Stardent

PROGRAMMER'S REFERENCE MANUAL, VOL. I

Stardent Computer Inc.

Part Number: 340-0121-01

Change History

340-0021-02	Original
340-0021-03	Software Release 2.0
340-0122-01	January, 1990

CONTENTS

2. System Calls

NOTE
Entries of the form *execle(2)(exec)* indicate that the function listed first is described in the entry for the function given in parentheses.

3X. Specialized Libraries

4. File Formats

PERMUTED INDEX

tigr(7)	1500/3000 graphics interface	tigr(7)
/ltol3(3C) convert between	3-byte integers and long integers	l3tol(3C)
Driver fcast(7) Dupont	4CAST Digital Color Printer	fcast(7)
long integer and base-64 ASCII/	a64l(3C) l64a(3C) convert between	a64l(3C)
	abort(3C) generate an IOT fault	abort(3C)
value	abs(3C) return integer absolute	abs(3C)
abs(3C) return integer	absolute value	abs(3C)
Euclidean distance, complex	absolute value /cabs(3M)	hypot(3M)
/floor, ceiling, remainder,	absolute value functions	floor(3M)
/endutent(3C) utmpname(3C)	access utmp file entry	getut(3C)
dials(3) subroutines for	accessing the dials	dials(3)
tablet(3) subroutines for	accessing the tablet	tablet(3)
acct(4) per-process	accounting file format	acct(4)
file format	acct(4) per-process accounting	acct(4)
/sin(3M) cos(3M) tan(3M) asin(3M)	acos(3M) atan(3M) atan2(3M)/	trig(3M)
hyperbolic functions asinh(3M)	acosh(3M) atanh(3M) inverse	asinh(3M)
putenv(3C) change or	add value to environment	putenv(3C)
arp(7)	Address Resolution Protocol	arp(7)
mallinfo(3X) fast main memory	allocator /calloc(3X) mallopt(3X)	malloc(3X)
calloc(3C) main memory	allocator /free(3C) realloc(3C)	malloc(3C)
link editor output	a.out(4) common assembler and	a.out(4)
	ar(4) common archive file format	ar(4)
manifest constants for Stardent 1500/3000	architecture values(5)	values(5)
cpio(4) format of cpio	archive	cpio(4)
ar(4) common	archive file format	ar(4)
tar(4) tape	archive file format	tar(4)
formatted output of a varargs	argument list /vsprintf(3S) print	vprintf(3S)
varargs(5) get variable	argument lists	varargs(5)
getopt(3C) get option letter from	argument vector	getopt(3C)
Protocol	arp(7) Address Resolution	arp(7)
routines to find the index in an	array of the element having/ /of	IDAMAX(3C)
ascii(5) map of	ASCII character set	ascii(5)
between long integer and base-64	ASCII string /l64a(3C) convert	a64l(3C)
convert date and time to	ASCII /tzset(3) tzsetwall(3)	ctime(3)
set	ascii(5) map of ASCII character	ascii(5)
ctime(3) localtime(3) gmtime(3)	asctime(3) tzset(3) tzsetwall(3)/	ctime(3)
trig(3M) sin(3M) cos(3M) tan(3M)	asin(3M) acos(3M) atan(3M)/	trig(3M)
inverse hyperbolic functions	asinh(3M) acosh(3M) atanh(3M)	asinh(3M)
a.out(4) common	assembler and link editor output	a.out(4)
assertion	assert(3X) verify program	assert(3X)
assert(3X) verify program	assertion	assert(3X)
setbuf(3S) setvbuf(3S)	assign buffering to a stream	setbuf(3S)
/asin(3M) acos(3M) atan(3M)	atan2(3M) trigonometric functions	trig(3M)

regexp(5) regular expression	compile and match routines	regexp(5)
term(4) format of	compiled term file.	term(4)
/erfc(3M) error function and	complementary error function	erf(3M)
cabs(3M) Euclidean distance,	complex absolute value hypot(3M)	hypot(3M)
resolver(5) resolver	configuration file	resolver(5)
an out-going terminal line	connection dial(3C) establish	dial(3C)
console(7)	console interface	console(7)
klog(7) kernel	console message logging device	klog(7)
	console(7) console interface	console(7)
math(5) math functions and	constants	math(5)
values(5) manifest	constants for Stardent 1500/3000 architecture	values(5)
file header for symbolic	constants unistd(4)	unistd(4)
fcntl(5) file	control options	fcntl(5)
tcp(7) internet transmission	control protocol (TCP)/	tcp(7)
tty(7)	controlling terminal interface	tty(7)
_toupper(3C) _tolower(3C)/	conv(3C) toupper(3C) tolower(3C)	conv(3C)
term(5)	conventional names for terminals	term(5)
and long/ l3tol(3C) ltol3(3C)	convert between 3-byte integers	l3tol(3C)
base-64 ASCII/ a64l(3C) l64a(3C)	convert between long integer and	a64l(3C)
/asctime(3) tzset(3) tzsetwall(3)	convert date and time to ASCII	ctime(3)
ecvt(3C) fcvt(3C) gcvt(3C)	convert floating-point number to/	ecvt(3C)
scanf(3S) fscanf(3S) sscanf(3S)	convert formatted input	scanf(3S)
strtod(3C) atof(3C)	convert string to/	strtod(3C)
strtol(3C) atol(3C) atoi(3C)	convert string to integer	strtol(3C)
core(4) format of	core image file	core(4)
mem(7) kmem(7)	core memory	mem(7)
	core(4) format of core image file	core(4)
atan(3M)/ trig(3M) sin(3M)	cos(3M) tan(3M) asin(3M) acos(3M)	trig(3M)
functions sinh(3M)	cosh(3M) tanh(3M) hyperbolic	sinh(3M)
cpio(4) format of	cpio archive	cpio(4)
	cpio(4) format of cpio archive	cpio(4)
clock(3C) report	CPU time used	clock(3C)
timing functions	cputim(3) systim(3) secnds(3)	cputim(3)
file tmpnam(3S) tempnam(3S)	create a name for a temporary	tmpnam(3S)
tmpfile(3S)	create a temporary file	tmpfile(3S)
generate hashing encryption	crypt(3C) setkey(3C) encrypt(3C)	crypt(3C)
encryption functions	crypt(3X) password and file	crypt(3X)
for terminal	ctermid(3S) generate file name	ctermid(3S)
asctime(3) tzset(3) tzsetwall(3)/	ctime(3) localtime(3) gmtime(3)	ctime(3)
islower(3C) isdigit(3C)/	ctype(3C) isalpha(3C) isupper(3C)	ctype(3C)
the slot in the utmp file of the	current user ttyslot(3C) find	ttyslot(3C)
getcwd(3C) get path-name of	current working directory	getcwd(3C)
scr_dump(4) format of	curses screen image file.	scr_dump(4)
handling and optimization/	curses(3X) terminal screen	curses(3X)
name of the user	cuserid(3S) get character login	cuserid(3S)
ttys(5) terminal initialization	data	ttys(5)
hosts(4) host name	data base	hosts(4)
networks(4) network name	data base	networks(4)
protocols(4) protocol name	data base	protocols(4)
rhosts(4) host name	data base	rhosts(4)
services(4) service name	data base	services(4)
termcap(5) terminal capability	data base	termcap(5)
terminfo(4) terminal capability	data base	terminfo(4)
stat(5)	data returned by stat system call	stat(5)
types(5) primitive system	data types	types(5)
udp(7) user	datagram protocol (UDP) interface	udp(7)

/tzset(3) tzsetwall(3) convert	date and time to ASCII	ctime(3)
timezone(4) set	default system time zone	timezone(4)
routines vmath(5) include	definitions for vector math	vmath(5)
queuedefs(4) at/batch/cron queue	description file	queuedefs(4)
dup2(3C) duplicate an open file	descriptor	dup2(3C)
	/dev/bell(7) speaker driver	/dev/bell(7)
Parallel Port Driver	/dev/centrnix(7) Centronix	/dev/centrnix(7)
kernel console message logging	device klog(7)	klog(7)
dlb(7)	dial box	dlb(7)
terminal line connection	dial(3C) establish an out-going	dial(3C)
subroutines for accessing the	dials dials(3)	dials(3)
accessing the dials	dials(3) subroutines for	dials(3)
fcast(7) Dupont 4CAST	Digital Color Printer Driver	fcast(7)
	dir(4) format of directories	dir(4)
dir(4) format of	directories	dir(4)
dirent(4) file system independent	directory entry	dirent(4)
get path-name of current working	directory getcwd(3C)	getcwd(3C)
rewinddir(3X) closedir(3X)	directory operations /seekdir(3X)	directory(3X)
readdir(3X) telldir(3X)/	directory(3X) opendir(3X)	directory(3X)
directory entry	dirent(4) file system independent	dirent(4)
dsk(7) SCSI	disk interface	dsk(7)
hypot(3M) cabs(3M) Euclidean	distance, complex absolute value	hypot(3M)
/lcong48(3C) generate uniformly	distributed pseudo-random numbers	drand48(3C)
	dlb(7) dial box	dlb(7)
udom(7) Unix	domain IPC driver	udom(7)
/atof(3C) convert string to	double-precision number	strtod(3C)
lrand48(3C) nrand48(3C)/	drand48(3C) erand48(3C)	drand48(3C)
/dev/bell(7) speaker	driver	/dev/bell(7)
et(7) generic ethernet	driver	et(7)
kbd(7) keyboard	driver	kbd(7)
mouse(7) mouse	driver	mouse(7)
udom(7) Unix domain IPC	driver	udom(7)
Centronix Parallel Port	Driver /dev/centrnix(7)	/dev/centrnix(7)
4CAST Digital Color Printer	Driver fcast(7) Dupont	fcast(7)
	dsk(7) SCSI disk interface	dsk(7)
descriptor	dup2(3C) duplicate an open file	dup2(3C)
dup2(3C)	duplicate an open file descriptor	dup2(3C)
Printer Driver fcast(7)	Dupont 4CAST Digital Color	fcast(7)
convert floating-point number to/	ecvt(3C) fcvt(3C) gcvt(3C)	ecvt(3C)
program _end(3C) _etext(3C)	_edata(3C) last locations in	_end(3C)
common assembler and link	editor output a.out(4)	a.out(4)
time fputim(3)	elapsed floating point processor	fputim(3)
find the index in an array of the	element having largest or/ /to	IDAMAX(3C)
uuencode(4) format of an	encoded uuencode file	uuencode(4)
encryption crypt(3C) setkey(3C)	encrypt(3C) generate hashing	crypt(3C)
encrypt(3C) generate hashing	encryption crypt(3C) setkey(3C)	crypt(3C)
crypt(3X) password and file	encryption functions	crypt(3X)
last locations in program	_end(3C) _etext(3C) _edata(3C)	_end(3C)
group/ /getgrnam(3C) setgrent(3C)	endgrent(3C) fgetgrent(3C) get	getgrent(3C)
/getpwnam(3C) setpwent(3C)	endpwent(3C) fgetpwent(3C) get/	getpwent(3C)
utmp/ /pututline(3C) setutent(3C)	endutent(3C) utmpname(3C) access	getut(3C)
nlist(3C) get	entries from name list	nlist(3C)
putpwent(3C) write password file	entry	putpwent(3C)
file system independent directory	entry dirent(4)	dirent(4)
utmp(4) wtmp(4) utmp and wtmp	entry formats	utmp(4)
fgetgrent(3C) get group file	entry /setgrent(3C) endgrent(3C)	getgrent(3C)

