns a pointer to the particular structure in which it was found. The *getspent* and *getspnam* routines populate the **sp_min** or **sp_max** field with -1 if the corresponding field in **/etc/shadow** is empty. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

The **/etc/.pwd.lock** file is the lock file, which is used to coordinate modification access to the password files in **/etc/passwd** and **/etc/shadow**. The *lckpwdf*()

and *ulckpwdf*( ) routines are used to gain modification access to the password files, through the lock file. A process first uses *lckpwdf*( ) to lock the lock file, thereby gaining exclusive rights to modify the **/etc/passwd** or **/etc/shadow** file. Upon completing modifications, a process should release the lock on the lock file by using *ulckpwdf*( ). This lock mechanism prevents simultaneous modification of the password files.

The *lckpwdf*( ) routine attempts to lock the file **/etc/.pwd.lock**. If the file is already locked, *lckpwdf*( ) tries for 15 seconds to lock the file. If unsuccessful, *lckpwdf*( ) returns a -1; if successful within 15 seconds, *lckpwdf*( ) returns a return code other than -1.

The *ulckpwdf*( ) routine attempts to unlock the file **/etc/.pwd.lock**. If successful, *ulckpwdf*( ) returns a 0; if unsuccessful (if the file is not locked), *ulckpwdf*( ) 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**.

FILES

/etc/shadow
/etc/passwd
/etc/.pwd.lock

SEE ALSO

putspent(3X), shadow(4).

DIAGNOSTICS

A NULL pointer is returned on **EOF** or error.

CAVEAT

All information is contained in a static area, so it must be copied if it is to be saved.

WARNING

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 for internal use only; compatibility is not guaranteed.

NAME

getut: getutent, getutid, getutline, pututline, setutent, endutent, utmpname - access utmp file entry

SYNOPSIS

#include <utmp.h>

struct utmp *getutent ( )

struct utmp *getutid (id)
struct utmp *id;

struct utmp *getutline (line)
struct utmp *line;

void pututline (utmp)
struct utmp *utmp;

void setutent ( )

void endutent ( )

void utmpname (file)
char *file;

DESCRIPTION

The *getutent*, *getutid* and *getutline* 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 entry from a *utmp*-like file. If the file is not already open, it opens it. If it reaches the end of the file, it fails.

The *getutid* routine searches forward from the current point in the *utmp* file until it finds an entry with a *ut_type* matching *id->ut_type* if the type specified is RUN_LVL, BOOT_TIME, OLD_TIME or NEW_TIME. If the type specified in *id* is INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS or DEAD_PROCESS, *getutid* returns a pointer to the first entry whose type is one of these four and whose *ut_id* field matches *id->ut_id*. If the end of file is reached without a match, it fails.

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 *line->ut_line* string. If the end of file is reached without a match, it fails.

The *pututline* routine writes out the supplied *utmp* structure into the *utmp* file. It uses *getutid* to search forward for the proper place if it finds that it is not already at the proper place. It is expected that normally the user of *pututline* will have searched for the proper entry using one of the *getut* routines. If so, *pututline* will not search. If *pututline* does not find a matching slot for the new entry, it will add a new entry to the end of the file.

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.

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 will be **/etc/wtmp**. If the file does not exist, this will not be apparent until the first attempt to refer to the file is made. The *utmpname* routine does not open the file. It just closes the old file if it is currently open and saves the new file name.

FILES

/etc/utmp
/etc/wtmp

SEE ALSO

ttyslot(3C), utmp(4).

DIAGNOSTICS

A NULL pointer is returned upon failure to read, whether for permissions or having reached the end of file, or upon failure to write.

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* sees the routine examine the static structure before performing more I/O. If the contents of the static structure match what it is searching for, it looks no further. For this reason, to use *getutline* to search for multiple occurrences, it would be necessary to zero out the static after each success, or *getutline* would just return the same pointer over and over again. There is one exception to the rule about removing the structure before further reads are done. The implicit read done by *pututline* (if it finds that it is not already at the correct place in the file) will not hurt the contents of the static structure returned by the *getutent*, *getutid* or *getutline* routines, if the user has just modified those contents and passed the pointer back to *pututline*.

These routines use buffered standard I/O for input, but *pututline* uses an unbuffered non-standard write to avoid race conditions between processes trying to modify the *utmp* and *wtmp* files.

NAME

hsearch, hcreate, hdestroy - manage hash search tables

SYNOPSIS

#include <search.h>

ENTRY *hsearch (item, action)
ENTRY item;
ACTION action;

int hcreate (nel)
unsigned nel;

void hdestroy ( )

DESCRIPTION

The *hsearch* routine 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.

The *hcreate* routine allocates sufficient space for the table, and must be called before *hsearch* is used. *Nel* is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

The *hdestroy* routine destroys the search table and can be followed by another call to *hcreate*.

NOTES

The *hsearch* routine uses *open addressing* with a *multiplicative* hash function. However, its source code has many other options available which the user can select by compiling the *hsearch* source with the following symbols defined to the preprocessor:

DIV Use the *remainder modulo table size* as the hash function instead of the multiplicative algorithm.

USCR Use a User Supplied Comparison Routine for ascertaining table membership. The routine should be named *hcompar* and should behave in a manner similar to *strcmp* [see *string*(3C)].

CHAINED Use a linked list to resolve collisions. If this option is selected, the following other options become available.

- **START** Place new entries at the beginning of the linked list (default is at the end).
- **SORTUP** Keep the linked list sorted by key in ascending order.
- **SORTDOWN** Keep the linked list sorted by key in descending order.

Additionally, there are preprocessor flags for obtaining debugging printout (**-DDEBUG**) and for including a test driver in the calling routine (**-DDRIVER**). The source code should be consulted for further details.

EXAMPLE

The following example will read in strings followed by two numbers and store them in a hash table, discarding duplicates. It will then read in strings and find the matching entry in the hash table and print it.

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

struct info {                   /* this is the info stored in the */
        int age, room;  /* table other than the key. */
};
#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)
             }
          }
     }
```

SEE ALSO

bsearch(3C), lsearch(3C), malloc(3C), malloc(3X), string(3C), tsearch(3C).

DIAGNOSTICS

The *hsearch* routine 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.

The *hcreate* routine returns zero if it cannot allocate sufficient space for the table.

CAVEAT

Only one hash search table can be active at any given time.

WARNING

The *hsearch* and *hcreate* routines use *malloc*(3C) to allocate space.

NAME

hypot - Euclidean distance function

SYNOPSIS

#include <math.h>

double hypot (x, y)
double x, y;

DESCRIPTION

The *hypot* routine returns the following, taking precautions against unwarranted overflows:

sqrt(x * x + y * y),

SEE ALSO

matherr(3M).

DIAGNOSTICS

When the correct value would overflow, *hypot* returns **HUGE** and sets *errno* to **ERANGE.**

These error-handling procedures may be changed with the function *matherr*(3M).

NAME

inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof - Internet address manipulation routines

SYNOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_addr(cp)
char *cp;

int inet_network(cp)
char *cp;

char *inet_ntoa(in)
struct inet_addr in;

struct in_addr inet_makeaddr(net, lna)
int net, lna;

int inet_lnaof(in)
struct in_addr in;

int inet_netof(in)
struct in_addr in;
```

DESCRIPTION

The routines *inet_addr* and *inet_network* each interpret character strings representing numbers expressed in the Internet standard dot notation, returning numbers suitable for use as Internet addresses and Internet network numbers, respectively. The **struct in_addr** returned by inet_addr is cast to an unsigned long for error checking (See Diagnostics, below.) The routine *inet_ntoa* takes an Internet address and returns an ASCII string representing the address in dot notation. The routine *inet_makeaddr* takes an Internet network number and a local network address and constructs an Internet address from it. The routines *inet_netof* and *inet_lnaof* break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet addresses are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES

Values specified using the dot notation take one of the following forms:

a.b.c.d
a.b.c
a.b
a

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right most two bytes of the network address. This makes the three part address format convenient for specifying Class B network addresses as **128**.*net*.*host*.

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses as *net*.*host*.

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as ''parts'' in a ''dot'' notation may be decimal, octal, or hexadecimal, as specified in the C language (that is, a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

EXAMPLE

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <sys/in.h>
#include <arpa/inet.h>
int check_address(address_s)
char      address_s[];      /* ascii string (e.g., first token
                                   * in /etc/hosts entry)
                                   */
{
          struct in_addr          addr;
          int                     rc;

          rc = 0;
```

```
                /* check that address is in valid format and get binary
                 * form to pass to gethostbyaddr
                 */
                addr.s_addr =  inet_addr (address_s);
                if (addr.s_addr == -1)
                {
                        fprintf (stderr,
                        "Address '%s' not in valid internet format\n",
                                address_s);
                        rc = -1;
                        return (rc);
                }

                /* check that address is not already in host database */
                if ((int) gethostbyaddr (&addr,
                                sizeof (addr), AF_INET) != NULL)
                {
                        fprintf (stderr,
                                "Address '%s' already in use\n",
                                address_s);
                        rc = -1;
                }

                return (rc);

        } /* check_address */
```

SEE ALSO

gethostbyname(3), getnetent(3), hosts(4), networks(4), inet(7).
CTIX Network Programmer's Primer.

DIAGNOSTICS

The value -1 is returned by *inet_addr* and *inet_network* for malformed requests.

BUGS

The string returned by *inet_ntoa* resides in a static memory area.

NAME

isnan: isnand, isnanf - test for floating point NaN (Not-A-Number)

SYNOPSIS

#include <ieeefp.h>

int isnand (dsrc)
double dsrc;

int isnanf (fsrc)
float fsrc;

DESCRIPTION

The *isnand and isnanf* routines 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 *isnanf()* routine is implemented as a macro included in **<ieeefp.h>**.

SEE ALSO

fpgetround(3C).

NAME

l3tol, ltol3 - convert between 3-byte integers and long integers

SYNOPSIS

```
void l3tol (lp, cp, n)
long *lp;
char *cp;
int n;

void ltol3 (cp, lp, n)
char *cp;
long *lp;
int n;
```

DESCRIPTION

The *l3tol* routine 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*.

The *ltol3* routine performs the reverse conversion from long integers (*lp*) to three-byte integers (*cp*).

These functions are useful for file-system maintenance where the block numbers are three bytes long.

SEE ALSO

fs(4).

CAVEAT

Because of possible differences in byte ordering, the numerical values of the long integers are machine-dependent.

NAME

ldahread - read the archive header of a member of an archive file

SYNOPSIS

#include <stdio.h>
#include <ar.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldahread (ldptr, arhead)
LDFILE *ldptr;
ARCHDR *arhead;

DESCRIPTION

If **TYPE**(*ldptr*) is the archive file magic number, *ldahread* reads the archive header of the common object file currently associated with *ldptr* into the area of memory beginning at *arhead*.

The *ldahread* routine returns **SUCCESS** or **FAILURE**; it fails if **TYPE**(*ldptr*) does not represent an archive file, or if it cannot read the archive header.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ar(4), ldfcn(4).

NAME

ldclose, ldaclose - close a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldclose (ldptr)
LDFILE *ldptr;

int ldaclose (ldptr)
LDFILE *ldptr;

DESCRIPTION

The *ldopen*(3X) and *ldclose* routines 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 free the memory allocated to the **LDFILE** structure associated with *ldptr*. If **TYPE**(*ldptr*) is the magic number of an archive file, and if there are any more files in the archive, *ldclose* reinitializes **OFFSET**(*ldptr*) to the file address of the next archive member and returns **FAILURE**. The **LDFILE** structure is prepared for a subsequent *ldopen*(3X). In all other cases, *ldclose* returns **SUCCESS**.

The *ldaclose* routine closes the file and frees the memory allocated to the **LDFILE** structure associated with *ldptr* regardless of the value of **TYPE***(ldptr)*. The *ldaclose* routine always returns **SUCCESS**. The function is often used in conjunction with *ldaopen*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

fclose(3S), ldopen(3X), ldfcn(4).

NAME

ldfhread - read the file header of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldfhread (ldptr, filehead)
LDFILE *ldptr;
FILHDR *filehead;
```

DESCRIPTION

The *ldfhread* routine reads the file header of the common object file currently associated with *ldptr* into the area of memory beginning at *filehead*.

The *ldfhread* routine 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 (4)]. The information in any field, *fieldname*, of the file header may be accessed using **HEADER(ldptr).fieldname.**

The program must be loaded with the object file access routine library **libld.a.**

FILES

/usr/lib/libld.a

SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4).

NAME

ldgetname - retrieve symbol name for common object file symbol table entry

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

char *ldgetname (ldptr, symbol)
LDFILE *ldptr;
SYMENT *symbol;

DESCRIPTION

The *ldgetname* routine 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* routine can be used to retrieve names from object files without any backward compatibility problems. The *ldgetname* routine returns NULL (defined in **stdio.h**) for an object file if the name cannot be retrieved, as in the following cases:

- 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).
- 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*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldtbseek(3X), ldfcn(4).

NAME

ldlread, ldlinit, ldlitem - manipulate line number entries of a common object file function

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <linenum.h>
#include <ldfcn.h>

int ldlread(ldptr, fcnindx, linenum, linent)
LDFILE *ldptr;
long fcnindx;
unsigned short linenum;
LINENO *linent;

int ldlinit(ldptr, fcnindx)
LDFILE *ldptr;
long fcnindx;

int ldlitem(ldptr, linenum, linent)
LDFILE *ldptr;
unsigned short linenum;
LINENO *linent;
```

DESCRIPTION

The *ldlread* routine searches the line number entries of the common object file currently associated with *ldptr*. It 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 routine reads the entry with the smallest line number equal to or greater than *linenum* into the memory beginning at *linent*.

The *ldlinit* and *ldlitem* routines together perform exactly the same function as *ldlread.* After an initial call to *ldlread* or *ldlinit, ldlitem* can be used to retrieve a series of line number entries associated with a single function. The *ldlinit* routine locates the line number entries for the function identified by *fcnindx*; *ldlitem* finds and reads the entry with the smallest line number equal to or greater than *linenum* into the memory beginning at *linent*.

The *ldlread*, *ldlinit*, and *ldlitem* routines each return either SUCCESS or FAILURE. The *ldlread* routine 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*. The *ldlinit* routine 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. The *ldlitem* routine 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**.

FILES

/usr/lib/libld.a

SEE ALSO

ldclose(3X), ldopen(3X), ldtbindex(3X), ldfcn(4).

NAME

ldlseek, ldnlseek - seek to line number entries of a section of a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldlseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnlseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;

DESCRIPTION

The *ldlseek* routine seeks to the line number entries of the section specified by *sectindx* of the common object file currently associated with *ldptr*.

The *ldnlseek* routine seeks to the line number entries of the section specified by *sectname*.

The *ldlseek* and *ldnlseek* routines return **SUCCESS** or **FAILURE**. The *ldlseek* routine 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*.

The program must be loaded with the object file access routine library **libld.a**.

FILES

/usr/lib/libld.a

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

NAME

ldohseek - seek to the optional file header of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldohseek (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

The *ldohseek* routine seeks to the optional file header of the common object file currently associated with *ldptr*. It returns **SUCCESS** or **FAILURE**. The routine fails if the object file has no optional header or if it cannot seek to the optional header.

The program must be loaded with the object file access routine library **libld.a**.

FILES

/usr/lib/libld.a

SEE ALSO

ldclose(3X), ldopen(3X), ldfhread(3X), ldfcn(4).

NAME

ldopen, ldaopen - open a common object file for reading

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

LDFILE *ldopen (filename, ldptr)
char *filename;
LDFILE *ldptr;

LDFILE *ldaopen (filename, oldptr)
char *filename;
LDFILE *oldptr;
```

DESCRIPTION

The *ldopen* and *ldclose*(3X) routines 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, *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* routines are designed to work in concert. The *ldclose* routine 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.

The following is a prototype for the use of *ldopen* and *ldclose*.

```
/* for each filename to be processed */
ldptr = NULL;
do
{
        if ( (ldptr = ldopen(filename, ldptr)) != NULL )
        {
                /* check magic number; process the file */
        }
} while (ldclose(ldptr) == FAILURE );
```

If the value of *oldptr* is not NULL, *ldaopen* opens *filename* anew and allocate and initialize a new **LDFILE** structure, copying the **TYPE**, **OFFSET**, and **HEADER** fields from *oldptr*. The *ldaopen* routine returns a pointer to the new **LDFILE** structure. This new pointer is independent of the old pointer, *oldptr*. The two pointers can be used concurrently to read separate parts of the object file. For example, one pointer can be used to step sequentially through the relocation information, while the other is used to read indexed symbol table entries.

Both *ldopen* and *ldaopen* open *filename* for reading. Both functions return NULL if *filename* cannot be opened, or if memory for the **LDFILE** structure cannot be allocated. A successful open does not insure that the given file is a common object file or an archived object file.

The program must be loaded with the object file access routine library **libld.a**.

FILES

/usr/lib/libld.a

SEE ALSO

fopen(3S), ldclose(3X), ldfcn(4).

NAME

ldrseek, ldnrseek - seek to relocation entries of a section of a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldrseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnrseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;

DESCRIPTION

The *ldrseek* routine seeks to the relocation entries of the section specified by *sectindx* of the common object file currently associated with *ldptr*.

The *ldnrseek* routine seeks to the relocation entries of the section specified by *sectname*.

The *ldrseek* and *ldnrseek* routines return **SUCCESS** or **FAILURE**. The *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*.

The program must be loaded with the object file access routine library **libld.a**.

FILES

/usr/lib/libld.a

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

NAME

ldshread, ldnshread - read an indexed/named section header of a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <scnhdr.h>
#include <ldfcn.h>

int ldshread (ldptr, sectindx, secthead)
LDFILE *ldptr;
unsigned short sectindx;
SCNHDR *secthead;

int ldnshread (ldptr, sectname, secthead)
LDFILE *ldptr;
char *sectname;
SCNHDR *secthead;

DESCRIPTION

The *ldshread* routine 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* routine reads the section header specified by *sectname* into the area of memory beginning at *secthead*.

The *ldshread* and *ldnshread* routines return **SUCCESS** or **FAILURE**. The *ldshread* routine 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*.

The program must be loaded with the object file access routine library **libld.a**.

FILES

/usr/lib/libld.a

SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4).

NAME

ldsseek, ldnsseek - seek to an indexed/named section of a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldsseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnsseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;

DESCRIPTION

The *ldsseek* routine seeks to the section specified by *sectindx* of the common object file currently associated with *ldptr*.

The *ldnsseek* routine seeks to the section specified by *sectname*.

The *ldsseek* and *ldnsseek* routines return SUCCESS or FAILURE. The *ldsseek* fails if *sectindx* is greater than the number of sections in the object file; *ldnsseek* 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*.

The program must be loaded with the object file access routine library **libld.a**.

FILES

/usr/lib/libld.a

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

NAME

ldtbindex - compute the index of a symbol table entry of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

long ldtbindex (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

ldtbindex returns the (**long**) index of the symbol table entry at the current position of the common object file associated with *ldptr*.

The index returned by *ldtbindex* can be used in subsequent calls to *ldtbread*(3X). However, since *ldtbindex* returns the index of the symbol table entry that begins at the current position of the object file, if *ldtbindex* is called immediately after a particular symbol table entry has been read, it will return the index of the next entry.

ldtbindex will fail if there are no symbols in the object file, or if the object file is not positioned at the beginning of a symbol table entry.

Note that the first symbol in the symbol table has an index of *zero*.

The program must be loaded with the object file access routine library **libld.a**.

FILES

/usr/lib/libld.a

SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldtbseek(3X), ldfcn(4).

NAME

ldtbread - read an indexed symbol table entry of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldtbread (ldptr, symindex, symbol)
LDFILE *ldptr;
long symindex;
SYMENT *symbol;
```

DESCRIPTION

ldtbread reads the symbol table entry specified by *symindex* of the common object file currently associated with *ldptr* into the area of memory beginning at *symbol* .

ldtbread returns SUCCESS or **FAILURE**. *ldtbread* will fail if *symindex* is greater than 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*.

The program must be loaded with the object file access routine library **libld.a**.

FILES

/usr/lib/libld.a

SEE ALSO

ldclose(3X), ldopen(3X), ldtbseek(3X), ldgetname(3X), ldfcn(4).

NAME

ldtbseek - seek to the symbol table of a common object file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldtbseek (ldptr)
LDFILE *ldptr;

DESCRIPTION

ldtbseek seeks to the symbol table of the common object file currently associated with *ldptr*.

ldtbseek returns **SUCCESS** or **FAILURE**. *ldtbseek* will fail if the symbol table has been stripped from the object file, or if it cannot seek to the symbol table.

The program must be loaded with the object file access routine library **libld.a**.

FILES

/usr/lib/libld.a

SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldfcn(4).

NAME

libdev - manipulate Volume Home Blocks (VHB)

SYNOPSIS

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

struct vhbd *vhbd;
short sl, *slp;
char *s, *device;
int fd;

int gdnsec(vhbd, sl)
int gdstrk(vhbd, sl)
int gdftrk(vhbd, sl)
int gdnszc(vhbd)
int isdisk(fd)
struct vhbd *readvhb(s, sl)
struct vhbd *sreadvhb(device)
struct vhbd *freadvhb(fd, sl)
char *adevname(fd)
char *bdevname(s)
int dismnt(fd)
char *gdname(s, slp)
char *fgdname(fd, slp)
int gdnlblk(fd)
```

DESCRIPTION

Each argument in the above subroutines denotes:

vhbd	A pointer to a disk volume home block, as returned by *readvhb*, *sreadvhb*, or *freadvhb*.
sl	Slice number on the drive.
slp	Pointer to a slice number. This argument is actually used by the subroutine to return a slice number.
s	The name of a special file in **/dev/rdsk**. This filename is used to obtain a file descriptor to access a VHB. The name need not be for slice zero of the disk.
device	The name of a special file in **/dev/rdsk**. This filename is used to obtain a file descriptor to access a VHB. The name must be for slice zero of a disk.

fd	Open file descriptor for slice zero of a disk.

The subroutines in **/usr/lib/libdev.a** form a device and machine independent interface to the VHB of CTIX disks. The function of each subroutine is described below:

gdnsec	Returns the number of sectors in slice *sl* of the VHB indicated by *vhbd.*
gdstrk	Returns the starting track of slice *sl* of the VHB pointed to by *vhbd.*
gdftrk	Returns 1 if slice *sl* of the VHB pointed to by *vhbd* extends to the end of the disk.
gdnszc	Returns the number of sectors per cylinder.
isdisk	Returns 1 if the file descriptor *fd* is opened to a *special* disk device.
readvhb, *sreadvhb*, and *freadvhb*	Return a pointer to a VHB for the device described by their arguments.
adevname	Returns the character device name for the disk drive to which the file descriptor *fd* is opened.
bdevname	Returns the block device name for the disk drive that the string *s* names. The filename *S* can be for any slice on a raw or block device.
dismnt	Exercises the GDDISMNT ioctl call for the disk drive that the file descriptor to which *fd* is opened.
gdname	Returns the file name for the character special slice zero of a disk the filename *s* name a slice of. The value pointed to by **slp** is set to the slice number of the filename *s*.
fgdname	Performs as does gdname, but uses the file descriptor *fd* instead of the filename *s*.
gdnlblk	Returns the number of logical blocks in the slice to which the file descriptor *fd* is opened.

FILES

/dev/rdsk/c*?*d*?*s*?*
/dev/dsk/c*?*d*?*s*?*
/usr/lib/libdev.a

SEE ALSO

iv(1) disk(7).

NAME

lockf - record locking on files

SYNOPSIS

#include <unistd.h>

int lockf (fildes, function, size)
long size;
int fildes, function;

DESCRIPTION

The *lockf* command will allow sections of a file to be locked; advisory or mandatory write locks depending on the mode bits of the file [see *chmod*(2)]. Locking calls from other processes that to lock the locked file section will either return an error value or be put to sleep until the resource becomes unlocked. All the locks for a process are removed when the process terminates. [See *fcntl*(2) for more information about record locking.]

fildes is an open file descriptor. The file descriptor must have O_WRONLY or O_RDWR permission in order to establish lock with this function call.

function is a control value that 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 resource to be locked starts at the current offset in the file and extends forward for a positive size and backward for a negative size (the preceding bytes up to but not including the current offset). If *size* is zero, the section from the current offset through the largest file offset is locked (that is, from the current offset through the present or any future end-of-file). An area need not be allocated to the file in order to be locked because such locks can 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 will cause the calling process to sleep until the resource is available. F_TLOCK will cause the function to return a -1 and set *errno* to [EACCES] error if the section is already locked by another process.

F_ULOCK requests may, in whole or in part, release one or more locked sections controlled by the process. When sections are not fully released, the remaining sections are still locked by the process. Releasing the center section of a locked section requires an additional element in the table of active locks. If this table is full, an [EDEADLK] error is returned and the requested section is not released.

A potential for deadlock occurs if a process controlling a locked resource is put to sleep by accessing another process's locked resource. Thus calls to *lockf* or *fcntl* scan for a deadlock prior to sleeping on a locked resource. An error return is made if sleeping on the locked resource would cause a deadlock.

Sleeping on a resource is interrupted with any signal. The *alarm*(2) command can be used to provide a timeout facility in applications that require this facility.

The *lockf* utility will fail if one or more of the following are true:

[EBADF]	*fildes* is not a valid open descriptor.
[EACCES]	*cmd* is F_TLOCK or F_TEST and the section is already locked by another process.
[EDEADLK]	*cmd* is F_LOCK and a deadlock would occur.
[ENOLCK]	The *cmd* is F_LOCK, FT_LOCK, or F_ULOCK and the number of entries in the lock table would exceed the number allocated on the sytem. (Note that this differs from EDEADLOCK).
[ECOMM]	*fildes* is on a remote machine and the link to that machine is no longer active.

SEE ALSO

chmod(2), close(2), creat(2), fcntl(2), intro(2), open(2), read(2), write(2).

DIAGNOSTICS

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

WARNINGS

Unexpected results may occur in processes that do buffering in the user address space. The process may later read/write data that 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.

NAME

logname - return login name of user

SYNOPSIS

char *logname( )

DESCRIPTION

logname returns a pointer to the null-terminated login name; it extracts the **LOGNAME** environment variable from the user's environment.

This routine is kept in **/lib/libPW.a**.

FILES

/etc/profile
/usr/lib/libPW.a

SEE ALSO

env(1), login(1), getenv(3C), profile(4), environ(5).

CAVEATS

The return values point to static data whose content is overwritten by each call.

This method of determining a login name is subject to forgery.

NAME

lsearch, lfind - linear search and update

SYNOPSIS

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

char *lsearch ((char *)key, (char *)base, nelp, sizeof(*key), compar)
unsigned *nelp;
int (*compar)( );

char *lfind ((char *)key, (char *)base, nelp, sizeof(*key), compar)
unsigned *nelp;
int (*compar)( );
```

DESCRIPTION

lsearch is a linear search routine generalized from Knuth (6.1) Algorithm S. It returns a pointer into a table indicating where a datum may be found. If the datum does not occur, it is added at the end of the table. *key* points to the datum to be sought in the table. *base* points to the first element in the table. *nelp* points to an integer containing the current number of elements in the table. The integer is incremented if the datum is added to the table. *compar* is the name of the comparison function which the user must supply (*strcmp*, for example). It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal and non-zero otherwise.

lfind is the same as *lsearch* except that if the datum is not found, it is not added to the table. Instead, a NULL pointer is returned.

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.
The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.
Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

EXAMPLE

This fragment will read in less than TABSIZE strings of length less than ELSIZE and store them in a table, eliminating duplicates.

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

#define TABSIZE 50
```

```
#define ELSIZE 120

        char line[ELSIZE], tab[TABSIZE][ELSIZE], *lsearch( );
        unsigned nel = 0;
        int strcmp( );
        . . .
        while (fgets(line, ELSIZE, stdin) != NULL &&
                nel < TABSIZE)
                        (void) lsearch(line, (char *)tab, &nel,
                                ELSIZE, strcmp);
        . . .
```

SEE ALSO

bsearch(3C), hsearch(3C), string(3C), tsearch(3C).

DIAGNOSTICS

If the searched for datum is found, both *lsearch* and *lfind* return a pointer to it. Otherwise, *lfind* returns NULL and *lsearch* returns a pointer to the newly added element.

BUGS

Undefined results can occur if there is not enough room in the table to add a new item.

NAME

malloc, free, realloc, calloc - main memory allocator

SYNOPSIS

char *malloc (size)
unsigned size;

void free (ptr)
char *ptr;

char *realloc (ptr, size)
char *ptr;
unsigned size;

char *calloc (nelem, elsize)
unsigned nelem, elsize;

DESCRIPTION

malloc and *free* provide a simple general-purpose memory allocation package. *malloc* returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to *free* is a pointer to a block previously allocated by *malloc*; after *free* is performed this space is made available for further allocation, but its contents are left undisturbed.

Undefined results will occur if the space assigned by *malloc* is overrun or if some random number is handed to *free*.

malloc allocates the first big enough contiguous reach of free space found in a circular search from the last block allocated or freed, coalescing adjacent free blocks as it searches. It calls *sbrk* [see *brk*(2)] to get more memory from the system when there is no suitable space already free.

realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If no free block of *size* bytes is available in the storage arena, then *realloc* will ask *malloc* to enlarge the arena by *size* bytes and will then move the data to the new space.

realloc also works if *ptr* points to a block freed since the last call of *malloc*, *realloc*, or *calloc*; thus sequences of *free*, *malloc* and *realloc* can exploit the search strategy of *malloc* to do storage compaction.

calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

Each allocation routine returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

SEE ALSO

brk(2), malloc(3X).

DIAGNOSTICS

malloc, *realloc* and *calloc* return a NULL pointer if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. When this happens the block pointed to by *ptr* may be destroyed.

NOTES

Search time increases when many objects have been allocated; that is, if a program allocates but never frees, then each successive allocation takes longer. For an alternate, more flexible implementation, see *malloc*(3X).

NAME

malloc, free, realloc, calloc, mallopt, mallinfo - fast main memory allocator

SYNOPSIS

#include <malloc.h>

char *malloc (size)
unsigned size;

void free (ptr)
char *ptr;

char *realloc (ptr, size)
char *ptr;
unsigned size;

char *calloc (nelem, elsize)
unsigned nelem, elsize;

int mallopt (cmd, value)
int cmd, value;

struct mallinfo mallinfo()

DESCRIPTION

malloc and *free* provide a simple general-purpose memory allocation package, which runs considerably faster than the *malloc*(3C) package. It is found in the library "malloc", and is loaded if the option "-lmalloc" is used with *cc*(1) or *ld*(1).

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 have been destroyed (but see *mallopt* below for a way to change this behavior).

Undefined results will 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 will be 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 for control over the allocation algorithm. The available values for *cmd* are:

M_MXFAST	Set *maxfast* to *value*. The algorithm allocates all blocks below the size of *maxfast* in large groups and then doles them out very quickly. The default value for *maxfast* is 24.
M_NLBLKS	Set *numlblks* to *value*. The above mentioned "large groups" each contain *numlblks* blocks. *Numlblks* must be greater than 0. The default value for *numlblks* is 100.
M_GRAIN	Set *grain* to *value*. The sizes of all blocks smaller than *maxfast* are considered to be rounded up to the nearest multiple of *grain*. *Grain* must be greater than 0. The default value of *grain* is the smallest number of bytes which will allow alignment of any data type. Value will be rounded up to a multiple of the default when *grain* is set.
M_KEEP	Preserve data in a freed block until the next *malloc*, *realloc*, or *calloc*. This option is provided only for compatibility with the old version of *malloc* and is not recommended.

These values are defined in the **<malloc.h>** header file.

mallopt can be called repeatedly, but cannot be called after the first small block is allocated.

mallinfo provides instrumentation describing space usage. It returns the structure:

```
struct mallinfo {
   int arena;       /* total space in arena */
   int ordblks;     /* number of ordinary blocks */
   int smblks;      /* number of small blocks */
   int hblkhd;      /* space in holding block headers */
   int hblks;       /* number of holding blocks */
   int usmblks;     /* 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.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

SEE ALSO

brk(2), malloc(3C).

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 *malloc*(3C).

The code size is also bigger than *malloc*(3C).

Note that unlike *malloc*(3C), 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 *malloc*(3C) have not been duplicated.

NAME

matherr - error-handling function

SYNOPSIS

```
#include <math.h>

int matherr (x)
struct exception *x;
```

DESCRIPTION

matherr is invoked by functions in the Math Library when errors are detected. Users can define their own procedures for handling errors by including a function named *matherr* in their programs. *matherr* must be of the form described above. When an error occurs, a pointer to the exception structure *x* will be 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.
SING	Argument singularity.
OVERFLOW	Overflow range error.
UNDERFLOW	Underflow range error.
TLOSS	Total loss of significance.
PLOSS	Partial loss of significance.

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 will be returned by the function unless the user's *matherr* sets it to a different value.

If the user's *matherr* function returns non-zero, no error message will be printed, and *errno* will not be set.

If *matherr* is not supplied by the user, the default error-handling procedures, described with the math functions involved, will be invoked upon error. These procedures are also summarized in the table below. In every case, *errno* is set to 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 abort */
        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 (1); /* take no other action */
    }
    return (0); /* all other errors, execute default procedure */
}
```

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.