fgetpwent(3C) get password file	entry /setpwent(3C) endpwent(3C)	getpwent(3C)
utmpname(3C) access utmp file	entry /setutent(3C) endutent(3C)	getut(3C)
	environ(5) user environment	environ(5)
environ(5) user	environment	environ(5)
putenv(3C) change or add value to	environment	putenv(3C)
profile(4) setting up an	environment at login time	profile(4)
getenv(3C) return value for	environment name	getenv(3C)
nrand48(3C)/ drand48(3C)	erand48(3C) lrand48(3C)	drand48(3C)
and complementary error function	erf(3M) erfc(3M) error function	erf(3M)
complementary error/ erf(3M)	erfc(3M) error function and	erf(3M)
sys_nerr(3C) system/ perror(3C)	errno(3C) sys_errlist(3C)	perror(3C)
error function erf(3M) erfc(3M)	error function and complementary	erf(3M)
error function and complementary	error function erf(3M) erfc(3M)	erf(3M)
sys_nerr(3C) system	error messages /sys_errlist(3C)	perror(3C)
line connection dial(3C)	establish an out-going terminal	dial(3C)
	et(7) generic ethernet driver	et(7)
locations in program _end(3C)	_etext(3C) _edata(3C) last	_end(3C)
et(7) generic	ethernet driver	et(7)
absolute/ hypot(3M) cabs(3M)	Euclidean distance, complex	hypot(3M)
regcmp(3X) regex(3X) compile and	execute regular expression	regcmp(3X)
sleep(3C) suspend	execution for interval	sleep(3C)
monitor(3C) prepare	execution profile	monitor(3C)
sqrt(3M) cbrt(3M) exponential,/	exp(3M) log(3M) log10(3M) pow(3M)	exp(3M)
square/ /pow(3M) sqrt(3M) cbrt(3M)	exponential, logarithm, power,	exp(3M)
routines regexp(5) regular	expression compile and match	regexp(5)
compile and execute regular	expression regcmp(3X) regex(3X)	regcmp(3X)
floor(3M) ceil(3M) fmod(3M)	fabs(3M) floor, ceiling,/	floor(3M)
/mallopt(3X) mallinfo(3X)	fast main memory allocator	malloc(3X)
abort(3C) generate an IOT	fault	abort(3C)
Color Printer Driver	fcast(7) Dupont 4CAST Digital	fcast(7)
flush a stream	fclose(3S) fflush(3S) close or	fclose(3S)
	fcntl(5) file control options	fcntl(5)
floating-point number/ ecvt(3C)	fcvt(3C) gcvt(3C) convert	ecvt(3C)
fopen(3S) freopen(3S)	fdopen(3S) open a stream	fopen(3S)
stream status/ ferror(3S)	feof(3S) clearerr(3S) fileno(3S)	ferror(3S)
fileno(3S) stream status/	ferror(3S) feof(3S) clearerr(3S)	ferror(3S)
stream fclose(3S)	fflush(3S) close or flush a	fclose(3S)
or word/ getc(3S) getchar(3S)	fgetc(3S) getw(3S) get character	getc(3S)
entry /setgrent(3C) endgrent(3C)	fgetgrent(3C) get group file	getgrent(3C)
entry /setpwent(3C) endpwent(3C)	fgetpwent(3C) get password file	getpwent(3C)
stream gets(3S)	fgets(3S) get a string from a	gets(3S)
core(4) format of core image	file	core(4)
group(4) group	file	group(4)
issue(4) issue identification	file	issue(4)
null(7) the null	file	null(7)
passwd(4) password	file	passwd(4)
sccsfile(4) format of SCCS	file	sccsfile(4)
term(4) format of compiled term	file.	term(4)
tmpfile(3S) create a temporary	file	tmpfile(3S)
fcntl(5)	file control options	fcntl(5)
dup2(3C) duplicate an open	file descriptor	dup2(3C)
crypt(3X) password and	file encryption functions	crypt(3X)
putpwent(3C) write password	file entry	putpwent(3C)
fgetgrent(3C) get group	file entry /endgrent(3C)	getgrent(3C)
fgetpwent(3C) get password	file entry /endpwent(3C)	getpwent(3C)
utmpname(3C) access utmp	file entry /endutent(3C)	getut(3C)

fileno(3S) stream status	inquiries /feof(3S) clearerr(3S)	ferror(3S)
abs(3C) return	integer absolute value	abs(3C)
/l64a(3C) convert between long	integer and base-64 ASCII string	a64l(3C)
atoi(3C) convert string to	integer strtol(3C) atol(3C)	strtol(3C)
/ltol3(3C) convert between 3-byte	integers and long integers	l3tol(3C)
between 3-byte integers and long	integers /ltol3(3C) convert	l3tol(3C)
console(7) console	interface	console(7)
dsk(7) SCSI disk	interface	dsk(7)
mtio(5) UNIX magtape manipulation	interface	mtio(5)
plot(4) graphics	interface	plot(4)
termio(7) general terminal	interface	termio(7)
tigr(7) 1500/3000 graphics	interface	tigr(7)
tty(7) controlling terminal	interface	tty(7)
protocol raw(7) raw	interface to internal network	raw(7)
user datagram protocol (UDP)	interface udp(7)	udp(7)
raw(7) raw interface to	internal network protocol	raw(7)
multiplexor ip(7)	internet protocol (IP)	ip(7)
protocol (TCP)/ tcp(7)	internet transmission control	tcp(7)
stdipc(3C) ftok(3C) standard	interprocess communication/	stdipc(3C)
sleep(3C) suspend execution for	interval	sleep(3C)
functions and libraries	intro(3) introduction to	intro(3)
formats	intro(4) introduction to file	intro(4)
miscellany	intro(5) introduction to	intro(5)
files	intro(7) introduction to special	intro(7)
intro(4)	introduction to file formats	intro(4)
libraries intro(3)	introduction to functions and	intro(3)
intro(5)	introduction to miscellany	intro(5)
intro(7)	introduction to special files	intro(7)
asinh(3M) acosh(3M) atanh(3M)	inverse hyperbolic functions	asinh(3M)
streamio(7) STREAMS	ioctl commands	streamio(7)
abort(3C) generate an	IOT fault	abort(3C)
ip(7) internet protocol	(IP) multiplexor	ip(7)
multiplexor	ip(7) internet protocol (IP)	ip(7)
udom(7) Unix domain	IPC driver	udom(7)
/isdigit(3C) isxdigit(3C)	isalnum(3C) isspace(3C)/	ctype(3C)
islower(3C)/ ctype(3C)	isalpha(3C) isupper(3C)	ctype(3C)
/isgraph(3C) iscntrl(3C)	isascii(3C) classify characters	ctype(3C)
terminal ttyname(3C)	isatty(3C) find name of a	ttyname(3C)
/isprint(3C) isgraph(3C)	iscntrl(3C) isascii(3C) classify/	ctype(3C)
/isupper(3C) islower(3C)	isdigit(3C) isxdigit(3C)/	ctype(3C)
/ispunct(3C) isprint(3C)	isgraph(3C) iscntrl(3C)/	ctype(3C)
ctype(3C) isalpha(3C) isupper(3C)	islower(3C) isdigit(3C)/	ctype(3C)
test for floating point NaN/	isnan(3C) isnand(3C) isnanf(3C)	isnan(3C)
floating point NaN/ isnan(3C)	isnand(3C) isnanf(3C) test for	isnan(3C)
point NaN/ isnan(3C) isnand(3C)	isnanf(3C) test for floating	isnan(3C)
/isspace(3C) ispunct(3C)	isprint(3C) isgraph(3C)/	ctype(3C)
/isalnum(3C) isspace(3C)	ispunct(3C) isprint(3C)/	ctype(3C)
/isxdigit(3C) isalnum(3C)	isspace(3C) ispunct(3C)/	ctype(3C)
system(3S)	issue a shell command	system(3S)
issue(4)	issue identification file	issue(4)
file	issue(4) issue identification	issue(4)
ctype(3C) isalpha(3C)	isupper(3C) islower(3C)/	ctype(3C)
/islower(3C) isdigit(3C)	isxdigit(3C) isalnum(3C)/	ctype(3C)
y1(3M) yn(3M) Bessel/ bessel(3M)	j0(3M) j1(3M) jn(3M) y0(3M)	bessel(3M)
yn(3M) Bessel/ bessel(3M) j0(3M)	j1(3M) jn(3M) y0(3M) y1(3M)	bessel(3M)
Bessel/ bessel(3M) j0(3M) j1(3M)	jn(3M) y0(3M) y1(3M) yn(3M)	bessel(3M)