NAME

memory: memccpy, memchr, memcmp, memcpy, memset - memory operations

SYNOPSIS

```
#include <memory.h>

char *memccpy (s1, s2, c, n)
char *s1, *s2;
int c, n;

char *memchr (s, c, n)
char *s;
int c, n;

int memcmp (s1, s2, n)
char *s1, *s2;
int n;

char *memcpy (s1, s2, n)
char *s1, *s2;
int n;

char *memset (s, c, n)
char *s;
int c, n;
```

DESCRIPTION

These functions operate 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.

CAVEATS

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.

NAME

mktemp - make a unique file name

SYNOPSIS

char *mktemp (template)
char *template;

DESCRIPTION

The *mktemp* utility replaces the contents of the string pointed to by *template* by a unique file name and returns the address of *template*. The string in *template* should look like a file name with six trailing Xs; *mktemp* replaces the Xs with a letter and the current process ID. The letter is chosen so that the resulting name does not duplicate an existing file.

SEE ALSO

getpid(2), tmpfile(3S), tmpnam(3S).

DIAGNOSTIC

The *mktemp* utility assigns to *template* the NULL string if it cannot create a unique name.

CAVEAT

If called more than 26 times in a single process, *mktemp* starts recycling previously used names.

NAME

monitor - prepare execution profile

SYNOPSIS

#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.

monitor is an interface to *profil*(2). *lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a (user supplied) array of *bufsize* WORDs (defined in the <*mon.h*> header file). *monitor* arranges to record a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. The lowest address sampled is that of *lowpc* and the highest is just below *highpc*. *lowpc* may not equal 0 for this use of *monitor*. At most *nfunc* call counts can be kept; only calls of functions compiled with the profiling option **-p** of *cc*(1) are recorded.

For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use:

```
extern etext;
...
monitor ((int (*)())2, &etext, buf, bufsize, nfunc);
```

etext lies just above all the program text; see *end*(3C).

To stop execution monitoring and write the results, use

```
monitor ((int (*)())0, 0, 0, 0, 0);
```

prof(1) 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 *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

SEE ALSO

cc(1), prof(1), profil(2), end(3C).

BUGS

The ``*dirname/pid*.mon.out'' form does not work; the ``*dirname/pid.progname*'' form (automatically called via **cc -p**) does work.

NAME

dbm_open, dbm_close, dbm_fetch, dbm_store, dbm_delete, dbm_firstkey, dbm_nextkey, dbm_error, dbm_clearerr - database subroutines

SYNOPSIS

```
#include <ndbm.h>

typedef struct {
    char *dptr;
    int dsize;
} datum;

DBM *dbm_open(file, flags, mode)
    char *file;
    int flags, mode;

void dbm_close(db)
    DBM *db;

datum dbm_fetch(db, key)
    DBM *db;
    datum key;

int dbm_store(db, key, content, flags)
    DBM *db;
    datum key, content;
    int flags;

int dbm_delete(db, key)
    DBM *db;
    datum key;

datum dbm_firstkey(db)
    DBM *db;

datum dbm_nextkey(db)
    DBM *db;

int dbm_error(db)
    DBM *db;

int dbm_clearerr(db)
    DBM *db;
```

DESCRIPTION

These functions maintain key/content pairs in a database. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses. This package replaces, and is incompatible with, the earlier *dbm*(3x) library, which managed only a single database.

*key*s and *content*s 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 *dbm_open*. This opens and/or creates the files *file*.**dir** and *file*.**pag** depending on the flags parameter [see *open*(2)].

Once open, the data stored under a key is accessed by *dbm_fetch* and data is placed under a key by *dbm_store*. The *flags* field can be either **DBM_INSERT** or **DBM_REPLACE. DBM_INSERT** inserts only new entries into the database and does not change an existing entry with the same key. **DBM_REPLACE** replaces an existing entry if it has the same key. A key (and its associated contents) is deleted by *dbm_delete*. A linear pass through all keys in a database can be made, in an (apparently) random order, by use of *dbm_firstkey* and *dbm_nextkey*. *dbm_firstkey* returns the first key in the database. *dbm_nextkey* returns the next key in the database. This code traverses the database:

```
for key = dbm_firstkey(db); key.dptr != NULL; key = dbm_nextkey(db)
```

dbm_error returns non-zero when an error has occurred reading or writing the database. *dbm_clearerr* resets the error condition on the named database.

SEE ALSO

dbm(3X).

DIAGNOSTICS

All functions that return an *int* indicate errors with negative values. A zero return indicates no error condition. Routines that return a *datum* indicate errors with a null (0) *dptr*. If, when *dbm_store* is called with a *flags* value of DBM_INSERT, it finds an existing entry with the same key, it returns 1.

WARNINGS

The .pag file will contain holes so that its apparent size is about four times its actual content. Such a file 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 4096 bytes). Moreover all key/content pairs that hash together

NAME

dbm_open, dbm_close, dbm_fetch, dbm_store, dbm_delete, dbm_firstkey, dbm_nextkey, dbm_error, dbm_clearerr - database subroutines

SYNOPSIS

```
#include <ndbm.h>

typedef struct {
    char *dptr;
    int dsize;
} datum;

DBM *dbm_open(file, flags, mode)
    char *file;
    int flags, mode;

void dbm_close(db)
    DBM *db;

datum dbm_fetch(db, key)
    DBM *db;
    datum key;

int dbm_store(db, key, content, flags)
    DBM *db;
    datum key, content;
    int flags;

int dbm_delete(db, key)
    DBM *db;
    datum key;

datum dbm_firstkey(db)
    DBM *db;

datum dbm_nextkey(db)
    DBM *db;

int dbm_error(db)
    DBM *db;

int dbm_clearerr(db)
    DBM *db;
```

DESCRIPTION

These functions maintain key/content pairs in a database. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses. This package replaces, and is incompatible with, the earlier *dbm*(3x) library, which managed only a single database.

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 *dbm_open*. This opens and/or creates the files *file*.**dir** and *file*.**pag** depending on the flags parameter [see *open*(2)].

Once open, the data stored under a key is accessed by *dbm_fetch* and data is placed under a key by *dbm_store*. The *flags* field can be either **DBM_INSERT** or **DBM_REPLACE. DBM_INSERT** inserts only new entries into the database and does not change an existing entry with the same key. **DBM_REPLACE** replaces an existing entry if it has the same key. A key (and its associated contents) is deleted by *dbm_delete*. A linear pass through all keys in a database can be made, in an (apparently) random order, by use of *dbm_firstkey* and *dbm_nextkey*. *dbm_firstkey* returns the first key in the database. *dbm_nextkey* returns the next key in the database. This code traverses the database:

```
for key = dbm_firstkey(db); key.dptr != NULL; key = dbm_nextkey(db)
```

dbm_error returns non-zero when an error has occurred reading or writing the database. *dbm_clearerr* resets the error condition on the named database.

SEE ALSO

dbm(3X).

DIAGNOSTICS

All functions that return an *int* indicate errors with negative values. A zero return indicates no error condition. Routines that return a *datum* indicate errors with a null (0) *dptr*. If, when *dbm_store* is called with a *flags* value of DBM_INSERT, it finds an existing entry with the same key, it returns 1.

WARNINGS

The .pag file will contain holes so that its apparent size is about four times its actual content. Such a file 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 4096 bytes). Moreover all key/content pairs that hash together must fit on a single block. *dbm_store* will return an error in the event that a disk block fills with inseparable data.

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

The order of keys presented by *dbm_firstkey* and *dbm_nextkey* depends on a hashing function, not on anything interesting.

NAME

nlist - get entries from name list

SYNOPSIS

#include <nlist.h>

int nlist (filename, nl)
char *filename;
struct nlist *nl;

DESCRIPTION

nlist 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 will be 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*(4) 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.

SEE ALSO

a.out(4).

DIAGNOSTICS

All value entries are set to 0 if the file cannot be read or if it does not contain a valid name list.

nlist 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.

NAME

nlsgetcall - get client's data passed through the listener

SYNOPSIS

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

struct t_call *nlsgetcall(fd);
int fd;
```

DESCRIPTION

Nlsgetcall allows server processes started by the *listener* process to access the client's *t_call* structure [that is, the *sndcall* argument of *t_connect*(3N)].

The *t_call* structure returned by *nlsgetcall* can be released by using *t_free*(3N).

Nlsgetcall returns the address of an allocated *t_call* structure or NULL if a *t_call* structure cannot be allocated. If the *t_alloc* succeeds, undefined environment variables are indicated by a negative *len* field in the appropriate *netbuf* structure. A *len* field of zero in the *netbuf* structure is valid and means that the original buffer in the listener's *t_call* structure was NULL.

FILES

/usr/lib/libnsl_s.a
/usr/lib/libnsl.a

SEE ALSO

nlsadmin(1M), getenv(3), t_connect(3N), t_alloc(3N), t_free(3N), t_error(3N).

DIAGNOSTICS

A NULL pointer is returned if *t_alloc* cannot allocate a *t_call* structure. Further error information can be found in *t_errno*. Undefined environment variables are indicated by a negative length field (*len*) in the appropriate *netbuf* structure.

CAVEATS

The listener process limits the amount of user data (*udata*) and options data (*opt*) to 128 bytes each. Address data *addr* is limited to 64 bytes. If the original data was longer, no indication of overflow is given.

NOTES

Server processes must call *t_sync*(3N) before calling *nlsgetcall*(3N).

WARNING

The *len* field in the *netbuf* structure is defined as being unsigned. In order to check for error returns, it should first be cast to an integer.

NAME

nlsprovider - get name of transport provider

SYNOPSIS

char *nlsprovider( );

DESCRIPTION

Nlsprovider returns a pointer to a null terminated character string that contains the name of the transport provider as placed in the environment by the *listener* process. If the variable is not defined in the environment, a NULL pointer is returned.

The environment variable is available only to server processes started by the *listener* process.

SEE ALSO

nlsadmin(1M).

DIAGNOSTICS

If the variable is not defined in the environment, a NULL pointer is returned.

FILES

/usr/lib/libnsl.a
/usr/lib/libnsl_s.a

NAME

nlsrequest - format and send listener service request message

SYNOPSIS

```
#include <listen.h>

int nlsrequest(fd, service_code);
int fd;
char *service_code;

extern int _nlslog, t_errno;
extern char *_nlsrmsg;
```

DESCRIPTION

Given a virtual circuit to a listener process (*fd*) and a service code of a server process, *nlsrequest* formats and sends a *service request message* to the remote listener process, requesting that it start the given service. *Nlsrequest* waits for the remote listener process to return a *service request response message*, made available to the caller in the static, null-terminated data buffer pointed to by *_nlsrmsg*. The *service request response message* includes a success or failure code and a text message. The entire message can be printed.

FILES

/usr/lib/libnsl.a
/usr/lib/libnsl_s.a

SEE ALSO

nlsadmin(1M), t_error(3).

DIAGNOSTICS

The success or failure code is the integer return code from *nlsrequest*. Zero indicates success. Negative values such as the following indicate *nlsrequest* failures:

-1	Error encountered by nlsrequest, see t_errno.

Postive values are error return codes from the *listener* process. Mnemonics for these codes are defined in **listen.h**.

2	Request message not interpretable.
3	Request service code unknown.
4	Service code known, but currently disabled.

If non-null, *_nlsrmsg* contains a pointer to a static, null-terminated character buffer containing the service request response message. Note that both *_nlsrmsg* and the data buffer are overwritten by each call to *nlsrequest*.

If *_nlslog* is non-zero, *nlsrequest* prints error messages on stderr. Initially, *_nlslog* is zero.

WARNING

Nlsrequest does not always ensure that the remote server process has been successfully started. If the process has not been started, *nlsrequest* returns with no indication of an error, and the caller receives notification of a disconnect event through a **T_LOOK** error before or during the first *t_snd* or *t_rcv* call.

NAME

ocurse - optimized screen functions

SYNOPSIS

#include <ocurse.h>

DESCRIPTION

Ocurse is the old Berkeley *curses* library that uses *termcap*(4). New programs should use *curses*(3X).

These functions optimally update the screen.

Each *curses* program begins by calling *initscr* and ends by calling *endwin*.

Before a program can change a screen, it must specify the changes. It stores changes in a variable of type **WINDOW** by calling *curses* functions with the variable as argument. Once the variable contains all the changes desired, the program calls *wrefresh* to write the changes to the screen.

Most programs need only a single **WINDOW** variable. *Ocurse* provides a standard **WINDOW** variable for this case and a group of functions that operate on it. The variable is called *stdscr*; its special functions have the same names as the general functions minus the initial **w**.

FUNCTIONS

addch(ch)	Add a character to *stdscr*.
addstr(str)	Add a string to *stdscr*.
box(win,vert,hor)	Draw a box around a window.
crmode( )	Set cbreak mode.
clear( )	Clear *stdscr*.
clearok(scr,boolf)	Set clear flag for *scr*.
clrtobot( )	Clear to bottom on *stdscr*.
clrtoeol( )	Clear to end of line on *stdscr*.
delch( )	Delete a character.
deleteln( )	Delete a line.
delwin(win)	Delete *win*.
echo( )	Set echo mode.
endwin( )	End window modes.
erase( )	Erase *stdscr*.
getch( )	Get a char through *stdscr*.
getcap(name)	Get terminal capability *name*.
getstr(str)	Get a string through *stdscr*.
gettmode( )	Get tty modes.
getyx(win,y,x)	Get y and x coordinates.

inch( )	Get char at current y and x coordinates.
initscr( )	Initialize screens.
insch(c)	Insert a char.
insertln( )	Insert a line.
leaveok(win,boolf)	Set leave flag for *win*.
longname(termbuf,name)	Get long name from *termbuf*.
move(y,x)	Move to y and x coordinates on *stdscr*.
mvcur(lasty,lastx,newy,newx)	Actually move cursor.
newwin(lines,cols,begin_y,begin_x)	Create a new window.
nl( )	Set newline mapping.
nocrmode( )	Unset cbreak mode.
noecho( )	Unset echo mode.
nonl( )	Unset newline mapping.
noraw( )	Unset raw mode.
overlay(win1,win2)	Overlay win1 on win2.
overwrite(win1,win2)	Overwrite win1 on top of win2.
printw(fmt,arg1,arg2,. . .)	Printf on *stdscr*.
raw( )	Set raw mode.
refresh( )	Make current screen look like *stdscr*.
resetty( )	Reset tty flags to stored value.
savetty( )	Store current tty flags.
BR scanw (fmt,arg1,arg2,. . .)	Scanf through *stdscr*.
scroll(win)	Scroll *win* one line.
scrollok(win,boolf)	Set scroll flag.
setterm(name)	Set term variables for name.
standend( )	End standout mode.
standout( )	Start standout mode.
subwin(win,lines,cols,begin_y,begin_x)	Create a subwindow.
touchwin(win)	Change all of *win*.
unctrl(ch)	Printable version of *ch*.
waddch(win,ch)	Add char to *win*.
waddstr(win,str)	Add string to *win*.
wclear(win)	Clear *win*.
wclrtobot(win)	Clear to bottom of *win*.
wclrtoeol(win)	Clear to end of line on *win*.
wdelch(win,c)	Delete char from *win*.
wdeleteln(win)	Delete line from *win*.

werase(win)	Erase *win*.
wgetch(win)	Get a char through *win*.
wgetstr(win,str)	Get a string through *win*.
winch(win)	Get char at current y and x coordinates in *win*.
winsch(win,c)	Insert char into *win*.
winsertln(win)	Insert line into *win*.
wmove(win,y,x)	Set current y and x coordinates on *win*.
wprintw(win,fmt,arg1,arg2,. . .)	Printf on *win*.
wrefresh(win)	Make screen look like *win*.
wscanw(win,fmt,arg1,arg2,. . .)	Scanf through *win*.
wstandend(win)	End standout mode on *win*.
wstandout(win)	Start standout mode on *win*.

FILES

/usr/include/ocurse.h	header file
/usr/lib/libocurse.a	curses library
/usr/lib/libtermcap.a	termcap library, used by curses

SEE ALSO

stty(2), otermcap(3X), setenv(3), termcap(4).

NAME

tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs - terminal independent operations

SYNOPSIS

```
char PC;
char *BC;
char *UP;
short ospeed;

tgetent(bp, name)
char *bp, *name;

tgetnum(id)
char *id;

tgetflag(id)
char *id;

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

char *
tgoto(cmstr, destcol, destline)
char *cmstr;

tputs(cp, affcnt, outc)
register char *cp;
int affcnt;
int (*outc)();
```

DESCRIPTION

These functions extract and use information from terminal descriptions that follow the conventions in *termcap*(4). The functions perform only basic screen manipulation: they find and output specified terminal function strings and interpret the **cm** string. The *ocurse*(3X) routine describes a screen updating package built on *termcap*.

The *tgetent* routine finds and copies a terminal description. The *name* argument is the name of the description; *bp* points to a buffer to hold the description. The routine passes *bp* to the other *termcap* functions; the buffer must remain allocated until the program finishes with the *termcap* functions.

Tgetent uses the **TERM** and **TERMCAP** environment variables to locate the terminal description.

- If **TERMCAP** isn't set or is empty, *tgetent* searches for *name* in */etc/termcap*.
- If **TERMCAP** contains the full pathname of a file (any string that begins with /), *tgetent* searches for *name* in that file.
- If **TERMCAP** contains any string that does not begin with /, and **TERM** is not set or matches *name*, *tgetent* copies the **TERMCAP** string.
- If **TERMCAP** contains any string that does not begin with /, and **TERM** does not match *name*, *tgetent* searches for *name* in */etc/termcap*.

The *tgetent* routine returns -1 if it cannot open the terminal capability file; it returns 0 if it cannot find an entry for *name;* it returns 1 upon success.

The *tgetnum* routine returns the value of the numeric capability whose name is *id*; It returns -1 if the terminal lacks the specified capability or if it is not a numeric capability.

The *tgetflag* routine returns 1 if the terminal has boolean capability whose name is *id*; it returns 0 if it does not or it is not a boolean capability.

The *tgetstr* routine copies and interprets the value of the string capability named by *id*. It expands instances in the string of \ and ^. It leaves the expanded string in the buffer .ul indirectly pointed to by *area* and leaves the buffer's direct pointer pointing to the end of the expanded string. For example:

```
tgetstr("cl", &ptr);
```

where *ptr* is a character pointer, not an array name. The *tgetstr* routine returns a (direct) pointer to the beginning of the string.

The *tgoto* routine interprets the % escapes in a **cm** string. It returns *cmstr* with the % sequences changed to the position indicated by *destcol* and *destline*. This function must have the external variables *BC* and *UP* set to the values of the **bc** and **up** capabilities; if the terminal lacks the capability, set the external variable to NULL. If *tgoto* cannot interpret all the % sequences in **cm**, it returns OPS.

The *tgoto* routine avoids producing characters that might be misinterpreted by the terminal interface. If expanding a % sequence would produce a null or Control-D, the function tries to send the cursor to the next line or column and use *BC* or *UP* to move to the correct location. Note that *tgoto* does not avoid producing tabs; a program must disable the **TAB3** feature of the terminal interface [*termio*(7)]. This is a good idea anyway: some terminals use the tab character as a nondestructive space.

The *tputs* routine directs the output of a string returned by *tgetstr* or *tgoto*. This function must have the external variable *PC* set to the value of the **pc** capability; if the terminal lacks the capability, set the external variable to null. The *tputs* routine interprets any delay at the beginning of the string. The *cp* argument is the string to be output; *affcnt* is the number of lines affected by the action (1 if ''number of lines affected'' doesn't mean anything); and *outc* points to a function that takes a single **char** argument and outputs it, such as *putchar*.

FILES

/usr/lib/libotermcap.a	library
/etc/termcap	database

SEE ALSO

ex(1), ocurse(3X), termcap(4).

NAME

perror, errno, sys_errlist, sys_nerr - system error messages

SYNOPSIS

void perror (s)
char *s;

extern int errno;

extern char *sys_errlist[ ];

extern int sys_nerr;

DESCRIPTION

perror produces a message on the standard error output, describing the last error encountered during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, then the message and a newline. (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 newline. *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(2).

NAME

plot - graphics interface subroutines

SYNOPSIS

```
openpl ()

erase ()

label (s)
char *s;

line (x1, y1, x2, y2)
int x1, y1, x2, y2;

circle (x, y, r)
int x, y, r;

arc (x, y, x0, y0, x1, y1)
int x, y, x0, y0, x1, y1;

move (x, y)
int x, y;

cont (x, y)
int x, y;

point (x, y)
int x, y;

linemod (s)
char *s;

space (x0, y0, x1, y1)
int x0, y0, x1, y1;

closepl ()
```

DESCRIPTION

These subroutines generate graphic output in a relatively device-independent manner. The *space* routine must be used before any of these functions to declare the amount of space necessary [see *plot*(4)]. The *openpl* routine must be used before any of the others to open the device for writing. The *closepl* routine flushes the output.

The *circle* routine draws a circle of radius *r* with center at the point *(x, y)*.

The *arc* routine 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 newlines characters.

See *plot*(4) for a description of the effect of the remaining functions.

The library files listed below provide several flavors of these routines.

FILES

/usr/lib/libplot.a	produces output for *tplot*(1G) filters
/usr/lib/lib300.pa	for DASI 300
/usr/lib/lib300.a	for DASI 300s
/usr/lib/lib450.a	for DASI 450
/usr/lib/lib4014.a	for TEKTRONIX 4014
/usr/lib/libgt.a	for Convergent Technologies Graphics Terminal

SEE ALSO

graph(1G), stat(1G), tplot(1G), plot(4).

WARNINGS

In order 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''.

The above routines use **<stdio.h>**, which causes them to increase the size of programs, not otherwise using standard I/O more than might be expected.

NAME

popen, pclose - initiate pipe to/from a process

SYNOPSIS

#include <stdio.h>

FILE *popen (command, type)
char *command, *type;

int pclose (stream)
FILE *stream;

DESCRIPTION

The *popen* routine creates a pipe between the calling program and the command to be executed. The arguments to *popen* are pointers to null-terminated strings. The *command* argument 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 can be used as an input filter and a type **w** as an output filter.

EXAMPLE

A typical call can be the following:

```
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*(3S)] all the file names in the current directory that have a ".c" suffix.

SEE ALSO

pipe(2), wait(2), fclose(3S), fopen(3S), stdio(3S), system(3S).

DIAGNOSTICS

The *popen* routine returns a NULL pointer if files or processes cannot be created.

The *pclose* routine returns -1 if *stream* is not associated with a *popen*ed command.

WARNING

If the original and *popen*ed processes concurrently read or write a common file, neither should use buffered I/O because the buffering gets all mixed up. Problems with an output filter can be forestalled by careful buffer flushing; for example, with *fflush* [see *fclose*(3S)].

NAME

printf, fprintf, sprintf - print formatted output

SYNOPSIS

#include <stdio.h>

int printf (format , arg ...)
char *format;

int fprintf (stream, format , arg ...)
FILE *stream;
char *format;

int sprintf (s, format [, arg] ...)
char *s, *format;

DESCRIPTION

printf places output on the standard output stream *stdout*. *fprintf* places output on the named output *stream*. *sprintf* places "output," followed by the null character (**\0**), in consecutive bytes starting at **s*; it is the user's responsibility to ensure that enough storage is available. Each function returns the number of characters transmitted (not including the **\0** in the case of *sprintf*), or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its *arg*s under control of the *format*. The *format* is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching of zero or more *arg*s. The results are undefined if there are insufficient *arg*s for the format. If the format is exhausted while *arg*s remain, the excess *arg*s are simply ignored.

Each conversion specification is introduced by the character **%**. After the **%**, the following appear in sequence:

- Zero or more *flags*, which modify the meaning of the conversion specification.
- An optional decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag (**-**), 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 can be indicated by an asterisk (*) instead of a digit string. In this case, an integer *arg* supplies the field width or precision. The *arg* that is actually converted is not fetched until the conversion letter is seen, so the *arg*s specifying field width or precision must appear *before* the *arg* (if any) to be converted. A negative field width argument is taken as a (-) flag followed by a positive field width. If the precision argument is negative, it will be changed to zero.

The flag characters and their meanings are:

-	The result of the conversion will be left-justified within the field.
+	The result of a signed conversion will always begin with a sign (+ or -).
blank	If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and + flags both appear, the blank flag will be ignored.
#	This flag specifies that the value is to be converted to an alternate form. For **c**, **d**, **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 will have **0x or 0X** prefixed to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For **g** and **G** conversions, trailing zeroes will *not* be removed from the result (which they normally are).

The conversion characters and their meanings are:

d,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 will be expanded with leading zeroes. The default precision is 1. The result of converting a zero value with a precision of zero is a null string.

f	The float or double *arg* is converted to decimal notation in the style [-]ddd.ddd, where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, 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.ddde±dd, where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, six digits are produced; if the precision is zero, no decimal point appears. The **E** format code will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits.
g,G	The float or double *arg* is printed in style **f** or **e** (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style **e** will be used only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.
c	The character *arg* is printed.
s	The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (**\0**) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed. A NULL value for *arg* will yield 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 non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by *printf* and *fprintf* are printed as if *putc*(3S) had been called.

EXAMPLES

To print a date and time in the following 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(3C), putc(3S), scanf(3S), stdio(3S).

NAME

putc, putchar, fputc, putw - put character or word on a stream

SYNOPSIS

```
#include <stdio.h>

int putc (c, stream)
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 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.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), printf(3S), puts(3S), setbuf(3S), stdio(3S).

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 will occur if the file *stream* is not open for writing or if the output file cannot grow. Because **EOF** is a valid integer, *ferror*(3S) should be used to detect *putw* errors.

CAVEATS

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 cannot be read using *getw* on a different processor.

NAME

putenv - change or add value to environment

SYNOPSIS

int putenv (string)
char *string;

DESCRIPTION

string points to a string of the form *name=value*. *putenv* makes the value of the environment variable *name* equal to *value* by altering an existing variable or creating a new one. In either case, the string pointed to by *string* becomes part of the environment, so altering the string will change the environment. The space used by *string* is no longer used once a new string-defining *name* is passed to *putenv*.

SEE ALSO

exec(2), getenv(3C), malloc(3C), environ(5).

DIAGNOSTICS

putenv returns non-zero if it was unable to obtain enough space via *malloc* for an expanded environment; otherwise, it returns zero.

WARNINGS

putenv manipulates the environment pointed to by *environ*, and can be used in conjunction with *getenv*. However, *envp* (the third argument to *main*) is not changed.

This routine uses *malloc*(3C) to enlarge the environment.

After *putenv* is called, environmental variables are not in alphabetical order.
A potential error is to call *putenv* with an automatic variable as the argument, then exit from the calling function while *string* is still part of the environment.

NAME

putpwent - write password file entry

SYNOPSIS

#include <pwd.h>

int putpwent (p, f)
struct passwd *p;
FILE *f;

DESCRIPTION

The *putpwent* routine is the inverse of *getpwent*(3C). Given a pointer to a passwd structure created by *getpwent* (or *getpwuid* or *getpwnam*), *putpwent* writes a line on the stream *f*, which matches the format of **/etc/passwd**.

SEE ALSO

getpwent(3C), getspent(3X), putspent(3X).

DIAGNOSTICS

The *putpwent* routine returns non-zero if an error is detected during its operation; otherwise it returns zero.

WARNING

If a program not otherwise using standard I/O uses this routine, the size of the program increases more than might be expected.

NAME

puts, fputs - put a string on a stream

SYNOPSIS

```
#include <stdio.h>

int puts (s)
char *s;

int fputs (s, stream)
char *s;
FILE *stream;
```

DESCRIPTION

puts writes the null-terminated string pointed to by *s*, followed by a newline 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.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), printf(3S), putc(3S), stdio(3S).

DIAGNOSTICS

Both routines return EOF on error. This will happen if the routines try to write on a file that has not been opened for writing.

NOTES

puts appends a newline character while *fputs* does not.

NAME

putspent - write shadow password file entry

SYNOPSIS

```
#include <shadow.h>

int putspent (p, fp)
struct spwd *p;
FILE *fp;
```

DESCRIPTION

The *putspent* routine is the inverse of *getspent*(3X). 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.

SEE ALSO

getpwent(3C), putpwent(3C), getspent(3X).

DIAGNOSTICS

The *putspent* routine returns non-zero if an error was detected during its operation; otherwise, it returns zero.

WARNING

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 for internal use only; compatibility is not guaranteed.

NAME

qsort - quicker sort

SYNOPSIS

void qsort ((char *) base, nel, sizeof (*base), compar)
unsigned nel;
int (*compar)( );

DESCRIPTION

qsort is an implementation of the quicker-sort algorithm. It sorts a table of data in place.

base points to the element at the base of the table. *nel* is the number of elements in the table. *compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer giving the results of the comparison: a negative, 0, or positive return indicates to the sort that the first operand is less than, equal to, or greater than the second operand, respectively.

NOTES

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

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

The order in the output of two items which compare as equal is unpredictable.

SEE ALSO

sort(1), bsearch(3C), lsearch(3C), string(3C).

NAME

rand, srand - simple random-number generator

SYNOPSIS

int rand ( )

void srand (seed)
unsigned seed;

DESCRIPTION

The *rand* routine 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* routine 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*(3C) routine provides a much better, though more elaborate, random-number generator.

SEE ALSO

drand48(3C).

NAME

rcmd, rresvport, ruserok - routines for returning a stream to a remote command

SYNOPSIS

rcmd (ahost, inport, locuser, remuser, cmd, fd2p);
char **ahost;
unsigned short inport;
char *locuser, *remuser, *cmd;
int *fd2p;

rresvport (port);
int *port;

ruserok (rhost, superuser, ruser, luser);
char *rhost;
int superuser;
char *ruser, *luser;

DESCRIPTION

The *rcmd* routine is used by the super-user to execute a command on a remote machine using an authentication scheme based on reserved port numbers. The *rresvport* routine returns a descriptor to a socket with an address in the privileged port space. The *ruserok* routine is used by servers to authenticate clients requesting service with *rcmd*. All three functions are present in the same file and are used by the *rshd*(1M) server (among others).

The *rcmd* routine looks up the host **ahost* using *gethostbyname* (3), returning -1 if the host does not exist; otherwise, **ahost* is set to the standard name of the host and a connection is established to a server residing at the well-known Internet port *inport*.

If the call succeeds, a socket of type SOCK_STREAM is returned to the caller and given to the remote command as *stdin* and *stdout*. If *fd2p* is non-zero, an auxiliary channel to a control process is set up, and a descriptor for it is placed in **fd2p*. The control process returns diagnostic output from the command (unit 2) on this channel and accepts bytes on this channel as being CTIX signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, the *stderr* (unit 2 of the remote command) is made the same as the *stdout* and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

The protocol is described in *rshd*(1M).

The *rresvport* routine is used to obtain a socket with a privileged address bound to it. This socket is suitable for use by *rcmd* and several other routines. Privileged addresses consist of a port in the range 0 to 1023. Only the super-user can bind an address of this sort to a socket.

The *ruserok* routine takes a remote host's name, as returned by a *gethostbyname*(3) routine, two user names, and a flag indicating if the local user's name is the super-user. It then checks the files **/etc/hosts.equiv** and, possibly, **.rhosts** in the current working directory (normally the local user's home directory) to see if the request for service is allowed. A 1 is returned if the machine name is listed in the **hosts.equiv** file or if the host and remote user name are found in the **.rhosts** file; otherwise, *ruserok* returns 0. If the *superuser* flag is 1, the checking of the **host.equiv** file is bypassed.

SEE ALSO

rcmd(1), rexecd(1M), rlogin(1), rlogind(1M), rshd(1M), rexec(3).

BUGS

There is no way to specify options to the *socket* call that *rcmd* makes.

NAME

regcmp, regex - compile and execute regular expression

SYNOPSIS

char *regcmp (string1 [, string2, ...], (char *)0)
char *string1, *string2, ...;

char *regex (re, subject[, ret0, ...])
char *re, *subject, *ret0, ...;

extern char *__loc1;

DESCRIPTION

regcmp compiles a regular expression (consisting of the concatenated arguments) and returns a pointer to the compiled form. *malloc*(3C) is used to create 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*(1) has been written to generally preclude the need for this routine at execution time.

regex executes a compiled pattern against the subject string. Additional arguments are passed to receive values back. *regex* returns NULL on failure or a pointer to the next unmatched character on success. A global character pointer *__loc1* points to where the match began. *regcmp* and *regex* were mostly borrowed from the editor, *ed*(1); however, the syntax and semantics have been changed slightly. The following are the valid symbols and their associated meanings.

[ ] * . ^	These symbols retain their meaning in *ed*(1).
$	Matches the end of the string; \n matches a newline.
-	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 will be 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.

EXAMPLES

This example matches a leading newline in the subject string pointed at by cursor:

```
char *cursor, *newcursor, *ptr;
        ...
newcursor = regex((ptr = regcmp("^\n", (char *)0)), cursor);
free(ptr);
```

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*:

```
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 applies a precompiled regular expression in **file.i** [see *regcmp*(1)] against *string*:

```
char *string, *newcursor;
        ...
newcursor = regex(name, string);
```

These routines are kept in **/lib/libPW.a**.