/nrand48(3C) mrand48(3C)	jrand48(3C) srand48(3C)/	drand48(3C)
	kbd(7) keyboard driver	kbd(7)
device klog(7)	kernel console message logging	klog(7)
kbd(7)	keyboard driver	kbd(7)
logging device	klog(7) kernel console message	klog(7)
mem(7)	kmem(7) core memory	mem(7)
between 3-byte integers and long/	l3tol(3C) ltol3(3C) convert	l3tol(3C)
integer and base-64/ a64l(3C)	l64a(3C) convert between long	a64l(3C)
in an array of the element having	largest or smallest value /index	IDAMAX(3C)
/srand48(3C) seed48(3C)	lcong48(3C) generate uniformly/	drand48(3C)
parts of/ frexp(3C)	ldexp(3C) modf(3C) manipulate	frexp(3C)
getopt(3C) get option	letter from argument vector	getopt(3C)
update lsearch(3C)	lfind(3C) linear search and	lsearch(3C)
	lgamma(3M) log gamma function	lgamma(3M)
introduction to functions and	libraries intro(3)	intro(3)
operating system magnitude	limits limits(4)	limits(4)
magnitude limits	limits(4) operating system	limits(4)
establish an out-going terminal	line connection dial(3C)	dial(3C)
lsearch(3C) lfind(3C)	linear search and update	lsearch(3C)
a.out(4) common assembler and	link editor output	a.out(4)
nlist(3C) get entries from name	list	nlist(3C)
fsck checklist(4)	list of file systems processed by	checklist(4)
hosts.equiv(4)	list of trusted hosts	hosts.equiv(4)
output of a varargs argument	list /print formatted	vprintf(3S)
varargs(5) get variable argument	lists	varargs(5)
tzset(3) tzsetwall(3)/ ctime(3)	localtime(3) gmtime(3) asctime(3)	ctime(3)
_etext(3C) _edata(3C) last	locations in program _end(3C)	_end(3C)
	lockf(3C) record locking on files	lockf(3C)
lockf(3C) record	locking on files	lockf(3C)
gamma(3M)	log gamma function	gamma(3M)
lgamma(3M)	log gamma function	lgamma(3M)
cbrt(3M)/ exp(3M) log(3M)	log10(3M) pow(3M) sqrt(3M)	exp(3M)
sqrt(3M) cbrt(3M)/ exp(3M)	log(3M) log10(3M) pow(3M)	exp(3M)
/sqrt(3M) cbrt(3M) exponential,	logarithm, power, square root/	exp(3M)
klog(7) kernel console message	logging device	klog(7)
getlogin(3C) get	login name	getlogin(3C)
cuserid(3S) get character	login name of the user	cuserid(3S)
logname(3X) return	login name of user	logname(3X)
setting up an environment at	login time profile(4)	profile(4)
user	logname(3X) return login name of	logname(3X)
setjmp(3C)	longjmp(3C) non-local goto	setjmp(3C)
drand48(3C) erand48(3C)	lrand48(3C) nrand48(3C)/	drand48(3C)
search and update	lsearch(3C) lfind(3C) linear	lsearch(3C)
integers and long/ l3tol(3C)	ltol3(3C) convert between 3-byte	l3tol(3C)
limits(4) operating system	magnitude limits	limits(4)
mtio(5) UNIX	magtape manipulation interface	mtio(5)
mallopt(3X) mallinfo(3X) fast	main memory allocator /calloc(3X)	malloc(3X)
free(3C) realloc(3C) calloc(3C)	main memory allocator malloc(3C)	malloc(3C)
allocator /calloc(3X) mallopt(3X)	mallinfo(3X) fast main memory	malloc(3X)
calloc(3C) main memory allocator	malloc(3C) free(3C) realloc(3C)	malloc(3C)
calloc(3X) mallopt(3X)/	malloc(3X) free(3X) realloc(3X)	malloc(3X)
/free(3X) realloc(3X) calloc(3X)	mallopt(3X) mallinfo(3X) fast/	malloc(3X)
	man(5) format of man pages	man(5)
/tfind(3C) tdelete(3C) twalk(3C)	manage binary search trees	tsearch(3C)
/hcreate(3C) hdestroy(3C)	manage hash search tables	hsearch(3C)
architecture values(5)	manifest constants for Stardent 1500/3000	values(5)

sccsfile(4) format of	SCCS file	sccsfile(4)
	sccsfile(4) format of SCCS file	sccsfile(4)
screen image file.	scr_dump(4) format of curses	scr_dump(4)
package curses(3X) terminal	screen handling and optimization	curses(3X)
scr_dump(4) format of curses	screen image file.	scr_dump(4)
inittab(4)	script for the init process	inittab(4)
dsk(7)	SCSI disk interface	dsk(7)
bsearch(3C) binary	search a sorted table	bsearch(3C)
lsearch(3C) lfind(3C) linear	search and update	lsearch(3C)
hdestroy(3C) manage hash	search tables /hcreate(3C)	hsearch(3C)
twalk(3C) manage binary	search trees /tdelete(3C)	tsearch(3C)
cputim(3) systim(3)	secnds(3) timing functions	cputim(3)
/jrand48(3C) srand48(3C)	seed48(3C) lcong48(3C) generate/	drand48(3C)
/readdir(3X) telldir(3X)	seekdir(3X) rewinddir(3X)/	directory(3X)
services(4)	service name data base	services(4)
base	services(4) service name data	services(4)
ascii(5) map of ASCII character	set	ascii(5)
timezone(4)	set default system time zone	timezone(4)
in an array of the/ IDAMAX(3C) a	set of routines to find the index	IDAMAX(3C)
buffering to a stream	setbuf(3S) setvbuf(3S) assign	setbuf(3S)
/getgrgid(3C) getgrnam(3C)	setgrent(3C) endgrent(3C)/	getgrent(3C)
goto	setjmp(3C) longjmp(3C) non-local	setjmp(3C)
hashing encryption crypt(3C)	setkey(3C) encrypt(3C) generate	crypt(3C)
/getpwuid(3C) getpwnam(3C)	setpwent(3C) endpwent(3C)/	getpwent(3C)
login time profile(4)	setting up an environment at	profile(4)
gettydefs(4) speed and terminal	settings used by getty	gettydefs(4)
/getutline(3C) pututline(3C)	setutent(3C) endutent(3C)/	getut(3C)
stream setbuf(3S)	setvbuf(3S) assign buffering to a	setbuf(3S)
system(3S) issue a	shell command	system(3S)
ssignal(3C) gsignal(3C) software	signals	ssignal(3C)
rand(3C) srand(3C)	simple random-number generator	rand(3C)
acos(3M) atan(3M)/ trig(3M)	sin(3M) cos(3M) tan(3M) asin(3M)	trig(3M)
hyperbolic functions	sinh(3M) cosh(3M) tanh(3M)	sinh(3M)
interval	sleep(3C) suspend execution for	sleep(3C)
current/ ttyslot(3C) find the	slot in the utmp file of the	ttyslot(3C)
of the element having largest or	smallest value /index in an array	IDAMAX(3C)
ssignal(3C) gsignal(3C)	software signals	ssignal(3C)
qsort(3C) quicker	sort	qsort(3C)
bsearch(3C) binary search a	sorted table	bsearch(3C)
/dev/bell(7)	speaker driver	/dev/bell(7)
intro(7) introduction to	special files	intro(7)
fspec(4) format	specification in text files	fspec(4)
by getty gettydefs(4)	speed and terminal settings used	gettydefs(4)
output printf(3S) fprintf(3S)	sprintf(3S) print formatted	printf(3S)
exp(3M) log(3M) log10(3M) pow(3M)	sqrt(3M) cbrt(3M) exponential,/	exp(3M)
exponential, logarithm, power,	square root functions /cbrt(3M)	exp(3M)
generator rand(3C)	srand(3C) simple random-number	rand(3C)
/mrand48(3C) jrand48(3C)	srand48(3C) seed48(3C)/	drand48(3C)
input scanf(3S) fscanf(3S)	sscanf(3S) convert formatted	scanf(3S)
signals	ssignal(3C) gsignal(3C) software	ssignal(3C)
package stdio(3S)	standard buffered input/output	stdio(3S)
stdipc(3C) ftok(3C)	standard interprocess/	stdipc(3C)
stat(5) data returned by	stat system call	stat(5)
system call	stat(5) data returned by stat	stat(5)
clearerr(3S) fileno(3S) stream	status inquiries /feof(3S)	ferror(3S)
input/output package	stdio(3S) standard buffered	stdio(3S)

mnttab(4) mounted file system	table	mnttab(4)
binary search a sorted	table bsearch(3C)	bsearch(3C)
hdestroy(3C) manage hash search	tables hsearch(3C) hcreate(3C)	hsearch(3C)
tb(7)	tablet	tb(7)
subroutines for accessing the	tablet tablet(3)	tablet(3)
accessing the tablet	tablet(3) subroutines for	tablet(3)
trig(3M) sin(3M) cos(3M)	tan(3M) asin(3M) acos(3M)/	trig(3M)
sinh(3M) cosh(3M)	tanh(3M) hyperbolic functions	sinh(3M)
tar(4)	tape archive file format	tar(4)
	tar(4) tape archive file format	tar(4)
	tb(7) tablet	tb(7)
transmission control protocol	(TCP) multiplexor /internet	tcp(7)
control protocol (TCP)/	tcp(7) internet transmission	tcp(7)
binary/ tsearch(3C) tfind(3C)	tdelete(3C) twalk(3C) manage	tsearch(3C)
/opendir(3X) readdir(3X)	telldir(3X) seekdir(3X)/	directory(3X)
temporary file tmpnam(3S)	tempnam(3S) create a name for a	tmpnam(3S)
tmpfile(3S) create a	temporary file	tmpfile(3S)
tempnam(3S) create a name for a	temporary file tmpnam(3S)	tmpnam(3S)
term(4) format of compiled	term file.	term(4)
file.	term(4) format of compiled term	term(4)
terminals	term(5) conventional names for	term(5)
data base	termcap(5) terminal capability	termcap(5)
termcap(5)	terminal capability data base	termcap(5)
terminfo(4)	terminal capability data base	terminfo(4)
generate file name for	terminal ctermid(3S)	ctermid(3S)
ttys(5)	terminal initialization data	ttys(5)
termio(7) general	terminal interface	termio(7)
tty(7) controlling	terminal interface	tty(7)
dial(3C) establish an out-going	terminal line connection	dial(3C)
optimization package curses(3X)	terminal screen handling and	curses(3X)
gettydefs(4) speed and	terminal settings used by getty	gettydefs(4)
isatty(3C) find name of a	terminal ttyname(3C)	ttyname(3C)
term(5) conventional names for	terminals	term(5)
data base	terminfo(4) terminal capability	terminfo(4)
interface	termio(7) general terminal	termio(7)
isnan(3C) isnand(3C) isnanf(3C)	test for floating point NaN/	isnan(3C)
fspec(4) format specification in	text files	fspec(4)
manage binary search/ tsearch(3C)	tfind(3C) tdelete(3C) twalk(3C)	tsearch(3C)
interface	tigr(7) 1500/3000 graphics	tigr(7)
time zone	timezone(4) set default system	timezone(4)
cputim(3) systim(3) secnds(3)	timing functions	cputim(3)
file	tmpfile(3S) create a temporary	tmpfile(3S)
name for a temporary file	tmpnam(3S) tempnam(3S) create a	tmpnam(3S)
/_toupper(3C) _tolower(3C)	toascii(3C) translate characters	conv(3C)
pclose(3S) initiate pipe	to/from a process popen(3S)	popen(3S)
/tolower(3C) _toupper(3C)	_tolower(3C) toascii(3C)/	conv(3C)
conv(3C) toupper(3C)	tolower(3C) _toupper(3C)/	conv(3C)
conv(3C) toupper(3C) tolower(3C)	_toupper(3C) _tolower(3C)/	conv(3C)
_toupper(3C)/ conv(3C)	toupper(3C) tolower(3C)	conv(3C)
/_tolower(3C) toascii(3C)	translate characters	conv(3C)
(TCP)/ tcp(7) internet	transmission control protocol	tcp(7)
ftw(3C) walk a file	tree	ftw(3C)
twalk(3C) manage binary search	trees /tfind(3C) tdelete(3C)	tsearch(3C)
asin(3M) acos(3M) atan(3M)/	trig(3M) sin(3M) cos(3M) tan(3M)	trig(3M)
acos(3M) atan(3M) atan2(3M)	trigonometric functions /asin(3M)	trig(3M)
hosts.equiv(4) list of	trusted hosts	hosts.equiv(4)

NAME

intro – introduction to system calls and error numbers

SYNOPSIS

#include <errno.h>

DESCRIPTION

This section describes all of the System V Release 3 system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value. This is almost always –1 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 complete list of the error numbers and their names are defined in *<errno.h>*.