SEE ALSO

ed(1), regcmp(1), malloc(3C), regexp(5).

BUGS

The user program may run out of memory if *regcmp* is called iteratively without freeing the vectors no longer required.

NAME

res_mkquery, res_send, res_init, dn_comp, dn_expand - resolver routines

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

res_mkquery(op, dname, class, type, data, datalen, newrr, buf, buflen)
int op;
char *dname;
int class, type;
char *data;
int datalen;
struct rrec *newrr;
char *buf;
int buflen;

res_send(msg, msglen, answer, anslen)
char *msg;
int msglen;
char *answer;
int anslen;

res_init()

dn_comp(exp_dn, comp_dn, length, dnptrs, lastdnptr)
char *exp_dn, *comp_dn;
int length;
char **dnptrs, **lastdnptr;

dn_expand(msg, eomorig, comp_dn, exp_dn, length)
char *msg, *eomorig, *comp_dn, exp_dn;
int length;
```

DESCRIPTION

These routines are used for making, sending and interpreting packets to Internet domain name servers. Global information that is used by the resolver routines is kept in the variable *_res*. Most of the values have reasonable defaults and can be ignored. Options stored in *_res.options* are defined in *resolv.h* and are as follows. Options are a simple bit mask and are ORed in to enable.

RES_INIT	True if the initial name server address and default domain name are initialized (that is, *res_init* has been called).

RES_DEBUG	Print debugging messages.
RES_AAONLY	Accept authoritative answers only. *res_send* will continue until it finds an authoritative answer or finds an error. Currently this is not implemented.
RES_USEVC	Use TCP connections for queries instead of UDP.
RES_STAYOPEN	Used with RES_USEVC to keep the TCP connection open between queries. This is useful only in programs that regularly do many queries. UDP should be the normal mode used.
RES_IGNTC	Unused currently (ignore truncation errors, that is, don't retry with TCP).
RES_RECURSE	Set the recursion desired bit in queries. This is the default. (*res_send* does not do iterative queries and expects the name server to handle recursion.)
RES_DEFNAMES	Append the default domain name to single label queries. This is the default.

res_init reads the initialization file, **/etc/resolv.conf**, to get the default domain name and the Internet address of the initial hosts running the name server. If the keyword *usefile* is present, no attempt is made to contact a name server, and the */etc/hosts* file is used. If the **nameserver** line does not exist, the host running the resolver is tried. (The current domain name is determined according to the following search scheme: first, *res_init* checks the value of the environment variable **LOCALDOMAIN**, and the value is used if this variable is set; if there is no value for **LOCALDOMAIN**, *res_init* checks the file **/etc/resolv.conf** for a domain specification [see *named*(1M) and *resolver*(4)], and the value is used if it can be obtained; if neither of the first two searches is successful, *res_init* uses the value specified in the start-up file **/etc/rcopts/INET-DOMAIN** if that file exists and contains a valid value.)

res_mkquery makes a standard query message and places it in *buf*. *res_mkquery* will return the size of the query or -1 if the query is larger than *buflen*. *op* is usually QUERY but can be any of the query types defined in *nameser.h*. *dname* is the domain name. If *dname* consists of a single label and the RES_DEFNAMES flag is enabled (the default), *dname* will be appended with the current domain name (determined in *res_init*). *newrr* is currently unused but is intended for making update messages.

res_send sends a query to name servers and returns an answer. It will call *res_init* if RES_INIT is not set, send the query to the local name server, and handle timeouts and retries. The length of the message is returned or -1 if there were errors.

dn_expand expands the compressed domain name *comp_dn* to a full domain name. Expanded names are converted to upper case. *msg* is a pointer to the beginning of the message, *exp_dn* is a pointer to a buffer of size *length* for the result. The size of compressed name is returned or -1 if there was an error.

dn_comp compresses the domain name *exp_dn* and stores it in *comp_dn*. The size of the compressed name is returned or -1 if there were errors. *length is the size of the comp_dn*. *dnptrs* is a list of pointers to previously compressed names in the current message. The first pointer points to to the beginning of the message and the list ends with NULL. *lastdnptr* is a pointer to the end of the array pointed to *dnptrs*. A side effect is to update the list of pointers for labels inserted into the message by *dn_comp* as the name is compressed. If *dnptr* is NULL, we don't try to compress names. If *lastdnptr* is NULL, we don't update the list.

FILES

/etc/resolv.conf

SEE ALSO

named(1M), resolver(4).
CTIX Network Administrator's Guide.
CTIX Network Programmer's Primer.

NAME

rexec - return stream to a remote command

SYNOPSIS

rexec (ahost, inport, user, passwd, cmd, fd2p);
char **ahost;
unsigned short inport;
char *user, *passwd, *cmd;
int *fd2p;

DESCRIPTION

rexec looks up the host **ahost* using *gethostbyname*(3), returning -1 if the host does not exist. Otherwise, **ahost* is set to the standard name of the host. If a user name and password are both specified, then these are used to authenticate to the foreign host; otherwise, the environment and then the user's **.netrc** file in his home directory are searched for appropriate information. If all this fails, the user is prompted for the information.

The port *inport* specifies which well-known DARPA Internet port to use for the connection; it will normally be the value returned from the call **getnameserv("exec", "tcp")** [see *getservent*(3)]. The protocol for connection is described in *rexecd*(1M).

If the call succeeds, a socket of type SOCK_STREAM is returned to the caller and given to the remote command as *stdin* and *stdout*. If *fd2p* is non-zero, then an auxiliary channel to a control process will be set up, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel and will also accept bytes on this channel as being CTIX signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the *stderr* (unit 2 of the remote command) will be made the same as the *stdout* and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

SEE ALSO

rexecd(1M), rcmd(3), gethostbyname(3).

BUGS

There is no way to specify options to the *socket* call which *rexec* makes.

NAME

scanf, fscanf, sscanf - convert formatted input

SYNOPSIS

#include <stdio.h>

int scanf (format [, pointer] ...)
char *format;

int fscanf (stream, format [, pointer] ...)
FILE *stream;
char *format;

int sscanf (s, format [, pointer] ...)
char *s, *format;

DESCRIPTION

scanf reads from the standard input stream *stdin*. *fscanf* reads from the named input *stream*. *sscanf* reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string *format* described below, and a set of *pointer* arguments indicating where the converted input should be stored. The 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 can contain:

1. White-space characters (blanks, tabs, newlines, 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 %), that 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 that 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 will store 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 space, 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 will be added automatically. The input field is terminated by a white-space character.

c A character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-space character, use **%1s**. If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read.

[Indicates string data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, 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 circumflex (^), when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters *not* contained in the remainder of the scanset string.

There are some conventions used in the construction of the scanset. A range of characters can be represented by the construct *first-last*; thus, [0123456789] can be expressed [0-9]. Using this convention, *first* must be lexically less than or equal to *last*, or else the dash will stand for itself. The dash will also stand for itself whenever it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset, and in this case it will not be syntactically interpreted as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating \0, which will be added automatically. At least one character must match for this conversion to be considered successful.

The conversion characters **d**, **u**, **o**, **x** and **i** can 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** can 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.

scanf conversion terminates at **EOF**, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

scanf returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, **EOF** is returned.

EXAMPLES

The call:

```
int n ; float x; char name[50];
n = scanf("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign to *n* the value 3, to *i* the value 25, to *x* the value 5.432, and *name* will contain **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
```

will assign **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*(3S)] will return **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
```

will assign **17** to *i*, **0** to *j*, **6** to *s*, will place the string **xy\0** in *name*, and will assign **8** to *e*. Thus, the length of *name* is *e* - *s* = 2 . The next call to *getchar* [see *getc*(3S)] will return a blank.

SEE ALSO

getc(3S), printf(3S), stdio(3S), strtod(3C), strtol(3C).

DIAGNOSTICS

These functions return **EOF** on end of input and a short count for missing or illegal data items.

CAVEATS

Trailing white space (including a newline) is left unread unless matched in the control string.

NAME

setbuf, setvbuf - assign buffering to a stream

SYNOPSIS

#include <stdio.h>

void setbuf (stream, buf)
FILE *stream;
char *buf;

int setvbuf (stream, buf, type, size)
FILE *stream;
char *buf;
int type, size;

DESCRIPTION

setbuf can 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 will be completely unbuffered.

A constant **BUFSIZ,** defined in the **<stdio.h>** header file, tells the size of the array needed:

```
char buf[BUFSIZ];
```

setvbuf can be used after a stream has been opened but before it is read or written. *Type* determines how *stream* will be buffered. Legal values for *type* (defined in stdio.h) are:

_IOFBF	Causes input/output to be fully buffered.
_IOLBF	Causes output to be line buffered; the buffer will be flushed when a newline 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 will be 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.

SEE ALSO

fopen(3S), getc(3S), malloc(3C), putc(3S), stdio(3S).

DIAGNOSTICS

If an illegal value for *type* or *size* is provided, *setvbuf* returns a non-zero value. Otherwise, the value returned will be zero.

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.

NAME

setjmp, longjmp - non-local goto

SYNOPSIS

#include <setjmp.h>

int setjmp (env)
jmp_buf env;

void longjmp (env, val)
jmp_buf env;
int val;

DESCRIPTION

These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

setjmp saves its stack environment in *env* (whose type, *jmp_buf*, is defined in the *<setjmp.h>* header file) for later use by *longjmp*. It returns the value 0.

longjmp restores the environment saved by the last call of *setjmp* with the corresponding *env* argument. After *longjmp* is completed, program execution continues as if the corresponding call of *setjmp* (which must not itself have returned in the interim) had just returned the value *val*. *longjmp* cannot cause *setjmp* to return the value 0. If *longjmp* is invoked with a second argument of 0, *setjmp* will return 1. 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 will be:

> value of *i* on 1st return from *setjmp*: 0
>
> value of *i* on 2nd return from *setjmp*: 1

SEE ALSO

signal(2).

WARNING

If *longjmp* is called even though *env* was never primed by a call to *setjmp*, or when the last such call was in a function that has since returned, absolute chaos is guaranteed.

NAME

sinh, cosh, tanh - hyperbolic functions

SYNOPSIS

#include <math.h>

double sinh (x)
double x;

double cosh (x)
double x;

double tanh (x)
double x;

DESCRIPTION

sinh, cosh, and *tanh* return, respectively, the hyberbolic sine, cosine and tangent of their argument.

SEE ALSO

matherr(3M).

DIAGNOSTICS

sinh and *cosh* return **HUGE** (and *sinh* may return **-HUGE** for negative *x*) when the correct value would overflow and set *errno* to **ERANGE.**

These error-handling procedures may be changed with the function *matherr*(3M).

NAME

sleep - suspend execution for interval

SYNOPSIS

unsigned sleep (seconds)
unsigned seconds;

DESCRIPTION

The current process is suspended from execution for the number of *seconds* specified by the argument. The actual suspension time may be less than that requested for two reasons: (1) Because scheduled wakeups occur at fixed one-second intervals, (on the second, according to an internal clock) and (2) because any caught signal will terminate the *sleep* following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount due to the scheduling of other activity in the system. The value returned by *sleep* will be the ''unslept'' amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested *sleep* time, or premature arousal due to another caught signal.

The routine is implemented by setting an alarm signal and pausing until it (or some other signal) occurs. The previous state of the alarm signal is saved and restored. The calling program may have set up an alarm signal before calling *sleep*. If the *sleep* time exceeds the time till such alarm signal, the process sleeps only until the alarm signal would have occurred. The caller's alarm catch routine is executed just before the *sleep* routine returns. But if the *sleep* time is less than the time till such alarm, the prior alarm time is reset to go off at the same time it would have without the intervening *sleep*.

SEE ALSO

alarm(2), pause(2), signal(2).

NAME

sputl, sgetl - access long integer data in a machine-independent fashion

SYNOPSIS

void sputl (value, buffer)
long value;
char *buffer;

long sgetl (buffer)
char *buffer;

DESCRIPTION

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.

A program that uses these functions must be loaded with the object-file access routine library **libld.a**.

NAME

ssignal, gsignal - software signals

SYNOPSIS

#include <signal.h>

int (*ssignal (sig, action))( )
int sig, (*action)( );

int gsignal (sig)
int sig;

DESCRIPTION

ssignal and *gsignal* implement a software facility similar to *signal*(2). This facility is used by the Standard C Library to enable users to indicate the disposition of error conditions, and is also made available to users for their own purposes.

Software signals made available to users are associated with integers in the inclusive range 1 through 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). *ssignal* returns the action previously established for that signal type; if no action has been established or the signal number is illegal, *ssignal* returns SIG_DFL.

gsignal raises the signal identified by its argument, *sig*:

- If an action function has been established for *sig*, then that action is reset to SIG_DFL and the action function is entered with argument *sig*. *gsignal* returns the value returned to it by the action function.
- If the action for *sig* is SIG_IGN, *gsignal* returns the value 1 and takes no other action.
- If the action for *sig* is SIG_DFL, *gsignal* returns the value 0 and takes no other action.
- If *sig* has an illegal value or no action was ever specified for *sig*, *gsignal* returns the value 0 and takes no other action.

SEE ALSO

signal(2), sigset(2).

NOTES

There are some additional signals with numbers outside the range 1 through 16 which are used by the Standard C Library to indicate error conditions. Thus, some signal numbers outside the range 1 through 16 are legal, although their use may interfere with the operation of the Standard C Library.

NAME

stdio - standard buffered input/output package

SYNOPSIS

#include <stdio.h>

FILE *stdin, *stdout, *stderr;

DESCRIPTION

The functions described in the entries of sub-class 3S of this manual constitute an efficient, user-level I/O buffering scheme. The in-line macros *getc*(3S) and *putc*(3S) handle characters quickly. The macros *getchar* and *putchar*, and the higher-level routines *fgetc*, *fgets*, *fprintf*, *fputc*, *fputs*, *fread*, *fscanf*, *fwrite*, *gets*, *getw*, *printf*, *puts*, *putw*, and *scanf* all use or act as if they use *getc* and *putc*; they can be intermixed freely.

A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type FILE. *fopen*(3S) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the <stdio.h> header file and associated with the standard open files:

stdin	standard input file
stdout	standard output file
stderr	standard error file

A constant NULL (0) designates a nonexistent pointer.

An integer-constant **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 functions and constants mentioned in the entries of sub-class 3S of this manual are declared in that header file and need no further declaration. The constants and the following functions are implemented as macros (redeclaration of these names is perilous): *getc*, *getchar*, *putc*, *putchar*, *ferror*, *feof*, *clearerr*, and *fileno*.

Output streams, with the exception of the standard error stream *stderr*, are by default buffered if the output refers to a file and line-buffered if the output refers to a terminal. The standard error output stream *stderr* is by default unbuffered, but use of *freopen* [see *fopen*(3S)] will cause it to become buffered or line-buffered. When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as written; when it is buffered, many characters are saved up and written as a block. When it is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a newline character is written or terminal input is requested). *setbuf*(3S) or *setvbuf*() in *setbuf*(3S) can be used to change the stream's buffering strategy.

SEE ALSO

close(2), lseek(2), open(2), pipe(2), read(2), write(2), ctermid(3S), cuserid(3S), fclose(3S), ferror(3S), fopen(3S), fread(3S), fseek(3S), getc(3S), gets(3S), popen(3S), printf(3S), putc(3S), puts(3S), scanf(3S), setbuf(3S), system(3S), tmpfile(3S), tmpnam(3S), ungetc(3S).

DIAGNOSTICS

Invalid *stream* pointers will usually cause grave disorder, possibly including program termination. Individual function descriptions describe the possible error conditions.

NAME

stdipc, ftok - standard interprocess communication package

SYNOPSIS

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

key_t ftok(path, id)
char *path;
char id;

DESCRIPTION

All interprocess communication facilities require the user to supply a key to be used by the *msgget*(2), *semget*(2), and *shmget*(2) system calls to obtain interprocess communication identifiers. One suggested method for forming a key is to use the *ftok* subroutine described below. Another way to compose keys is to include the project ID in the most significant byte and to use the remaining portion as a sequence number. There are many other ways to form keys, but each system must define standards for forming them. If some standard is not adhered to, unrelated processes can unintentionally interfere with another's operation. Therefore, it is recommended that the most significant byte of a key in some sense refer to a project so that keys do not conflict across a given system.

ftok returns a key based on *path* and *id* that is usable in subsequent *msgget*, *semget*, and *shmget* system calls. *Path* must be the path name of an existing file that is accessible to the process. *Id* is a character which uniquely identifies a project. Note that *ftok* will return the same key for linked files when called with the same *id* and that it will return different keys when called with the same file name but different *ids*.

SEE ALSO

intro(2), msgget(2), semget(2), shmget(2).

DIAGNOSTICS

ftok returns **(key_t) -1** if *path* does not exist or if it is not accessible to the process.

WARNING

If the file whose *path* is passed to *ftok* is removed when keys still refer to the file, future calls to *ftok* with the same *path* and *id* will return an error. If the same file is recreated, then *ftok* is likely to return a different key than it did when it was called originally.

NAME

string: strcat, strdup, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok - string operations

SYNOPSIS

```
#include <string.h>
#include <sys/types.h>

char *strcat (s1, s2)
char *s1, *s2;

char *strdup (s1)
char *s1;

char *strncat (s1, s2, n)
char *s1, *s2;
size_t n;

int strcmp (s1, s2)
char *s1, *s2;

int strncmp (s1, s2, n)
char *s1, *s2;
size_t n;

char *strcpy (s1, s2)
char *s1, *s2;

char *strncpy (s1, s2, n)
char *s1, *s2;
size_t n;

int strlen (s)
char *s;

char *strchr (s, c)
char *s;
int c;

char *strrchr (s, c)
char *s;
int c;

char *strpbrk (s1, s2)
char *s1, *s2;

int strspn (s1, s2)
char *s1, *s2;
```

int strcspn (s1, s2)
char *s1, *s2;

char *strtok (s1, s2)
char *s1, *s2;

DESCRIPTION

The arguments **s1, s2** and **s** point to strings (arrays of characters terminated by a null character). The functions *strcat*, *strncat*, *strcpy*, and *strncpy* all alter **s1.** These functions do not check for overflow of the array pointed to by **s1.**

The *strcat* routine appends a copy of string **s2** to the end of string **s1.**

The *strdup* routine returns a pointer to a new string which is a duplicate of the string pointed to by **s1.** The space for the new string is obtained using *malloc*(3C). If the new string can not be created, null is returned.

The *strncat* routine appends at most **n** characters. Each returns a pointer to the null-terminated result.

The *strcmp* routine 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.** The *strncmp* routine makes the same comparison but looks at at most **n** characters.

The *strcpy* routine copies string **s2** to **s1,** stopping after the null character has been copied. The *strncpy* routine 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.**

The *strlen* routine returns the number of characters in **s,** not including the terminating null character.

The *strchr* (*strrchr*) routine returns a pointer to the first (last) occurrence of character **c** in string **s,** or a NULL pointer if **c** does not occur in the string. The null character terminating a string is considered to be part of the string.

The *strpbrk* routine 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.**

The *strspn* (*strcspn*) routine returns the length of the initial segment of string **s1** which consists entirely of characters from (not from) string **s2.**

The *strtok* routine considers the string **s1** to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string **s2.** The first call (with pointer **s1** specified) returns a pointer to the first character of the first token, and will have written a null character into **s1** immediately following the returned token. The function keeps track of its position in the string between separate calls, so that subsequent calls (which must be made with the first argument a NULL pointer) work through the string **s1** immediately following that token. 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.

SEE ALSO

malloc(3C), malloc(3X).

CAVEATS

The *strcmp* and *strncmp* routines use native character comparison. Characters are 8-bit signed values; all ASCII characters have values of at least 0; non-ASCII are negative. On some machines, all characters are positive. Thus, programs that only compare ASCII values are portable; programs that compare ASCII with non-ASCII values are not.

Character movement is performed differently in different implementations. Thus, overlapping moves may yield surprises.

NAME

strtod, atof - convert string to double-precision number

SYNOPSIS

double strtod (str, ptr)
char *str, **ptr;

double atof (str)
char *str;

DESCRIPTION

The *strtod* routine 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* routine recognizes an optional string of "white-space" characters [as defined by *isspace* in *ctype*(3C)], then an optional sign, then a string of digits optionally containing a decimal point, then an optional **e** or **E** followed by an optional sign or space, followed by an integer.

If the value of *ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no number can be formed, **ptr* is set to *str*, and zero is returned.

The *atof(str)* variable is equivalent to *strtod(str, (char **)NULL)*.

SEE ALSO

ctype(3C), scanf(3S), strtol(3C).

DIAGNOSTICS

If the correct value would cause overflow, ±HUGE (as defined in **<math.h>**) is returned (according to the sign of the value), and *errno* is set to **ERANGE.**

If the correct value would cause underflow, zero is returned and *errno* is set to **ERANGE.**

NAME

strtol, atol, atoi - convert string to integer

SYNOPSIS

```
long strtol (str, ptr, base)
char *str, **ptr;
int base;

long atol (str)
char *str;

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

DESCRIPTION

The *strtol* routine 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* (3C)] 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, of course, take place upon assignment or by an explicit cast.

The *atol(str)* variable is equivalent to *strtol(str, (char **)NULL, 10)*.

The *atoi(str)* variable is equivalent to *(int) strtol(str, (char **)NULL, 10)*.

SEE ALSO

ctype(3C), scanf(3S), strtod(3C).

CAVEAT

Overflow conditions are ignored.

NAME

swab - swap bytes

SYNOPSIS

```
void swab (from, to, nbytes)
char *from, *to;
int nbytes;
```

DESCRIPTION

The *swab* routine copies *nbytes* bytes pointed to by *from* to the array pointed to by *to*, exchanging adjacent even and odd bytes. The *nbytes* argument should be even and non-negative. If *nbytes* is odd and positive *swab* uses *nbytes*-1 instead. If *nbytes* is negative, *swab* does nothing.

NAME

system - issue a shell command

SYNOPSIS

#include <stdio.h>

int system (string)
char *string;

DESCRIPTION

The *system* routine causes the *string* to be given to *sh*(1) as input, as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status of the shell.

FILES

/bin/sh

SEE ALSO

sh(1), exec(2).

DIAGNOSTICS

system forks to create a child process that in turn exec's /bin/sh in order to execute *string*. If the fork or exec fails, *system* returns a negative value and sets *errno*.

NAME

t_accept - accept a connect request

SYNOPSIS

```
#include <tiuser.h>

int t_accept(fd, resfd, call)
int fd;
int 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. The *t_call* structure pointed to by *call* contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

The *netbuf* structure is described in *intro*(3). 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 can 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 (*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 TBADF.

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*(3N)] before the *t_accept* is issued.

For both types of endpoints, *t_accept* fails and sets *t_errno* to TLOOK if there are indications (such as 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* [see *netbuf* in *intro*(3)] field of *udata* is zero, no data is sent to the caller.

On failure, *t_errno* can be set to one of the following:

[TBADF]	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.
[TOUTSTATE]	The function was issued in the wrong sequence on the transport endpoint referenced by *fd*, or the transport endpoint referred to by *resfd* is not in the T_IDLE state.
[TACCES]	The user does not have permission to accept a connection on the responding transport endpoint or use the specified options.
[TBADOPT]	The specified options were 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.
[TBADSEQ]	An invalid sequence number was specified.
[TLOOK]	An asynchronous event has occurred on the transport endpoint referenced by *fd* and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

intro(3), t_connect(3N), t_getstate(3N), t_listen(3N), t_open(3N), t_rcvconnect(3N).
CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.

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.

NAME

t_alloc - allocate a library structure

SYNOPSIS

```
#include <tiuser.h>

char *t_alloc(fd, struct_type, fields)
int fd;
int struct_type;
int fields;
```

DESCRIPTION

The *t_alloc* function dynamically allocates memory for the various transport function argument structures as specified below. This function will allocate memory for the specified structure, and will also allocate memory for buffers referenced by the structure.

The structure to allocate is specified by *struct_type* and can be one of the following:

T_BIND	struct t_bind
T_CALL	struct t_call
T_OPTMGMT	struct t_optmgmt
T_DIS	struct t_discon
T_UNITDATA	struct t_unitdata
T_UDERROR	struct t_uderr
T_INFO	struct t_info

where each structure can be used subsequently as an argument to one or more transport functions.

Each of the above structures except T_INFO contains at least one field of type *struct netbuf*. *netbuf* is described in *intro*(3). For each field of this type, the user can 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_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.
T_ALL	All relevant fields of the given structure.

For each field specified in *fields*, *t_alloc* will allocate memory for the buffer associated with the field and initialize the *buf* pointer and *maxlen* [see *netbuf* in *intro*(3) for description of *buf* and *maxlen*] field accordingly. The length of the buffer allocated will be 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 will be 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* will be unable to determine the size of the buffer to allocate and will fail, setting *t_errno* to TSYSERR and *errno* to EINVAL. For any field not specified in *fields*, *buf* will be set to NULL and *maxlen* will be set to zero.

Use of *t_alloc* to allocate structures will help 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.
[TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

intro(3), t_free(3N), t_getinfo(3N), t_open(3N).
CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.

DIAGNOSTICS

On successful completion, *t_alloc* returns a pointer to the newly allocated structure. On failure, NULL is returned.

NAME

t_bind - bind an address to a transport endpoint

SYNOPSIS

```
#include <tiuser.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 *intro*(3). The *addr* field of the *t_bind* structure specifies a protocol address and the *qlen* field indicates the maximum number of outstanding connect indications.

req requests that an address, represented by the *netbuf* structure, be bound to the given transport endpoint. *len* [see *netbuf* in *intro*(3); 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 will result.

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 will assign an appropriate address to be bound and will return 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 want 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 who expects other users to call it. The value of *qlen* will be negotiated by the transport provider and can be changed if the transport provider cannot support the specified number of outstanding connect indications. On return, the *qlen* field in *ret* will contain 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 will assign 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 protocol address will be 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 will prevent more than one transport endpoint bound to the same protocol address from accepting connect indications.

On failure, *t_errno* may be set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TOUTSTATE]	The function was issued in the wrong sequence.

[TBADADDR]	The specified protocol address was in an incorrect format or contained illegal information.
[TNOADDR]	The transport provider could not allocate an address.
[TACCES]	The user does not have permission to use the specified address.
[TBUFOVFLW]	The number of bytes allowed for an incoming argument is not sufficient to store the value of that argument. The provider's state will change to T_IDLE and the information to be returned in *ret* will be discarded.
[TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

intro(3), t_open(3N), t_optmgmt(3N), t_unbind(3N).
CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.

DIAGNOSTICS

t_bind returns 0 on success and -1 on failure and *t_errno* is set to indicate the error.

NAME

t_close - close a transport endpoint

SYNOPSIS

#include <tiuser.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.

t_close should be called from the T_UNBND state [see *t_getstate* (3N)]. 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*(2) 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.

SEE ALSO

t_getstate(3N), t_open(3N), t_unbind(3N).
UNIX System V Release 3.2 Network Programmer's Guide.

DIAGNOSTICS

t_close returns 0 on success and -1 on failure and *t_errno* is set to indicate the error.

NAME

t_connect - establish a connection with another transport user

SYNOPSIS

#include <tiuser.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 *intro*(3). 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 can 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* (3N) or *t_getinfo* (3N). If the *len* [see *netbuf* in *intro*(3)] 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 *intro*(3)] 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, return value of zero) indicates that the requested connection has been established. However, if O_NDELAY 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 it 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:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TOUTSTATE]	The function was issued in the wrong sequence.
[TNODATA]	O_NDELAY was set, so the function successfully initiated the connection establishment procedure but did not wait for a response from the remote user.
[TBADADDR]	The specified protocol address was in an incorrect format or contained illegal information.
[TBADOPT]	The specified protocol options were 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.
[TACCES]	The user does not have permission to use the specified address or options.

[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 *rcvcall* is discarded.
[TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

intro(3), t_accept(3N), t_getinfo(3N), t_listen(3N), t_open(3N), t_optmgmt(3N), t_rcvconnect(3N).
CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.

DIAGNOSTICS

t_connect returns 0 on success and -1 on failure and *t_errno* is set to indicate the error.

NAME

t_error - produce error message

SYNOPSIS

```
#include <tiuser.h>

void t_error(errmsg)
char *errmsg;
extern int t_errno;
extern char *t_errlist[];
extern int t_nerr;
```

DESCRIPTION

The *t_error* routine 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. The *t_error* routine 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* also prints the standard error message for the current value contained in *errno* [see *intro*(2)]. The *t_errlist* array of message strings allows 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.

SEE ALSO

CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.

NAME

t_free - free a library structure

SYNOPSIS

#include <tiuser.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 will free memory for the specified structure, and will also free 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:

T_BIND	struct t_bind
T_CALL	struct t_call
T_OPTMGMT	struct t_optmgmt
T_DIS	struct t_discon
T_UNITDATA	struct t_unitdata
T_UDERROR	struct t_uderr
T_INFO	struct t_info

where each of these structures is an argument to one or more transport functions.

t_free will check the *addr*, *opt*, and *udata* fields of the given structure (as appropriate) and free the buffers pointed to by the *buf* field of the *netbuf* structure [see *intro*(3)]. *buf* is NULL, *t_free* will not attempt to free memory. After all buffers are freed, *t_free* will free the memory associated with the structure pointed to by *ptr*.

Undefined results will 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:

[TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

intro(3), t_alloc(3N).
CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.

DIAGNOSTICS

t_free returns 0 on success and -1 on failure and *t_errno* is set to indicate the error.

NAME

t_getinfo - get protocol-specific service information

SYNOPSIS

```
#include <tiuser.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 returns 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 a *t_info* structure which contains the following members:

```
long addr;        /* max size of the transport protocol address */
long options;     /* max number of bytes of protocol-specific options */
long tsdu;        /* max size of a transport service data unit (TSDU) */
long etsdu;       /* max size of an expedited transport service data unit (ETSDU) */
long connect;     /* max amount of data allowed on connection establishment */
                  /* functions */
long discon;      /* max amount of data allowed on t_snddis and t_rcvdis */
                  /* functions */
long servtype;    /* service type supported by the transport provider */
```

The values of the fields have the following meanings:

addr	A value greater than or equal to zero indicates the maximum size of a transport protocol address; a value of -1 specifies that there is no limit on the address size; and a value of -2 specifies that the transport provider does not provide user access to transport protocol addresses.
options	A value greater than or equal to zero indicates the maximum number of bytes of protocol-specific options supported by the provider; a value of -1 specifies that there is no limit on the option size; and a value of -2 specifies that the transport provider does not support user-settable options.
tsdu	A value greater than zero specifies the maximum size of a transport service data unit (TSDU); a value of zero specifies that the transport provider does not support the concept of TSDU, although it does

support the sending of a data stream with no logical boundaries preserved across a connection; a value of -1 specifies that there is no limit on the size of a TSDU; and a value of -2 specifies that the transfer of normal data is not supported by the transport provider.

etsdu	A value greater than zero specifies the maximum size of an expedited transport service data unit (ETSDU); a value of zero specifies that the transport provider does not support the concept of ETSDU, although it does support the sending of an expedited data stream with no logical boundaries preserved across a connection; a value of -1 specifies that there is no limit on the size of an ETSDU; and a value of -2 specifies that the transfer of expedited data is not supported by the transport provider.
connect	A value greater than or equal to zero specifies the maximum amount of data that can be associated with connection establishment functions; a value of -1 specifies that there is no limit on the amount of data sent during connection establishment; and a value of -2 specifies that the transport provider does not allow data to be sent with connection establishment functions.
discon	A value greater than or equal to zero specifies the maximum amount of data that can be associated with the *t_snddis* and *t_rcvdis* functions; a value of -1 specifies that there is no limit on the amount of data sent with these abortive release functions; and a value of -2 specifies that the transport provider does not allow data to be sent with the abortive release functions.
servtype	This field specifies the service type supported by the transport provider, as described below.