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

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

3 ESRCH No such process
: No process can be found corresponding to that specified by *pid* in *kill*(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 20480 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) [respectively, *write*(2)] request is made to a file which is open only for writing (respectively, reading).

10 ECHILD No children
: 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. Or a system call failed because of insufficient memory or swap space.

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.

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

14 EFAULT Bad address
The system encountered a hardware fault in attempting to use an argument of a system call.

15 ENOTBLK Block device required
A non-block file was mentioned where a block device was required, e.g., in *mount*(2).

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

20 ENOTDIR Not a directory
A non-directory was specified where a directory is required, for example in a path prefix or as an argument to *chdir*(2).

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

22 EINVAL Invalid argument
Some invalid argument (e.g., dismounting a non-mounted device; mentioning an undefined signal in *signal*(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 64) descriptors open at a time.

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 *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.

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 Math result not representable
: 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.

38 EL2NSYNC
: Level 2 not synchronized.

39 EL3HLT
: Level 3 halted.

40 EL3RST
: Level 3 reset.

41 ELNRNG
: Link number out of range.

42 EUNATCH
: Protocol driver not attached.

43 ENOCSI
: No CSI structure available.

44 EL2HLT
: Level 2 halted.

45 EDEADLK Deadlock condition
: A deadlock situation was detected and avoided. This error pertains to file and record locking. This condition is identical with the BSD error EWOULDBLOCK.

46 ENOLCK No record locks available
: 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.

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

61 ENODATA No data (for no delay I/O.

62 ETIME Timer expired
: 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 Out of 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 Package not installed
: This error occurs when users attempt to use a system call from a package which has not been installed.

66 EREMOTE The object 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 The link has been severed
: 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 on send
: 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.

75 ELBIN
: Inode is remote (not really an error).

76 EDOTDOT
: Cross mount point (not really an error).

77 EBADMSG Trying to read unreadable
: 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.

80 ENOTUNIQ
: Given login name is not unique.

81 EBADFD
: File descriptor invalid for this operation.

82 EREMCHG
: Remoted address changed.

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 too many libraries
: 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 System Administrator's Guide.

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

90 EINPROGRESS Operation now in progress
: An operation that takes a long time to complete (such as a *connect*(2)) was attempted on a non-blocking object (see *fcntl*(2)).

91 EALREADY Operation already in progress
: An operation was attempted on a non-blocking object that already had an operation in progress.

92 ENOTSOCK Socket operation on a non-socket
Self-explanatory.

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

94 EMSGSIZE Message too long
A message sent on a socket was larger than the internal message buffer or some other network limit.

95 EPROTOTYPE Protocol wrong type for socket
A protocol was specified that does not support the semantics of the socket type requested. For example, you cannot use the ARPA Internet UDP protocol with type SOCK_STREAM.

96 ENOPROTOOPT not available
A bad option or level was specified in a *getsockopt*(2) or *setsockopt*(2) call.

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

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

99 EOPNOTSUPP Operation not supported on socket
For example, trying to *accept* a connection on a datagram socket.

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

101 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 us NS addresses with ARPA Internet protocols.

102 EADDRINUSE Address already in use
Only one usage of each address is normally permitted.

103 EADDRNOTAVAIL Can't assign requested address
Normally results from an attempt to create a socket with an address not on this machine.

104 ENETDOWN Network is down
A socket operation encountered a dead network.

105 ENETUNREACH Network is unreachable
A socket operation was attempted to an unreachable network.

106 ENETRESET Network dropped connection on reset
The host you were connected to crashed and rebooted.

107 ECONNABORTED Software caused connection abort
A connection abort was caused internal to your host machine.

108 ECONNRESET Connection reset by peer
A connection was forcibly closed by a peer. This normally results from a loss of the connection on the remote socket due to a timeout or a reboot.

109 ENOBUFS No buffer space available
An operation on a socket or pipe was not performed because the system lacked sufficient buffer space or because a queue was full.

110 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 when already connected.

111 ENOTCONN Socket is not connected
A request to send or receive data was disallowed because the socket is not connected and (when sending on a datagram socket) no address was supplied.

112 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.

113 ETOOMANYREFS Too many references: can't splice

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

115 ECONNREFUSED Connection refused
No connection could be made becuase the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the foreign host.

116 ELOOP Too many levels of symbolic links
A pathname lookup involved more than 8 symbolic links.

117 ENAMETOOLONG File name too long
A component of a pathname exceeded 255 (MAXNAMELEN) characters, or an entire pathname exceeded 1023 (MAXPATHLEN-1) characters.

118 EHOSTDOWN Host is down
A socket operation failed because the destination host was down.

119 EHOSTUNREACH No route to host
A socket operation was attempted to an unreachable host.

120 ENOTEMPTY Directory not empty
A directory with entries other than "." and ".." was supplied to a remove directory or rename call.

121 EPROCLIM Too many processes

122 EUSERS Too many users

123 EDQUOT Disk quota exceeded
A *write* to an ordinary file, the creation of a directory or symbolic link, or the creation of a directory entry failed because the user's quota of disk blocks was exhausted, or the allocation of an inode for a newly created file failed because the user's quota of inodes was exhausted.

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 */
struct   region *shm_reg;    /*ptr to region structure */
char     pad[4];             /* for swap compatibility */
ushort   shm_lpid;           /* pid of last operation */
ushort   shm_cpid;           /* creator pid */
ushort   shm_nattch;         /* number of current attaches */
ushort   shm_cnattch;        /* used only for shminfo */
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 # */
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 *driver*s. 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*.

Thread A thread is a copy of the text, data, and stack regions of a process that is shared

SEE ALSO

intro(3).

NAME

access – determine accessibility of a file

SYNOPSIS

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

DESCRIPTION

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

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

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

[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

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), sigset(2).

DIAGNOSTICS

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

NAME

astat – get status of an asynchronous file read

SYNOPSIS

int astat(fildes) int fildes;

DESCRIPTION

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

astat returns the status of a pending asynchronous file read operation. Upon return from the call, and if the file was opened with the **O_BASYNC** flag, a 1 is returned if the read is complete, and 0 is returned if the read is still pending.

If the file was not opened with the **O_BASYNC** flag, a 1 is returned.

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

[EBADF]	*fildes* is not a valid file descriptor open for reading.
[EINTR]	A signal was caught during the *await*(2) system call.

SEE ALSO

creat(2), *fcntl*(2), *open*(2), *read*(2), *await*(2)

NAME

await – wait for asynchronous read to complete

SYNOPSYS

int await(fildes) int fildes;

DESCRIPTION

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

await suspends execution of the process until no asynchronous file read is pending on the specified file descriptor.

Upon return from the call, and if the file was opened with the **O_BASYNC** flag, the number of bytes actually read is returned if the read is successful. A 0 is returned if end of file was encountered, and a -1 is returned if an error was encountered.

If the file was not opened with the **O_BASYNC** flag, return from the *await* is immediate, and a 1 is returned.

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

[EBADF]	*fildes* is not a valid file descriptor open for reading.
[EINTR]	A signal was caught during the *await* system call.

RETURN VALUE

In asynchronous mode, upon return, a non-negative integer is returned indicating the number of bytes actually read. A 0 is returned for end of file, and a -1 is returned to indicate an error condition, and errno is set to indicate the exact error. SEE ALSO *creat*(2), *fcntl*(2), *open*(2), *read*(2), *astat*(2)

NAME

brk, sbrk – change data segment space allocation

SYNOPSIS

int brk (endds)
char *endds;

char *sbrk (incr)
int incr;

DESCRIPTION

brk and *sbrk* are used to change dynamically the amount of space allocated for the calling process's data segment [see *exec*(2)]. The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases. 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 IO 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 fails and the current working directory is not changed 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 will not delete the program text from the swap area when the last user process terminates. If a 413 executable file has the sticky bit set, the operating system will not delete the program text from memory when the last user process terminates. In either case, if the sticky bit is set the text will already be available (either in a swap area or in memory) when the next user of the file executes it, thus making execution faster.

If the mode bit 02000 (set group ID on execution) is set and the mode bit 00010 (execute or search by group) is not set, mandatory file/record locking will exist on a regular file. This may effect future calls to *open*(2), *creat*(2), *read*(2), and *write*(2) on this file.

chmod will fail and the file mode will be 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

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

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

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

chown fails and the owner and group of the named file are not changed 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

chmod(2), chown(1)

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

path points to a path name naming a directory. *chroot* causes the named directory to become the root directory, the starting point for path searches for path names beginning with /. The 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.

chroot fails and the root directory is not changed 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 ctive.

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

creat – create a new file or rewrite an existing one

SYNOPSIS

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

DESCRIPTION

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

If the file exists, the length is truncated to 0 and the mode and owner are unchanged. Otherwise, the file's owner ID is set to the 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 may have more than 64 files open simultaneously. A new file may be created with a mode that forbids writing.

creat 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.

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 (i.e., both file descriptors share one file pointer).

Same access mode (read, write or read/write).

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

The file descriptor returned is the lowest one available.

dup fails 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

exec in all its forms transforms the calling process into a new process. The new process is constructed from an ordinary, executable file called the *new process file*. This file consists of a header [see *a.out*(4)], a text segment, a data segment, and an optional threadlocal data segment (see *thread*(2)). 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.

Optionally, the *new process file* may be an *interpreter file*. An *interpreter file* begins with a line of the following form:

!#*interpreter*

When an *interpreter file* is *exec*'d, the system *exec*s the specified *interpreter*, giving it the name of the originally *exec*'d file as an argument and shifting over the rest of the optional arguments.

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) and *csh*(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;

and it is used to pass the environment of the calling process to the new process.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set; see *fcntl*(2). For those file descriptors that remain open, the file pointer is unchanged.

Signals set to terminate the calling process will be set to terminate the new process. Signals set to be ignored by the calling process will be set to be ignored by the new process. Signals set to be caught by the calling process will be set to terminate 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 will fail and return 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.
[ENOEXEC]	A multi-threaded process may not *exec*.
[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 NCARGS 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.

SEE ALSO

a.out(4), alarm(2), environ(5), exit(2), fcntl(2), fork(2), lockf(3C), nice(2), ptrace(2), semop(2), sh(1), signal(2), sigset(2), spawn(2), thread(2), times(2), ulimit(2), umask(2)

DIAGNOSTICS

If *exec* returns to the calling process an error has occurred; the return value will be –1 and *errno* will be set to indicate the error.

NAME

exit, _exit – terminate process

SYNOPSIS

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

DESCRIPTION

exit terminates the calling process with the following consequences:

All of the file descriptors open in the calling process are closed.

If the parent process of the calling process is executing a *wait*, it is notified of the calling process's termination and the low order eight bits (i.e., bits 0377) of *status* are made available to it [see *wait*(2)].

If the parent process of the calling process is not executing a *wait*, the calling process is transformed into a zombie process. A *zombie process* is a process that only occupies a slot in the process table. It has no other space allocated either in user or kernel space. The process table slot that it occupies is partially overlaid with time accounting information (see **<sys/proc.h>**) to be used by *times*.