If a transport user is concerned with protocol independence, the above sizes can be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the *t_alloc* function can be used to allocate these buffers. An error will result if a transport user exceeds the allowed data size on any function. The value of each field may change as a result of option negotiation, and *t_getinfo* enables a user to retrieve the current characteristics.

The *servtype* field of *info* may specify one of the following values on return:

T_COTS	The transport provider supports a connection-mode service but does not support the optional orderly release facility.
T_COTS_ORD	The transport provider supports a connection-mode service with the optional orderly release facility.

T_CLTS	The transport provider supports a connectionless-mode service. For this service type, *t_open* will return -2 for *etsdu*, *connect*, and *discon*.

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.

SEE ALSO

t_open(3N).
CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.

DIAGNOSTICS

t_getinfo returns 0 on success and -1 on failure and *t_errno* is set to indicate the error.

NAME

t_getstate - get the current state

SYNOPSIS

#include <tiuser.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* can be set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TSTATECHNG]	The transport provider is undergoing a state change.
[TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

t_open(3N).
CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.

DIAGNOSTICS

t_getstate returns the current state on successful completion, -1 on failure; *t_errno* is set to indicate the error. The current state can be one of the following:

T_UNBND	unbound
T_IDLE	idle
T_OUTCON	outgoing connection pending
T_INCON	incoming connection pending
T_DATAXFER	data transfer
T_OUTREL	outgoing orderly release (waiting for an orderly release indication)
T_INREL	incoming orderly release (waiting for an orderly release request)

If the provider is undergoing a state transition when *t_getstate* is called, the function will fail.

NAME

t_listen - listen for a connect request

SYNOPSIS

```
#include <tiuser.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 *intro*(3). 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 *intro*(3)] 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_NDELAY 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* can 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. The provider's state, as seen by the user,

	changes to T_INCON, and the connect indication information to be returned in *call* is discarded.
[TNODATA]	O_NDELAY was set, but no connect indications had been queued.
[TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

CAVEATS

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 will wait forever because no connect indications will arrive on that endpoint.

SEE ALSO

intro(3), t_accept(3N), t_bind(3N), t_connect(3N), t_open(3N), t_rcvconnect(3N).
CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.

DIAGNOSTICS

t_listen returns 0 on success and -1 on failure and *t_errno* is set to indicate the error.

NAME

t_look - look at the current event on a transport endpoint

SYNOPSIS

#include <tiuser.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.

On failure, *t_errno* can 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.

SEE ALSO

t_open(3N).
CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.

DIAGNOSTICS

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:

T_LISTEN	connection indication received
T_CONNECT	connect confirmation received
T_DATA	normal data received
T_EXDATA	expedited data received
T_DISCONNECT	disconnect received
T_ERROR	fatal error indication
T_UDERR	datagram error indication

T_ORDREL	orderly release indication

On failure, -1 is returned and *t_errno* is set to indicate the error.

NAME

t_open - establish a transport endpoint

SYNOPSIS

#include <tiuser.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 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 path name of the file to open, and *oflag* identifies any open flags [as in *open*(2)]. *t_open* returns a file descriptor that will be 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 a *t_info* which contains the following members:

```
long addr;       /* max size of the transport protocol address */
long options;    /* max number of bytes of protocol-specific options */
long tsdu;       /* max size of a transport service data unit (TSDU) */
long etsdu;      /* max size of an expedited transport service data unit */
                 /* (ETSDU) */
long connect;    /* max amount of data allowed on connection */
                 /* establishment functions */
long discon;     /* max amount of data allowed on t_snddis and t_rcvdis */
                 */ functions */
long servtype;   /* service type supported by the transport provider */
```

The values of the fields have the following meanings:

addr A value greater than or equal to zero indicates the maximum size of a transport protocol address; a value of -1 specifies that there is no limit on the address size; and a value of -2 specifies that the transport provider does not provide user access to transport protocol addresses.

options	A value greater than or equal to zero indicates the maximum number of bytes of protocol-specific options supported by the provider; a value of -1 specifies that there is no limit on the option size; and a value of -2 specifies that the transport provider does not support user-settable options.
tsdu	A value greater than zero specifies the maximum size of a transport service data unit (TSDU); a value of zero specifies that the transport provider does not support the concept of TSDU, although it does support the sending of a data stream with no logical boundaries preserved across a connection; a value of -1 specifies that there is no limit on the size of a TSDU; and a value of -2 specifies that the transfer of normal data is not supported by the transport provider.
etsdu	A value greater than zero specifies the maximum size of an expedited transport service data unit (ETSDU); a value of zero specifies that the transport provider does not support the concept of ETSDU, although it does support the sending of an expedited data stream with no logical boundaries preserved across a connection; a value of -1 specifies that there is no limit on the size of an ETSDU; and a value of -2 specifies that the transfer of expedited data is not supported by the transport provider.
connect	A value greater than or equal to zero specifies the maximum amount of data that can be associated with connection establishment functions; a value of -1 specifies that there is no limit on the amount of data sent during connection establishment; and a value of -2 specifies that the transport provider does not allow data to be sent with connection establishment functions.
discon	A value greater than or equal to zero specifies the maximum amount of data that can be associated with the *t_snddis* and *t_rcvdis* functions; a value of -1 specifies that there is no limit on the amount of data sent with these abortive release functions; and a value of -2 specifies that the transport provider does not allow data to be sent with the abortive release functions.
servtype	This field specifies the service type supported by the transport provider, as described below.

If a transport user is concerned with protocol independence, the above sizes can be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the *t_alloc* function can be used to allocate these buffers. An error will result if a transport user exceeds the allowed data size on any function.

The *servtype* field of *info* can specify one of the following values on return:

T_COTS	The transport provider supports a connection-mode service but does not support the optional orderly release facility.
T_COTS_ORD	The transport provider supports a connection-mode service with the optional orderly release facility.
T_CLTS	The transport provider supports a connectionless-mode service. For this service type, *t_open* will return -2 for *etsdu*, *connect*, and *discon*.

A single transport endpoint can 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:

[TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

open(2).
CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.

DIAGNOSTICS

t_open returns a valid file descriptor on success and -1 on failure and *t_errno* is set to indicate the error.

NAME

t_optmgmt - manage options for a transport endpoint

SYNOPSIS

```
#include <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 specifies the action to take with those options.

The options are represented by a *netbuf* [see *intro*(3); also for *len*, *buf* and *maxlen*] structure in a manner similar to the address in *t_bind*. *req* requests 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 are imposed by the transport provider.

The *flags* field of *req* can specify one of the following actions:

T_NEGOTIATE	This action enables the user to negotiate the values of the options specified in *req* with the transport provider. The provider will evaluate the requested options and negotiate the values, returning the negotiated values through *ret*.
T_CHECK	This action enables the user to verify whether or not the options specified in *req* are supported by the transport

	provider. On return, the *flags* field of *ret* will have either T_SUCCESS or T_FAILURE set to indicate to the user whether or not the options are supported. These flags are only meaningful for the T_CHECK request.
T_DEFAULT	This action enables a user to retrieve the default options supported by the transport provider into the *opt* field of *ret*. In *req*, the *len* field of *opt* must be zero and the *buf* field may be NULL.

If issued as part of the connectionless-mode service, *t_optmgmt* may block due to flow control constraints. The function will not complete until the transport provider has processed all previously sent data units.

On failure, *t_errno* can be set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TOUTSTATE]	The function was issued in the wrong sequence.
[TACCES]	The user does not have permission to negotiate the specified options.
[TBADOPT]	The specified protocol options were in an incorrect format or contained illegal information.
[TBADFLAG]	An invalid flag was specified.
[TBUFOVFLW]	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 *ret* will be discarded.
[TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

intro(3), t_getinfo(3N), t_open(3N).
CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.

DIAGNOSTICS

The *t_optmgmt* call returns 0 on success and -1 on failure and *t_errno* is set to indicate the error.

NAME

t_rcv - receive data or expedited data sent over a connection

SYNOPSIS

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 will arrive, *buf* points to a receive buffer where user data will be 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 will wait for data to arrive if none is currently available. However, if O_NDELAY is set (via *t_open* or *fcntl*), *t_rcv* will execute in asynchronous mode and will fail 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 TSDU. The end of the TSDU 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* will set T_EXPEDITED and T_MORE on return from the initial call. Subsequent calls to retrieve the remaining ETSDU will 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 will be suspended until the ETSDU has been processed. Only after the full ETSDU has been retrieved (T_MORE not set) will the remainder of the TSDU be available to the user.

On failure, *t_errno* can be set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TNODATA]	O_NDELAY was set, but no data is currently available from the transport provider.
[TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

t_open(3N), t_snd(3N).
CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.

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.

NAME

t_rcvconnect - receive the confirmation from a connect request

SYNOPSIS

```
#include <tiuser.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 will be established on successful completion of this function.

fd identifies the local transport endpoint where communication will be 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 *intro*(3). 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 *intro*(3)] 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_NDELAY 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*.

On failure, *t_errno* can 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 *call* will be discarded. The provider's state, as seen by the user, will be changed to DATAXFER.
[TNODATA]	O_NDELAY was set, but a connect confirmation has not yet arrived.
[TLOOK]	An asynchronous event has occurred on this transport connection and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

intro(3), t_accept(3N), t_bind(3N), t_connect(3N), t_listen(3N), t_open(3N).
CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.

DIAGNOSTICS

t_rcvconnect returns 0 on success and -1 on failure and *t_errno* is set to indicate the error.

NAME

t_rcvdis - retrieve information from disconnect

SYNOPSIS

#include <tiuser.h>

t_rcvdis(fd, discon)
int fd;
struct t_discon *discon;

DESCRIPTION

This function identifies the cause of a disconnect and retrieves 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 *intro*(3). *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* can 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 will be discarded. However, if a user has retrieved more than one outstanding connect indication (via *t_listen*) and *discon* is NULL, the user will be 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.
[TNODIS]	No disconnect indication currently exists on the specified 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, will change to T_IDLE, and the

	disconnect indication information to be returned in *discon* will be discarded.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

intro(3), t_connect(3N), t_listen(3N), t_open(3N), t_snddis(3N).
CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.

DIAGNOSTICS

t_rcvdis returns 0 on success and -1 on failure and *t_errno* is set to indicate the error.

NAME

t_rcvrel - acknowledge receipt of an orderly release indication

SYNOPSIS

#include <tiuser.h>

t_rcvrel(fd)
int fd;

DESCRIPTION

This function acknowledges receipt of an orderly release indication. *fd* identifies the local transport endpoint where the connection exists. After receipt of this indication, the user cannot receive more data because such an attempt will block forever. However, the user can 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* can be set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TNOREL]	No orderly release indication currently exists on the specified transport endpoint.
[TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

t_open(3N), t_sndrel(3N).
CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.

DIAGNOSTICS

t_rcvrel returns 0 on success and -1 on failure *t_errno* is set to indicate the error.

NAME

t_rcvudata - receive a data unit

SYNOPSIS

```
#include <tiuser.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 will be 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;
struct netbuf opt;
struct netbuf udata;
```

The *maxlen* [see *netbuf*in *intro*(3)] 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 will wait for a data unit to arrive if none is currently available. However, if O_NDELAY is set (via *t_open* or *fcntl*), *t_rcvudata* will execute in asynchronous mode and will fail 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 will be filled and T_MORE will be 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) will return zero for the length of the address and options until the full data unit has been received.

On failure, *t_errno* can be set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TNODATA]	O_NDELAY was set, but no data units are currently available from the transport provider.

[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 *unitdata* will be discarded.
[TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

intro(3), t_rcvuderr(3N), t_sndudata(3N).
CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.

DIAGNOSTICS

t_rcvudata returns 0 on successful completion and -1 on failure and *t_errno* is set to indicate the error.

NAME

t_rcvuderr - receive a unit data error indication

SYNOPSIS

#include <tiuser.h>

int t_rcvuderr(fd, uderr)
int fd;
struct t_uderr *uderr;

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 will be received, and *uderr* points to a *t_uderr* structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
long      error;
```

netbuf is described in *intro*(3). The *maxlen* [see *netbuf* in *intro*(3)] 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* will simply clear the error indication without reporting any information 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.
[TNOUDERR]	No unit data error indication currently exists on the specified 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 *uderr* will be discarded.

[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

intro(3), t_rcvudata(3N), t_sndudata(3N).
CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.

DIAGNOSTICS

t_rcvuderr returns 0 on successful completion and -1 on failure and *t_errno* is set to indicate the error.

NAME

t_snd - send data or expedited data over a connection

SYNOPSIS

#include <tiuser.h>

```
int t_snd(fd, buf, nbytes, flags)
int fd;
char *buf;
unsigned nbytes;
int flags;
```

DESCRIPTION

This function sends 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_NDELAY is set (via *t_open* or *fcntl*), *t_snd* will execute in asynchronous mode and will fail immediately if there are flow control restrictions.

Even when there are no flow control restrictions, *t_snd* will wait if STREAMS internal resources are not available, regardless of the state of O_NDELAY.

On successful completion, *t_snd* returns the number of bytes accepted by the transport provider. Normally this will be equal to the number of bytes specified in *nbytes*. However, if O_NDELAY is set, it is possible that only part of the data will be accepted by the transport provider. In this case, *t_snd* will set T_MORE for the data that was sent (see below) and will return a value less than *nbytes*. If *nbytes* is zero, no data will be passed to the provider and *t_snd* will return zero.

If T_EXPEDITED is set in *flags*, the data will be sent as expedited data, and will be 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* will follow with more data for the current TSDU. The end of the TSDU (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 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.

The size of each TSDU or ETSDU must not exceed the limits of the corresponding parameters of the transport provider as returned by *t_open* or *t_getinfo*. (Note that *t_getinfo* returns the limits of several parameters, not just those of TSDU or ETSDU.) If the size is exceeded, a TSYSERR with system error EPROTO will occur. 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 will fail 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 will generate a TSYSERR with system error EPROTO (which may be reported in the manner described above).

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]	O_NDELAY was set, but the flow control mechanism prevented the transport provider from accepting data at this time.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error [see *intro*(2)] has been detected during execution of this function.

SEE ALSO

t_open(3N), t_rcv(3N).
CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.

DIAGNOSTICS

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.

NAME

t_snddis - send user-initiated disconnect request

SYNOPSIS

```
#include <tiuser.h>

int t_snddis(fd, call)
int fd;
struct t_call *call;
```

DESCRIPTION

This function initiates an abortive release on an already established connection or rejects 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 that contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

netbuf is described in *intro*(3). 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 want 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 cannot 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 will be 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.
[TOUTSTATE]	The function was issued in the wrong sequence. The transport provider's outgoing queue may be flushed, so data may be lost.
[TBADDATA]	The amount of user data specified was not within the bounds allowed by the transport provider. The transport

	provider's outgoing queue will be 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 will be flushed, so data may be lost.
[TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

intro(3), t_connect(3N), t_getinfo(3N), t_listen(3N), t_open(3N).
CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.

DIAGNOSTICS

t_snddis returns 0 on success and -1 on failure and *t_errno* is set to indicate the error.

NAME

t_sndrel - initiate an orderly release

SYNOPSIS

#include <tiuser.h>

int t_sndrel(fd)
int fd;

DESCRIPTION

This function initiates 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 cannot send any more data over the connection. However, a user can 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]	O_NDELAY was set, but the flow control mechanism prevented the transport provider from accepting the function at this time.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

t_open(3N), t_rcvrel(3N).
CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.

DIAGNOSTICS

t_sndrel returns 0 on success and -1 on failure and *t_errno* is set to indicate the error.

NAME

t_sndudata - send a data unit

SYNOPSIS

```
#include <tiuser.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 will be 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 *intro*(3). 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 can 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 can use default options.

If the *len* field of *udata* is zero, no data unit will be passed to the transport provider; *t_sndudata* will 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_NDELAY is set (via *t_open* or *fcntl*), *t_sndudata* will execute in asynchronous mode and will fail 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 will generate an EPROTO protocol error. (See TSYSERR below.)

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]	O_NDELAY was set, but the flow control mechanism prevented the transport provider from accepting data at this time.

[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

intro(3), t_rcvudata(3N), t_rcvuderr(3N).
CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.

DIAGNOSTICS

t_sndudata returns 0 on successful completion and -1 on failure *t_errno* is set to indicate the error.

NAME

t_sync - synchronize transport library

SYNOPSIS

#include <tiuser.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*(2), *dup*(2), or as a result of a *fork*(2) and *exec*(2)] to an initialized transport endpoint, assuming that the file descriptor made a reference to 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 will fail.

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 has occurred during execution of this function.

SEE ALSO

dup(2), exec(2), fork(2), open(2).
CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.

DIAGNOSTICS

t_sync 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 can be one of the following:

T_UNBND	unbound
T_IDLE	idle
T_OUTCON	outgoing connection pending
T_INCON	incoming connection pending
T_DATAXFER	data transfer
T_OUTREL	outgoing orderly release (waiting for an orderly release indication)
T_INREL	incoming orderly release (waiting for an orderly release request)

NAME

t_unbind - disable a transport endpoint

SYNOPSIS

#include <tiuser.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* (3N). On completion of this call, no further data or events destined for this transport endpoint will be 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.
[TOUTSTATE]	The function was issued in the wrong sequence.
[TLOOK]	An asynchronous event has occurred on this transport endpoint.
[TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

t_bind(3N).
CTIX Network Programmer's Primer.
UNIX System V Release 3.2 Network Programmer's Guide.

DIAGNOSTICS

t_unbind returns 0 on success and -1 on failure and *t_errno* is set to indicate the error.

NAME

tmpfile - create a temporary file

SYNOPSIS

#include <stdio.h>

FILE *tmpfile ()

DESCRIPTION

tmpfile creates a temporary file using a name generated by *tmpnam*(3S), and returns a corresponding FILE pointer. If the file cannot be opened, an error message is printed using *perror*(3C), and a NULL pointer is returned. The file will automatically be deleted when the process using it terminates. The file is opened for update (w+).

SEE ALSO

creat(2), unlink(2), fopen(3S), mktemp(3C), perror(3C), stdio(3S), tmpnam(3S).

NAME

tmpnam, tempnam - create a name for a temporary file

SYNOPSIS

```
#include <stdio.h>

char *tmpnam (s)
char *s;

char *tempnam (dir, pfx)
char *dir, *pfx;
```

DESCRIPTION

These functions generate file names that can safely be used for a temporary file.

tmpnam always generates a file name using the path-prefix defined as **P_tmpdir** in the **<stdio.h>** header file. If *s* is NULL, *tmpnam* leaves its result in an internal static area and returns a pointer to that area. The next call to *tmpnam* will destroy the contents of the area. If *s* is not NULL, it is assumed to be the address of an array of at least **L_tmpnam** bytes, where **L_tmpnam** is a constant defined in **<stdio.h>**; *tmpnam* places its result in that array and returns *s*.

tempnam allows the user to control the choice of a directory. The argument *dir* points to the 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** will be used as a last resort. This entire sequence can be up-staged by providing an environment variable, **TMPDIR**, in the user's environment, whose value is 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-file name.

tempnam uses *malloc*(3C) to get space for the constructed file name and returns a pointer to this area. Thus, any pointer value returned from *tempnam* may serve as an argument to *free* [see *malloc*(3C)]. If *tempnam* cannot return the expected result for any reason—that is, *malloc*(3C) failed—or none of the above mentioned attempts to find an appropriate directory was successful, a NULL pointer will be returned. to remove the file when its use is ended.

SEE ALSO

creat(2), unlink(2), fopen(3S), malloc(3C), mktemp(3C), tmpfile(3S).

CAVEATS

If called more than 17,576 times in a single process, these functions will start recycling previously used names.

Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using these functions or *mktemp*, and the file names are chosen to render duplication by other means unlikely.

NOTES

These functions generate a different file name each time they are called.

Files created using these functions and either *fopen*(3S) or *creat*(2) are temporary only in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to use *unlink*(2)

NAME

trig: sin, cos, tan, asin, acos, atan, atan2 - trigonometric functions

SYNOPSIS

```
#include <math.h>

double sin (x)
double x;

double cos (x)
double x;

double tan (x)
double x;

double asin (x)
double x;

double acos (x)
double x;

double atan (x)
double x;

double atan2 (y, x)
double y, x;
```

DESCRIPTION

sin, *cos* and *tan* return respectively the sine, cosine and tangent of their argument, *x*, measured in radians.

asin returns the arcsine of *x*, in the range [-π/2,π/2].

acos returns the arccosine of *x*, in the range [0,π].

atan returns the arctangent of *x*, in the range [-π/2,π/2].

atan2 returns the arctangent of *y/x*, in the range (-π,π], using the signs of both arguments to determine the quadrant of the return value.

SEE ALSO

matherr(3M).

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 can be changed with the function *matherr*(3M).

SEE ALSO

matherr(3M).

NAME

tsearch, tfind, tdelete, twalk - manage binary search trees

SYNOPSIS

#include <search.h>

char *tsearch ((char *) key, (char **) rootp, compar)
int (*compar)( );

char *tfind ((char *) key, (char **) rootp, compar)
int (*compar)( );

char *tdelete ((char *) key, (char **) rootp, compar)
int (*compar)( );

void twalk ((char *) root, action)
void (*action)( );

DESCRIPTION

tsearch, tfind, tdelete, and *twalk* are routines for manipulating binary search trees. They are generalized from Knuth (6.2.2) Algorithms T and D. All comparisons are done with a user-supplied routine. This routine is called with two arguments, the pointers to the elements being compared. It returns an integer less than, equal to, or greater than 0, according to whether the first argument is to be considered less than, equal to, or greater than the second argument. The comparison function need not compare every byte, so arbitrary data can be contained in the elements in addition to the values being compared.

tsearch is used to build and access the tree. **key** is a pointer to a datum to be accessed or stored. If there is a datum in the tree equal to *key (the value pointed to by key), a pointer to this found datum is returned. Otherwise, *key is inserted, and a pointer to it returned. Only pointers are copied, so the calling routine must store the data. **rootp** points to a variable that points to the root of the tree. A NULL value for the variable pointed to by **rootp** denotes an empty tree; in this case, the variable will be set to point to the datum which will be at the root of the new tree.

Like *tsearch*, *tfind* will search for a datum in the tree, returning a pointer to it if found. However, if it is not found, *tfind* will return a NULL pointer. The arguments for *tfind* are the same as for *tsearch*.

tdelete deletes a node from a binary search tree. The arguments are the same as for *tsearch*. The variable pointed to by **rootp** will be changed if the deleted node was the root of the tree. *tdelete* returns a pointer to the parent of the deleted node, or a NULL pointer if the node is not found.

twalk traverses a binary search tree. root is the root of the tree to be traversed. (Any node in a tree can be used as the root for a walk below that node.) *action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The second argument is a value from an enumeration data type *typedef enum { preorder, postorder, endorder, leaf } VISIT;* (defined in the *<search.h>* header file), depending on whether this is the first, second or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level zero.

The pointers to the key and the root of the tree should be of type pointer-to-element, and cast to type pointer-to-character. Similarly, although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

EXAMPLE

The following code reads in strings and stores structures containing a pointer to each string and a count of its length. It then walks the tree, printing out the stored strings and their lengths in alphabetical order.

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

struct node {           /* pointers to these are stored in the tree */
          char  *string;
          int length;
};
char string_space[10000];   /* space to store strings */
struct node nodes[500];     /* nodes to store */
struct node *root = NULL;   /* this points to the root */

main( )
{
          char  *strptr = string_space;
          struct node  *nodeptr = nodes;
          void print_node( ),  twalk( );
          int i = 0, node_compare( );

          while (gets(strptr) != NULL && i++ < 500)  {
                    /* set node */
                    nodeptr->string = strptr;
                    nodeptr->length = strlen(strptr);
                    /* put node into the tree */
```

NAME

tsearch, tfind, tdelete, twalk - manage binary search trees

SYNOPSIS

#include <search.h>

char *tsearch ((char *) key, (char **) rootp, compar)
int (*compar)( );

char *tfind ((char *) key, (char **) rootp, compar)
int (*compar)( );

char *tdelete ((char *) key, (char **) rootp, compar)
int (*compar)( );

void twalk ((char *) root, action)
void (*action)( );

DESCRIPTION

tsearch, tfind, tdelete, and *twalk* are routines for manipulating binary search trees. They are generalized from Knuth (6.2.2) Algorithms T and D. All comparisons are done with a user-supplied routine. This routine is called with two arguments, the pointers to the elements being compared. It returns an integer less than, equal to, or greater than 0, according to whether the first argument is to be considered less than, equal to, or greater than the second argument. The comparison function need not compare every byte, so arbitrary data can be contained in the elements in addition to the values being compared.

tsearch is used to build and access the tree. **key** is a pointer to a datum to be accessed or stored. If there is a datum in the tree equal to *key (the value pointed to by key), a pointer to this found datum is returned. Otherwise, *key is inserted, and a pointer to it returned. Only pointers are copied, so the calling routine must store the data. **rootp** points to a variable that points to the root of the tree. A NULL value for the variable pointed to by **rootp** denotes an empty tree; in this case, the variable will be set to point to the datum which will be at the root of the new tree.

Like *tsearch*, *tfind* will search for a datum in the tree, returning a pointer to it if found. However, if it is not found, *tfind* will return a NULL pointer. The arguments for *tfind* are the same as for *tsearch*.

tdelete deletes a node from a binary search tree. The arguments are the same as for *tsearch*. The variable pointed to by **rootp** will be changed if the deleted node was the root of the tree. *tdelete* returns a pointer to the parent of the deleted node, or a NULL pointer if the node is not found.

twalk traverses a binary search tree. **root** is the root of the tree to be traversed. (Any node in a tree can be used as the root for a walk below that node.) *action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The second argument is a value from an enumeration data type *typedef enum { preorder, postorder, endorder, leaf } VISIT;* (defined in the *<search.h>* header file), depending on whether this is the first, second or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level zero.

The pointers to the key and the root of the tree should be of type pointer-to-element, and cast to type pointer-to-character. Similarly, although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

EXAMPLE

The following code reads in strings and stores structures containing a pointer to each string and a count of its length. It then walks the tree, printing out the stored strings and their lengths in alphabetical order.

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

struct node {          /* pointers to these are stored in the tree */
          char  *string;
          int length;
};
char string_space[10000];   /* space to store strings */
struct node nodes[500];     /* nodes to store */
struct node *root = NULL;   /* this points to the root */

main( )
{
          char  *strptr = string_space;
          struct node  *nodeptr = nodes;
          void print_node( ),  twalk( );
          int i = 0, node_compare( );

          while (gets(strptr) != NULL && i++ < 500)  {
                    /* set node */
                    nodeptr->string = strptr;
                    nodeptr->length = strlen(strptr);
```

```
                        /* put node into the tree */
                        (void) tsearch((char *)nodeptr,
                                (char **) &root, node_compare);
                        /* adjust pointers, don't overwrite tree */
                        strptr += nodeptr->length + 1;
                        nodeptr++;
                }
                twalk((char *)root, print_node);
        }
                /* This routine compares two nodes, based on an
                   alphabetical ordering of the string field.  */
        int
        node_compare(node1, node2)
        char  *node1,  *node2;
        {
                return strcmp(((struct node *)node1)->string,
                ((struct node *) node2)->string);
        }
                /* This routine prints out a node, the first time
                   twalk encounters it.  */
        void
        print_node(node, order, level)
        char  **node;
        VISIT  order;
        int level;
        {
                if (order == preorder || order == leaf)  {
                        (void)printf("string = %20s,  length = %d\n",
                    (*((struct node **)node))->string,
                    (*((struct node **)node))->length);
                }
        }
```

SEE ALSO

bsearch(3C), hsearch(3C), lsearch(3C).

DIAGNOSTICS

A NULL pointer is returned by *tsearch* if there is not enough space available to create a new node.

A NULL pointer is returned by *tfind* and *tdelete* if **rootp** is NULL on entry.

If the datum is found, both *tsearch* and *tfind* return a pointer to it. If not, *tfind* returns NULL, and *tsearch* returns a pointer to the inserted item.

CAVEAT

If the calling function alters the pointer to the root, results are unpredictable.

WARNINGS

The **root** argument to *twalk* is one level of indirection less than the **rootp** arguments to *tsearch* and *tdelete*.

There are two nomenclatures used to refer to the order in which tree nodes are visited. *tsearch* uses preorder, postorder, and endorder in respective order to refer to visting 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.

NAME

ttyname, isatty - find name of a terminal

SYNOPSIS

char *ttyname (fildes)
int fildes;

int isatty (fildes)
int fildes;

DESCRIPTION

ttyname returns a pointer to a string containing the null-terminated path name of the terminal device associated with file descriptor *fildes*.

isatty returns 1 if *fildes* is associated with a terminal device, 0 otherwise.

FILES

/dev/*

DIAGNOSTICS

ttyname returns a NULL pointer if *fildes* does not describe a terminal device in directory **/dev**.

CAVEAT

The return value points to static data whose content is overwritten by each call.

NAME

ttyslot - find the slot in the utmp file of the current user

SYNOPSIS

int ttyslot ( )

DESCRIPTION

ttyslot returns the index of the current user's entry in the **/etc/utmp** file. This is accomplished by actually scanning the file **/etc/inittab** for the name of the terminal associated with the standard input, the standard output, or the error output (0, 1, or 2).

FILES

/etc/inittab
/etc/utmp

SEE ALSO

getut(3C), ttyname(3C).

DIAGNOSTICS

A value of 0 is returned if an error was encountered while searching for the terminal name or if none of the above file descriptors is associated with a terminal device.

NAME

ungetc - push character back into input stream

SYNOPSIS

#include <stdio.h>

int ungetc (c, stream)
int c;
FILE *stream;

DESCRIPTION

ungetc inserts the character *c* into the buffer associated with an input *stream*. That character, *c*, will be returned by the next *getc*(3S) call on that *stream*. *ungetc* returns *c* and leaves the file *stream* unchanged.

One character of pushback is guaranteed, provided something has already been read from the stream and the stream is actually buffered.

If *c* equals **EOF**, *ungetc* does nothing to the buffer and returns **EOF**.

fseek(3S) erases all memory of inserted characters.

SEE ALSO

fseek(3S), getc(3S), setbuf(3S), stdio(3S).

DIAGNOSTICS

ungetc returns **EOF** if it cannot insert the character.

BUGS

When *stream* is *stdin*, one character may be pushed back onto the buffer without a previous read statement.

NAME

vprintf, vfprintf, vsprintf - print formatted output of a varargs argument list

SYNOPSIS

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

int vprintf (format, ap)
char *format;
va_list ap;

int vfprintf (stream, format, ap)
FILE *stream;
char *format;
va_list ap;

int vsprintf (s, format, ap)
char *s, *format;
va_list ap;
```

DESCRIPTION

vprintf, *vfprintf*, and *vsprintf* are the same as *printf*, *fprintf*, and *sprintf* respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by *varargs*(5).

EXAMPLE

The following demonstrates the use of *vfprintf* to write an error routine.

```
#include <stdio.h>
#include <varargs.h>
          .
          .
          .
/*
 *        error should be called like
 *    error(function_name, format, arg1, arg2...);  */
/*VARARGS*/
void
error(va_alist)
/* Note that the function_name and format arguments */
/* cannot be separately declared because of the */
/* definition of varargs.  */
va_dcl
{
          va_list args;
```

```
                char *fmt;

                va_start(args);
                /* print out name of function causing error */
                (void)fprintf(stderr, "ERROR in %s: ", va_arg(args,
                        char *));
                fmt = va_arg(args, char *);
                /* print out remainder of message */
                (void)vfprintf(stderr, fmt, args);
                va_end(args);
                (void)abort( );
        }
```

SEE ALSO

printf(3S), varargs(5).