The parent process ID of all of the calling 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, 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.

In a multi-threaded process, a thread that calls *exit* or (*_exit*) causes all other threads of the process to join before exiting. The process itself then effectively calls *exit*.

SEE ALSO

acct(2), intro(2), plock(2), semop(2), signal(2), sigset(2), thread(2), wait(2).

WARNING

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

DIAGNOSTICS

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

NAME

fcntl – file control

SYNOPSIS

#include <fcntl.h>

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

DESCRIPTION

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

The 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 (i.e., both file descriptors share one file pointer). Same access mode (read, write or read/write). Same file status flags (i.e., both file descriptors share the same file status flags). The close-on-exec flag associated with the new file descriptor is set to remain open across *exec*(2) system calls.
F_GETFD	Get the close-on-exec flag associated with the file descriptor *fildes*. If the low-order bit is 0 the file remains open across *exec*; otherwise the file is 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 is set to F_UNLCK.
F_SETLK	Set or clear a file segment lock according to the variable of type *struct flock* pointed to by *arg* [see *fcntl*(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* returns immediately with an error value of –1.
F_SETLKW	This *cmd* is the same as F_SETLK except that if a read or write lock is blocked by other locks, the process sleeps until the segment is free to be locked.
F_GETOWN	Get the process ID or process group currently receiving SIGIO and SIGURG signals; process groups are returned as negative values.

F_SETOWN	Set the process or process groups to receive SIGIO and SIGURG signals; process groups are specified by supplying *arg* as negative, otherwise *arg* is interpreted as a process ID.

F_GETOWN and F_SETOWN apply only to STREAMS devices, not regular files or pipes.

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 always to extend to the end of file by setting *l_len* to zero (0). If such a lock also has *l_whence* and *l_start* set to zero (0), the whole file is locked. Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments for either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process in a *fork*(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.

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

[EBADF]	*fildes* is not a valid open file descriptor.
[EINVAL]	*cmd* is F_DUPFD. *arg* is either negative, or greater than or equal to the configured value for the maximum number of open file descriptors allowed each user.
[EINVAL]	*cmd* is F_GETLK, F_SETLK, or SETLKW and *arg* or the data it points to is not valid.
[EACCES]	*cmd* is F_SETLK the type of lock (*l_type*) is a read (F_RDLCK) lock and the segment of a file to be locked is already write locked by another process or the type is a write (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), signal(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.
F_GETOWN	Process or process group ID of the file owner.

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.

NAME

fork – create a new process

SYNOPSIS

int fork ( )

DESCRIPTION

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

environment
close-on-exec flag [see *exec*(2)]
signal handling settings (i.e., SIG_DFL, SIG_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 (i.e., the process ID of the parent process).

The child process has its own copy of the parent's file descriptors. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.

All semadj values are cleared [see *semop*(2)].

Process locks, text locks and data locks are not inherited by the child [see *plock*(2)].

The child process's *utime, stime, cutime,* and *cstime* are set to 0. The time left until an alarm clock signal is reset to 0.

fork fails and no child process is 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.
[EAGAIN]	A thread attempts to fork.

SEE ALSO

exec(2), nice(2), plock(2), ptrace(2), semop(2), shmop(2), signal(2), sigset(2), thread(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

fildes is a file descriptor obtained from an *open*(2) or *dup*(2) system call.

getdents 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.

getdents 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

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

SYNOPSIS

hostid = gethostid()
long hostid;

sethostid(hostid)
long hostid;

DESCRIPTION

sethostid establishes a 32-bit identifier for the current processor 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.

gethostid returns the 32-bit identifier for the current processor.

SEE ALSO

hostid(1), gethostname(2)

BUGS

32 bits for the identifier is too small.

NAME

getitimer, setitimer – get/set value of interval timer

SYNOPSIS

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

#define ITIMER_REAL       0      /* real time intervals */
#define ITIMER_VIRTUAL    1      /* virtual time intervals */
#define ITIMER_PROF       2      /* user and system virtual time */

getitimer(which, value)
int which;
struct itimerval *value;

setitimer(which, value, ovalue)
int which;
struct itimerval *value, *ovalue;
```

DESCRIPTION

The system provides each process with three interval timers, defined in <*sys/time.h*>. The *getitimer* call returns the current value for the timer specified in *which* in the structure at *value*. The *setitimer* call sets a timer to the specified *value* (returning the previous value of the timer if *ovalue* is nonzero).

A timer value is defined by the *itimerval* structure:

```
struct itimerval {
        struct   timeval it_interval;      /* timer interval */
        struct   timeval it_value;         /* current value */
};
```

If *it_value* is non-zero, it indicates the time to the next timer expiration. If *it_interval* is non-zero, it specifies a value to be used in reloading *it_value* when the timer expires. Setting *it_value* to 0 disables a timer. Setting *it_interval* to 0 causes a timer to be disabled after its next expiration (assuming *it_value* is non-zero).

Time values smaller than the resolution of the system clock are rounded up to this resolution (on the VAX, 10 milliseconds).

The ITIMER_REAL timer decrements in real time. A SIGALRM signal is delivered when this timer expires.

The ITIMER_VIRTUAL timer decrements in process virtual time. It runs only when the process is executing. A SIGVTALRM signal is delivered when it expires.

The ITIMER_PROF timer decrements both in process virtual time and when the system is running on behalf of the process. It is designed to be used by interpreters in statistically profiling the execution of interpreted programs. Each time the ITIMER_PROF timer expires, the SIGPROF signal is delivered. Because this signal may interrupt in-progress system calls, programs using this timer must be prepared to restart interrupted system calls.

NOTES

Three macros for manipulating time values are defined in <*sys/time.h*>. *Timerclear* sets a time value to zero, *timerisset* tests if a time value is non-zero, and *timercmp* compares two time values (beware that >= and <= do not work with this macro).

RETURN VALUE

If the calls succeed, a value of 0 is returned. If an error occurs, the value –1 is returned, and a more precise error code is placed in the global variable *errno*.

ERRORS

The possible errors are:

[EFAULT]	The *value* parameter specified a bad address.
[EINVAL]	A *value* parameter specified a time was too large to be handled.

SEE ALSO

sigvec(2), gettimeofday(2)

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* only processes 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* continues to operate normally, as described above, until the *stream head* read queue is empty. Thereafter, it returns 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).

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 retrieve the remainder of the message.

NAME

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

SYNOPSIS

int getpid ( )

int getpgrp ( )

int getppid ( )

DESCRIPTION

getpid returns the process ID of the calling process.

getpgrp returns the process group ID of the calling process.

getppid returns the parent process ID of the calling process.

SEE ALSO

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

NAME

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

SYNOPSIS

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

ioctl 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).

fildes is an open file descriptor that refers to a device. *Request* selects the control function to be performed and depends on the device being addressed. *arg* represents additional information that is needed by this specific device to perform the requested function. The data type of *arg* depends upon the particular control request, but it is either an integer or a pointer to a device-specific data structure.

In addition to device-specific and STREAMS functions, generic functions are provided by more than one device driver, for example, the general terminal interface [see *termio*(7)].

ioctl 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.

ioctl also fails if the device driver detects an error. In this case, the error is passed through *ioctl* without change to the caller. A particular driver might not exhibit 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 are referred to below as *proc0* and *proc1*, respectively.

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

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

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

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

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

kill fails and no signal is sent if one or more of the following are true:

Sending a signal to a multi-threaded process causes the signal to be sent to one and only one of the threads.

[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

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

path1 points to a path name naming an existing file. *path2* points to a path name naming the new directory entry to be created. *link* creates a new link (directory entry) for the existing file.

link 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* reside 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

symlink(2), 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

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* returns the file pointer even if it is negative.

lseek fails and the file pointer remains 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 routine *mkdir* 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 "." and "..".

mkdir 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

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

0170000 file type; one of the following:

- 0010000 fifo special
- 0020000 character special
- 0040000 directory
- 0060000 block special
- 0100000 or 0000000 ordinary file

0004000 set user ID on execution
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.

mknod may be invoked only by the super-user for file types other than FIFO special.

mknod 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

chmod(2), exec(2), mkdir(1), 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 (Remote File Sharing), the major and minor device numbers are interpreted by the server.

NAME

mount – mount a file system

SYNOPSIS

#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 the MS_FSS flag bit of *mflag* is off, the file system type defaults to the root file system type. If the bit is on, then *fstyp* is used to indicate the file system type. Additionally, if the MS_DATA flag is on in *mflag*, then *dataptr* and *datalen* are used to pass mount parameters to the system. If the MS_DATA flag is off or if either *dataptr* or *datalen*is zero, there is no additional data. In the normal case of a local mount, *dataptr* should be null.

Upon successful completion, references to the file *dir* 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 fails 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]	Pathname 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.

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

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

IPC_STAT	Place the current value of each member of the data structure associated with *msqid* into the structure pointed to by *buf*. The contents of this structure are defined in *intro*(2). {READ}
IPC_SET	Set the value of the following members of the data structure associated with *msqid* to the corresponding value found in the structure pointed to by *buf*: msg_perm.uid msg_perm.gid msg_perm.mode /* only low 9 bits */ msg_qbytes This *cmd* can only be executed by a process that has an effective user ID equal to either that of super user, or to the value of **msg_perm.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 only be executed by a process that has an effective user ID equal to either that of super user, or to the value of **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with *msqid*.

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

[EINVAL]	*msqid* is not a valid message queue 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**. The effective user ID of the calling process is not equal to that of super user, or to the value of **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with *msqid*.
[EPERM]	*cmd* is equal to **IPC_SET**, an attempt is being made to increase to the value of **msg_qbytes,** and the effective user ID of the calling process is not equal to that of super user.
[EFAULT]	*buf* points to an illegal address.

SEE ALSO

intro(2), msgget(2), msgop(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

msgget – get message queue

SYNOPSIS

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

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

DESCRIPTION

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

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

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

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

> **Msg_perm.cuid**, **msg_perm.uid**, **msg_perm.cgid**, and **msg_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.
>
> The low-order 9 bits of **msg_perm.mode** are set equal to the low-order 9 bits of *msgflg*.
>
> **Msg_qnum**, **msg_lspid**, **msg_lrpid**, **msg_stime**, and **msg_rtime** are set equal to 0.
>
> **Msg_ctime** is set equal to the current time.
>
> **msg_qbytes** is set equal to the system limit.

msgget 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

msgsnd is used to send a message to the queue associated with the message queue identifier specified by *msqid*. {WRITE} *msgp* points to a structure containing the message. This structure is composed of the following members:

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

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

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

The number of bytes already on the queue is equal to **msg_qbytes** [see *intro*(2)].

The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

If (*msgflg* & IPC_NOWAIT) is "true", the message 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).

msgsnd 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 actions are specified as follows:

If (*msgflg* & **IPC_NOWAIT**) is "true", the calling process will return immediately with a return value of –1 and *errno* set to ENOMSG.

If (*msgflg* & **IPC_NOWAIT**) is "false", the calling process will suspend execution until one of the following occurs:

A message of the desired type is placed on the queue.

msqid is removed from the system. When this occurs, *errno* is set equal to EIDRM, and a value of –1 is returned.

The calling process receives a signal that is to be caught. In this case a message is not received and the calling process resumes execution in the manner prescribed in *signal*(2).

msgrcv 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 set 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

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.

A maximum nice value of 39 and a minimum nice value of 0 are imposed by the system. (The default nice value is 20.) Requests for values above or below these limits result in the nice value being set to the corresponding limit.

[EPERM]	*nice* fails and does not change the nice value if *incr* is negative or greater than 39 and the effective user ID of the calling process is not super-user.

SEE ALSO

exec(2), nice(1)

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

open – open for reading or writing

SYNOPSIS

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

DESCRIPTION

path points to a path name naming a file. *open* opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*. 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 returns without delay. An *open* for writing-only returns an error if no process currently opens the file for reading.

If O_NDELAY is clear:

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

When opening a file associated with a communication line:

If O_NDELAY is set:

The *open* returns without waiting for carrier.

If O_NDELAY is clear:

The *open* blocks until carrier is present.

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

O_SYNC When opening a regular file, this flag affects subsequent writes. If set, each *write*(2) waits 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* fails 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 opened the file 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.

SEE ALSO

chmod(2), close(2), creat(2), dup(2), fcntl(2), intro(2), lseek(2), read(2), getmsg(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

pause suspends the calling process until it receives a signal. The signal must be one that is not currently set to be ignored by the calling process.

If the signal causes termination of the calling process, *pause* 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), sigpause(2), wait(2).

NAME

phys – allow a process to access physical addresses

DESCRIPTION

The argument vaddr specifies a process virtual address and the argument paddr specifies a processor physical address. The range [vaddr, vaddr + nbytes - 1] is rounded up to an integral number of pages and the backing store for these virtual pages is replaced by the set of real pages specified by the range [paddr + paddr + nbytes - 1]. The addresses vaddr and paddr must both specify the same offset within a page. That is, vaddr modulo pagesize must equal paddr modulo pagesize.

The virtual pages defined by the parameters must be a set of already valid mappings in the process address space. Use brk(2) or sbrk(2) to ensure this.

This call may only be executed by the super-user.

RETURN VALUE

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

ERRORS

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

[EPERM]	The caller is not the super-user.
[EINVAL]	No current valid mapping exists for the range specified by the virtual address vaddr and nbytes argument.
[EINVAL]	vaddr modulo pagesize does not equal paddr modulo pagesize.

SEE ALSO

brk(2), sbrk(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 5120 bytes of data are buffered by the pipe before the writing process is blocked. A read only file descriptor *fildes*[0] accesses the data written to *fildes*[1] on a first-in-first-out (FIFO) basis.

pipe fails if:

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

SEE ALSO

read(2), sh(1), 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 segment (data lock), or both its text and data segments (process lock) into memory. Locked segments are immune to all routine swapping. *plock* also allows these segments to be unlocked. The effective user ID of the calling process must be super-user to use this call. *Op* specifies the following:

PROCLOCK –	lock text and data segments into memory (process lock)
TXTLOCK –	lock text segment into memory (text lock)
DATLOCK –	lock data segment into memory (data lock)
UNLOCK –	remove locks

plock fails and does 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* fails.

If the value *fd* is less than zero, *events* is ignored and *revents* is set to 0 in that entry on return from *poll*.

The results of the *poll* query are stored in the *revents* field in the *pollfd* structure. Bits are set in the *revents* bitmask to indicate which of the requested events are true. If none is 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 has 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), streamio(7), write(2).

DIAGNOSTICS

Upon successful completion, a non-negative value is returned. A positive value indicates the total number of file descriptors that has been selected (i.e., 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 is 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. 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 occur if the parent does not expect to trace the child.

The remainder of the requests can only be used by the parent process. For each, *pid* is the process ID of the child. The child must be in a stopped state before these requests are made.

1, 2 With these requests, the word at location *addr* in the address space of the child is returned to the parent process. Either request **1** or request **2** may be used with equal results. The *data* argument is ignored. These two requests 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. The *data* argument is ignored. This request fails 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*. Either request 4 or request 5 may be used with equal results. Upon successful completion, the value written into the address space of the child is returned to the parent. These two requests fail if *addr* is 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 scalar floating point status and registers, and certain bits of the processor status word. The old value at the address is returned. The few entries that can be written are the general registers and the 32 general registers.

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 a breakpoint in the text of the child and then executes the same steps as listed above for request 7. The breakpoint causes an interrupt upon completion of one machine instruction. This effectively allows single stepping of the child. This breakpoint is invisible to both the parent and 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 stops before executing the first instruction of the new image showing signal SIGTRAP.

General Errors

ptrace fails in general 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

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

putmsg creates a message [see *intro*(2)] from user specified buffer(s) and sends the message to a STREAMS file. The message may contain either a data part, a control part or both. The data and control parts to be sent are distinguished by placement in separate buffers, as described below. The semantics of each part is defined by the STREAMS module that receives the message.

fd specifies a file descriptor referencing an open *stream*. *ctlptr* and *dataptr* each point to a *strbuf* structure which contains the following members:

```
int maxlen;      /* not used */
int len;  /* length of data */
char *buf;       /* ptr to buffer */
```

ctlptr points to the structure describing the control part, if any, to be included in the message. The *buf* field in the *strbuf* structure points to the buffer where the control information resides, and the *len* field indicates the number of bytes to be sent. The *maxlen* field is not used in *putmsg* [see *getmsg*(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 is sent if either *dataptr* (*ctlptr*) is NULL or the *len* field of *dataptr* (*ctlptr*) is set to –1.

If a control part is specified, and *flags* is set to RS_HIPRI, a *priority* message is sent. If *flags* is set to 0, a non-priority message is sent. If no control part is specified, and *flags* is set to RS_HIPRI, *putmsg* fails and sets *errno* to EINVAL. If no control part and no data part are specified, and *flags* is set to 0, no message is sent, and 0 is returned.

For non-priority messages, *putmsg* blocks if the *stream* write queue is full due to internal flow control conditions. For priority messages, *putmsg* does not block on this condition. For non-priority messages, *putmsg* does not block when the write queue is full and O_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)

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

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

read attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*.

On devices capable of seeking, the *read* starts at a position in the file given by the file pointer associated with *fildes*. Upon return from *read*, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a file is undefined.

Upon successful completion, *read* returns the number of bytes actually read and placed in the buffer; this number may be less than *nbyte* if the file is associated with a communication line [see *ioctl*(2) and *termio*(7)], or if the number of bytes left in the file is less than *nbyte* bytes. A value of 0 is returned when an end-of-file has been reached. 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 messsage boundaries.

In STREAMS message-nondiscard mode, *read* retrieves data until it has read *nbyte* bytes, or 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 (i.e. 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 sets 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 a 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 sets 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. *read* 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*.

read 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 I/O 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.

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), 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

readlink – read value of a symbolic link

SYNOPSIS

cc = readlink(path, buf, bufsiz)
int cc;
char *path, *buf;
int bufsiz;

DESCRIPTION

readlink places the contents of the symbolic link *name* in the buffer *buf,* which has size *bufsiz*. The contents of the link are not null terminated when returned.

RETURN VALUE

The call returns the count of characters placed in the buffer if it succeeds, or a –1 if an error occurs, placing the error code in the global variable *errno*.

ERRORS

readlink fails and the file mode is not changed if:

[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.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EINVAL]	The named file is not a symbolic link.
[EIO]	An I/O error occurred while reading from the file system.
[EFAULT]	*buf* extends outside the process's allocated address space.

SEE ALSO

stat(2), lstat(2), symlink(2)

NAME

rename – change the name of a file

SYNOPSIS

rename(from, to)
char *from, *to;

DESCRIPTION

rename causes the link named *from* to be renamed as *to*. If *to* exists, then it is first removed. Both *from* and *to* must be of the same type (that is, both directories or both non-directories), and must reside on the same file system.

rename guarantees that an instance of *to* will always exist, even if the system should crash in the middle of the operation.

If the final component of *from* is a symbolic link, the symbolic link is renamed, not the file or directory to which it points.

CAVEAT

The system can deadlock if a loop in the file system graph is present. This loop takes the form of an entry in directory "a", say "a/foo", being a hard link to directory "b", and an entry in directory "b", say "b/bar", being a hard link to directory "a". When such a loop exists and two separate processes attempt to perform "rename a/foo b/bar" and "rename b/bar a/foo", respectively, the system may deadlock attempting to lock both directories for modification. Hard links to directories should be replaced by symbolic links by the system administrator.

RETURN VALUE

A 0 value is returned if the operation succeeds, otherwise *rename* returns –1 and the global variable *errno* indicates the reason for the failure.

ERRORS

rename fails and neither of the argument files is affected if any of the following are true:

[ENOENT]	A component of the *from* path does not exist, or a path prefix of FIto *does not exist.*
[EACCES]	A component of either path prefix denies search permission.
[EACCES]	The requested link requires writing in a directory with a mode that denies write permission.
[EPERM]	The directory containing *from* is marked sticky, and neither the containing directory nor *from* are owned by the effective user ID.
[EPERM]	The *to* file exists, the directory containing *to* is marked sticky, and neither the containing directory nor *to* are owned by the effective user ID.
[ELOOP]	Too many symbolic links were encountered in translating either pathname.
[ENOTDIR]	A component of either path prefix is not a directory.
[ENOTDIR]	*from* is a directory, but *to* is not a directory.
[EISDIR]	*to* is a directory, but *from* is not a directory.
[EXDEV]	The link named by *to* and the file named by *from* are on different logical devices (file systems). Note that this error code will not be returned if the implementation permits cross-device links.

[ENOSPC]	The directory in which the entry for the new name is being placed cannot be extended because there is no space left on the file system containing the directory.
[EDQUOT]	The directory in which the entry for the new name is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.
[EIO]	An I/O error occurred while making or updating a directory entry.
[EROFS]	The requested link requires writing in a directory on a read-only file system.
[EFAULT]	*path* points outside the process's allocated address space.
[EINVAL]	*from* is a parent directory of *to*, or an attempt is made to rename "." or "..".
[ENOTEMPTY]	*to* is a directory and is not empty.

SEE ALSO

open(2)

NAME

rmdir – remove a directory

SYNOPSIS

int rmdir (path)
char *path;

DESCRIPTION

rmdir removes the directory named by the path name pointed to by *path*. The directory must not have any entries other than "." and "..".

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

[EINVAL]	The current directory may not be removed.
[EINVAL]	The "." entry of a directory may not be removed.
[EEXIST]	The directory contains entries other than those for "." and "..".
[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.
[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.

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

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

NAME

semctl – semaphore control operations

SYNOPSIS

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

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

DESCRIPTION

semctl provides a variety of semaphore control operations as specified by *cmd*.

The following *cmd*s are executed with respect to the semaphore specified by *semid* and *semnum:*

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

The following *cmd*s return and set, respectively, every semval in the set of semaphores.

GETALL	Place semvals into array pointed to by *arg.array*. {READ}
SETALL	Set semvals according to the array pointed to by *arg.array*. {ALTER} When this cmd is successfully executed the semadj values corresponding to each specified semaphore in all processes are cleared.

The following *cmd*s are also available:

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

IPC_SET Set the value of the following members of the data structure associated with *semid* to the corresponding value found in the structure pointed to by *arg.buf*:
sem_perm.uid
sem_perm.gid
sem_perm.mode /* only low 9 bits */

This cmd can only be executed by a process that has an effective user ID equal to either that of super-user, or to the value of **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with *semid*.

IPC_RMID	Remove the semaphore identifier specified by *semid* from the system and destroy the set of semaphores and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of super-user, or to the value of **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with *semid*.

semctl 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

semget returns the semaphore identifier associated with *key*.

A semaphore identifier and associated data structure and set containing *nsems* semaphores [see *intro*(2)] are created for *key* if one of the following is true:

> *key* is equal to IPC_PRIVATE.
>
> *key* does not already have a semaphore identifier associated with it, and (*semflg* & IPC_CREAT) is "true".

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

> **sem_perm.cuid**, **sem_perm.uid**, **sem_perm.cgid**, and **sem_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.
>
> The low-order 9 bits of **sem_perm.mode** are set equal to the low-order 9 bits of *semflg*.
>
> **sem_nsems** is set equal to the value of *nsems*.
>
> **sem_otime** is set equal to 0 and **sem_ctime** is set equal to the current time.

semget 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

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

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

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

sem_op specifies one of three semaphore operations as follows:

If *sem_op* is a negative integer, one of the following 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 suspend execution of the calling process until one of the following conditions occurs:

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

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

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

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

If *sem_op* is zero, one of the following 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 suspend execution of the calling process until one of the following occurs:

Semval becomes zero, at which time the value of semzcnt associated with the specified semaphore is decremented.

The semid for which the calling process is awaiting action is removed from the system. When this occurs, *errno* is set equal to EIDRM, and a value of –1 is returned.

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

semop 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

setpgrp – set process group ID

SYNOPSIS

int setpgrp ( )

DESCRIPTION

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

SEE ALSO

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

DIAGNOSTICS

setpgrp returns the value of the new process group ID.

NAME

setppri – set process priority

SYNOPSIS

int setppri (pid, pri)
int pid, pri;

DESCRIPTION

setppri sets the priority of the specified process to the indicated priority.

Setting the process priority to a positive value has little effect since the normal system priority will quickly override whatever positive priority is set. Setting the process priority to a negative value makes the process into a real-time process.

Real-time processes are scheduled differently from normal processes. Real-time processes never have their priority altered by the kernel. Real-time processes are scheduled before any non real-time process and execute until either they voluntarily relinquish the CPU, another real-time process of higher (smaller numeric) priority becomes ready to run, or the process expires its time slice and another real-time process of the same priority is ready to run.

The process nice value has no effect on real-time processes and the nice value does not affect real-time process priority values.

Real-time processes can be quite dangerous, since a real-time process in an infinite loop cannot be interrupted by any other process (other than a real-time process of higher priority).

setppri may be invoked only by the super-user.

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

[EINVAL]	The *pid* does not correspond to any current process in the system.
[EINVAL]	The *pri* value specified is not between -128 and +127, inclusive.

SEE ALSO

nice(2)

DIAGNOSTICS

setppri returns zero on successful execution. Otherwise a value of -1 is returned and *errno* is set to indicate the error.

NAME

setuid, setgid – set user and group IDs

SYNOPSIS

int setuid (uid)
int uid;

int setgid (gid)
int gid;

DESCRIPTION

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

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

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

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*).

setuid (*setgid*) fails if either of the following conditions is true:

[EPERM]
: If the real user (group) ID of the calling process is not equal to *uid* (*gid*) and its effective user ID is not super-user.

[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

shmctl provides a variety of shared memory control operations as specified by *cmd*. The following *cmd*s are available:

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

IPC_SET Set the value of the following members of the data structure associated with *shmid* to the corresponding value found in the structure pointed to by *buf*: shm_perm.uid
shm_perm.gid
shm_perm.mode /* only low 9 bits */

This *cmd* can only be executed by a process that has an effective user ID equal to 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 only be executed by a process that has an effective user ID equal to that of super user, or to the value of **shm_perm.cuid** or **shm_perm.uid** in the data structure associated with *shmid*.

SHM_LOCK Lock the shared memory segment specified by *shmid* in memory. This *cmd* can only be executed by a process that has an effective user ID equal to super user.

SHM_UNLOCK
Unlock the shared memory segment specified by *shmid*. This *cmd* can only be executed by a process that has an effective user ID equal to super user.

shmctl 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

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

shmget 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.cgid**, and **shm_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.
>
> The low-order 9 bits of **shm_perm.mode** are set equal to the low-order 9 bits of *shmflg*. **shm_segsz** is set equal to the value of *size*.
>
> **shm_lpid**, **shm_nattch**, **shm_atime**, and **shm_dtime** are set equal to 0.
>
> **shm_ctime** is set equal to the current time.

shmget 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 9 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 fails and does 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* fails and does not detach the shared memory segment if *shmaddr* is not the data segment start address of a shared memory segment.

SEE ALSO

exec(2), exit(2), fork(2), intro(2), shmctl(2), shmget(2).

DIAGNOSTICS

Upon successful completion, the return value is as follows:

shmat returns the data segment start address of the attached shared memory segment.

shmdt returns a value of 0.

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

NOTES

The user must explicitly remove shared memory segments after the last reference to them has been removed.

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 (rubout)
SIGQUIT	03 [1]	quit (ASCII FS)
SIGILL	04 [1]	illegal instruction (not reset when caught)
SIGTRAP	05 [1]	trace trap (not reset when caught)
SIGIOT	06 [1]	IOT instruction (obsolete)
SIGABRT	06 [1]	used by **abort**, replaces SIGIOT
SIGEMT	07 [1]	EMT instruction (obsolete)
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 restart
SIGWIND	20	window change
SIGURG	21	urgent condition on an I/O channel
SIGPOLL	22 [3]	selectable event pending
SIGSTOP	23	sendable stop signal, not from tty
SIGTSTP	24	stop signal from tty
SIGTTIN	25	process stop by background tty read
SIGTTOU	26	process stop by background tty write
SIGCONT	27	continue a stopped process
SIGXCPU	28	exceeded CPU time limit
SIGXFSZ	29	exceeded file size limit
SIGVTALRM	30	virtual time alarm
SIGPROF	31	profiling time alarm

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 listed below:

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: 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* is passed as the first argument to the signal-catching function. Additional arguments are passed to the signal-catching function for hardware-generated signals as described in Note [4], below. Before entering the signal-catching function, the value of *func* for the caught signal is set to SIG_DFL unless the signal is SIGKILL, SIGTRAP, or SIGPWR.

Upon return from the signal-catching function, the receiving process resumes execution at the point it was interrupted.

When a signal that is to be caught occurs during a *read*(2), *write*(2), *open*(2), or an *ioctl*(2) system call on a slow device (like a terminal; but not a file), during a *pause*(2) or *wait*(2) that does not return immediately due to the existence of a previously stopped or zombie process, the signal catching function is executed and then the interrupted system call may return a –1 to the calling process with *errno* set to EINTR.

signal does not catch an invalid function argument, *func*, and results are undefined when an attempt is made to execute the function at the bad address.

Note: The signal SIGKILL cannot be caught.

A call to *signal* cancels a pending signal *sig* except for a pending SIGKILL signal.

signal fails 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" is constructed in the current working directory of the process, if the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named **core** exists and is writable or can be created. If the file must be created, it inherits 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 does 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 are 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* blocks until all of the calling process's child processes terminate; it then returns 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 does 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 never receives **SIGPOLL**.

[4] The handler routine to catch signals can be delared as follows:

```
handler(sig, code, junk, context)
int sig, code, junk
struct sigcontext *context;
```

Here, *sig* is the signal number. *code* is a value which further interprets *sig*; it may be one of the following:

SIGFPE
  0: integer overflow
  1: floating exception

SIGTRAP
  CAUSESINGLE: single step
  CAUSEBREAK: breakpoint instruction

The value of *junk* is the address of the handler routine itself. *context* is a pointer to the machine state at the time of the exception. It is defined in */usr/include/sys/signal.h*.

For the Stardent 1500: On floating point exceptions, the FPU is halted. Explicit user action of restarting the FPU will be necessary. For the Stardent 3000: On floating point exceptions, the FPU state is preserved in the *context* structure, and the FPU state is made idle and ready for use.

SEE ALSO

intro(2), kill(1), 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	01	hangup
SIGINT	02	interrupt (rubout)
SIGQUIT	03 *	quit (ASCII FS)
SIGILL	04 *	illegal instruction (not reset when caught)
SIGTRAP	05 *	trace trap (not reset when caught)
SIGIOT	06 *	IOT instruction (obsolete)
SIGABRT	06 *	used by **abort**, replaces SIGIOT
SIGEMT	07 *	EMT instruction (obsolete)
SIGFPE	08 *	floating point exception
SIGKILL	09	kill (cannot be caught or ignored)
SIGBUS	10 *	bus error
SIGSEGV	11 *	segmentation violation
SIGSYS	12 *	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user-defined signal 1
SIGUSR2	17	user-defined signal 2
SIGCLD	18	death of a child
SIGPWR	19	power fail restart
SIGWIND	20	window change
SIGURG	21	urgent condition on an I/O channel
SIGPOLL	22	selectable event pending
SIGSTOP	23	sendable stop signal, not from tty
SIGTSTP	24	stop signal from tty
SIGTTIN	25	process stop by background tty read
SIGTTOU	26	process stop by background tty write

SIGCONT	27	continue a stopped process
SIGXCPU	28	exceeded CPU time limit
SIGXFSZ	29	exceeded file size limit
SIGVTALRM	30	virtual time alarm
SIGPROF	31	profiling time alarm

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 contains 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" is 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 inherits 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 calls this function upon receipt of signal *sig*. Any pending signal of this type is released. This handler address is retained across calls to the other signal management functions listed here.

When a signal occurs, the signal number *sig* is passed as the only argument to the signal-catching handler. Before calling the signal-catching handler, the system signal action is set to SIG_HOLD . During normal return from the signal-catching handler, the system signal action is restored to *func* and any held signal of this type released. If a non-local goto (*longjmp*) is taken, then *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 resumes execution at the point it was interrupted. However, when a signal is caught during a *read*(2), *write*(2), *open*(2), or an *ioctl*(2) system call during a *sigpause* or *wait*(2) that does not return immediately due to the existence of a previously stopped or zombie process, the signal-catching handler is executed and then the interrupted system call may return a –1 to the calling process with *errno* set to EINTR.

sighold and *sigrelse* 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 fails 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*.

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.

SEE ALSO

kill(2), pause(2), signal(2), wait(2), setjmp(3C).

WARNING

Two signals behave differently from the signals described above:

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 listed below:

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 do 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 is 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* blocks until all of the calling process's child processes terminate; it then returns 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 does 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.

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 never receives **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.

NAME

spawn: spawnl, spawnv, spawnle, spawnve, spawnlp, spawnvp – fork and execute a file

SYNOPSIS

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

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

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

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

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

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

DESCRIPTION

spawn in all its forms forks the calling process and transforms the child process into a new process. It is almost exactly equivalent to a *fork* immediately followed by an *exec*.

The primary difference between a *fork* followed by an *exec* and a *spawn* is that a *spawn* does not duplicate the memory regions of the parent process. This makes *spawn* a bit faster than a *fork* followed by an *exec*. It also allows large processes to spawn sub-processes without a need for the sub-process to reserve all of the virtual memory resources needed on a *fork*.

The child 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, a data segment, and an optional threadlocal data segment (see *thread*(2)). The data segment contains an initialized portion and an uninitialized portion (bss).

Optionally, the *new process file* may be an *interpreter file*. An *interpreter file* begins with a line of the form "#! *interpreter*". When an *interpreter file* is *spawn'd* the system *execs* the specified *interpreter*, giving it the name of the originally *spawn'd* file as an argument and shifting over the rest of the optional arguments.

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 *spawnl* and *spawnv*, the C run-time start-off routine places a pointer to the environment of the calling process in the global cell:

extern char **environ;

and it is used to pass the environment of the calling process to the new process.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set; see *fcntl*(2). For those file descriptors that remain open, the child process retains 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.

Signals set to terminate the calling process will be set to terminate the new process. Signals set to be ignored by the calling process will be set to be ignored by the new process. Signals set to be caught by the calling process will be set to terminate new process; see *signal*(2).

For signals set by *sigset*(2), *spawn* 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)], *spawn* 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 inherits the following attributes from the calling process:

nice value [see *nice*(2)]
process group ID
tty group ID [see *exit*(2) and *signal*(2)]
trace flag [see *ptrace*(2) request 0]
current working directory
root directory
file mode creation mask [see *umask*(2)]
file size limit [see *ulimit*(2)]
file-locks [see *fcntl*(2) and *lockf*(3C)]

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

The child process has a unique process ID.

The child process has a different parent process ID (i.e., the process ID of the parent process).

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.

spawn will fail and return 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 spawn is not an *spawnlp* or *spawnvp*, and the new process file has the appropriate access permission but an invalid magic number in its header.
[ENOEXEC]	A multi-threaded process may not *spawn*.
[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 NCARGS 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 *spawn*(2) a shared library directly.
[EINTR]	A signal was caught during the *spawn* system call.
[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

a.out(4), alarm(2), environ(5), exec(2), exit(2), fcntl(2), fork(2), lockf(3C), nice(2), ptrace(2), semop(2), sh(1), signal(2), sigset(2), thread(2), times(2), ulimit(2), umask(2)

DIAGNOSTICS

Upon successful completion, *spawn* begins execution of 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

stat, lstat, fstat – get file status

SYNOPSIS

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

int stat (path, buf)
char *path;
struct stat *buf;

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)].

lstat is like *stat* except when the named file is a symbolic link, in which case *lstat* returns information about the link, while *stat* returns information about the file the link references.

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:

```
ino_t    st_ino;     /* Inode number */
ushort   st_mode;    /* File mode [see mknod(2)] */
dev_t    st_dev;     /* ID of device containing */
                     /* a directory entry for this file */
dev_t    st_rdev;    /* ID of device */
                     /* This entry is 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_ino This field uniquely identifies the file in a given file system. The pair st_ino and st_dev uniquely identifies regular files.

st_mode	The mode of the file as described in the *mknod*(2) system call.
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 and *lstat* 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 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), 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

statfs returns a "generic superblock" describing a file system. It can be used to acquire information about mounted as well as unmounted file systems, and usage is slightly different in the two cases. In all cases, *buf* is a pointer to a structure (described below) which is filled by the system call, and *len* is the number of bytes of information which the system should return in the structure. *len* must be no greater than **sizeof (struct statfs)** and ordinarily it contains exactly that value; if it holds a smaller value the system fills the structure with that number of bytes. (This allows future versions of the system to enlarge 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 */
```

fstatfs is similar, except that the file named by *path* in *statfs* is instead identified by an open file descriptor *filedes* obtained from a successful *open*(2), *creat*(2), *dup*(2), *fcntl*(2), or *pipe*(2) system call.

statfs obsoletes *ustat*(2) and should be used in preference to it in new programs.

statfs and *fstatfs* fails 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* fails 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

symlink – make symbolic link to a file

SYNOPSIS

symlink(name1, name2)
char *name1, *name2;

DESCRIPTION

A symbolic link *name2* is created to *name1* (*name2* is the name of the file created, *name1* is the string used in creating the symbolic link). Either name may be an arbitrary path name; the files need not be on the same file system.

RETURN VALUE

Upon successful completion, a zero value is returned. If an error occurs, the error code is stored in *errno* and a –1 value is returned.

ERRORS

The symbolic link is made unless on or more of the following are true:

[ENOTDIR]	A component of the *name2* prefix is not a directory.
[ENAMETOOLONG]	A component of either pathname exceeded 255 characters, or the entire length of either path name exceeded 1023 characters.
[ENOENT]	The named file does not exist.
[EACCES]	A component of the *name2* path prefix denies search permission.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EEXIST]	*Name2* already exists.
[EIO]	An I/O error occurred while making the directory entry for *name2*, or allocating the inode for *name2*, or writing out the link contents of *name2*.
[EROFS]	The file *name2* would reside on a read-only file system.
[ENOSPC]	The directory in which the entry for the new symbolic link is being placed cannot be extended because there is no space left on the file system containing the directory.
[ENOSPC]	The new symbolic link cannot be created because there there is no space left on the file system that will contain the symbolic link.
[ENOSPC]	There are no free inodes on the file system on which the symbolic link is being created.
[EDQUOT]	The user's quota of inodes on the file system on which the symbolic link is being created has been exhausted.
[EIO]	An I/O error occurred while making the directory entry or allocating the inode.
[EFAULT]	*name1* or *name2* points outside the process's allocated address space.

SEE ALSO

link(2), ln(1), unlink(2)

NAME

sync – update super block

SYNOPSIS

void sync ( )

DESCRIPTION

sync causes all information in memory that should be on disk to be written out. This includes modified super blocks, modified i-nodes, and delayed block I/O.

It should be used by programs which examine a file system, for example *fsck*, *df*, etc. It is mandatory before a re-boot.

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 fails 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* points 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

sysmips – machine specific functions

SYNOPSIS

#include <sys/sysmips.h>

int sysmips (cmd, arg1, arg2, arg3)

int cmd, arg1, arg2, arg3;

DESCRIPTION

sysmips implements machine specific functions. The *cmd* argument determines the function performed. The number of arguments expected is dependent on the function.

Command SETNAME

When *cmd* is SETNAME, the name of the system may be changed. One argument, the address of a string containing the new name of the system, is expected. This name is copied into the kernel *utsname* data structure, into both the *sysname* and *nodename* fields. It is also copied into *hostname* so that future calls to *hostname* (2-BSD) will return this value. You must be the super-user to use this function.

Command FLUSH_CACHE

This function flushes both data and instruction caches on all processors. You need not be super-user to use this function.

Command FLUSH_ICACHE

This function flushes the instruction cache on all processors. You need not be super-user to use this function.

Command ENABLE_CPU

This function allows the processor specified by the first argument to dispatch processes (processors are all normally enabled). Must be super-user to use this function.

Command DISABLE_CPU

This function prevents the processor specified from dispatching. There is big time trouble if the last processor in the system! You must be super-user to use this function.

Command SLED_ON

This function turns on the front panel yellow light emitting diode (LED). Must be super-user to use this function.

Command SLED_OFF

This function turns off the front panel yellow light emitting diode (LED). Must be super-user to use this function.

Command SLED_FLASH

This function causes front panel yellow light emitting diode (LED) to flash approximately once a second. Must be super-user to use this function.

Command SREAD_NVRAM

This function reads the contents of the non-volatile memory which is maintained on the system I/O board. Transfers the contents into the byte array pointed to by the first argument. There are 2048 bytes of nvram, organized as a list of null delimited strings of the form <nvram name>=<nvram value>. Need not be super-user to use this function.

Command SWRITE_NVRAM

This function expects the first argument to be a pointer to a string of the form <nvram name>=<nvram value>. The nvram string identified by <nvram name> is replaced by <nvram value>. If <nvram value> is null, then <nvram name> is removed from nvram. Must be super-user to use this function.

Command SREAD_IDPROM

This function transfers the contents of the ID PROM on each of the cpu boards present in the system into the byte array pointed to by the first argument. Each cpu board maintains 32 bytes of ID PROM information and there is a limit of 4 cpu boards per system. The specified array must be at least 128 bytes long. Need not be super-user to use this function.

Command SBOOT_CPU

This function returns the cpu id of the boot processor. Need not be super-user to use this function.

Command SFORCE_RUN

This function forces the calling process (or thread of a multi-threaded process) to always execute on the cpu specified in the first argument. For instance, *sysmips*(SFORCE_RUN, 2) would force a process to always run on cpu 2. Need not be super-user to use this function.

Command SFORCE_NORUN

This function forces the calling process (or thread of a multi-threaded process) to never execute on the cpu specified in the first argument. For instance, *sysmips*(SFORCE_NORUN, 2) would force a process off of cpu 2 (if it were currently executing on cpu 2) and would not schedule the process on cpu 2 in the future. Need not be super-user to use this function.

Command WHICHPROC

This function returns the processor id of the cpu that is executing the calling process. Processor id's are 0, 1, 2, etc.

Command PROCPRESENT

This function returns a bit mask indicating which processors are present in the system and which are currently enabled. Each bit corresponds to a processor id. For instance, if processor id 0 is present and enabled, bit 0 of the return value is set, if processor id 1 is present and enabled, bit 1 of the return value is set, etc.

Command SMIPSDENORM

Denormalized floating point numbers are handled differently by the Mips 3010 FPU and the Stardent 1500/3000 vector FPU. The vector FPU replaces "denorms" by zero while the Mips FPU causes a trap. Setting *arg1* to *1* causes the Mips FPU to treat "denorms" the same way as the vector FPU. Setting *arg1* to zero returns to the default Mips behavior.

Command SMIPSSWPI

When *cmd* is SMIPSSWPI, individual swapping areas may be added, deleted or the current areas determined. The address of an appropriately primed swap buffer is passed as the only argument. (Refer to *sys/swap.h* header file for details of loading the buffer.)

The format of the swap buffer is:

```
struct swapint {
  char si_cmd;         /*command: list, add, delete*/
  char *si_buf; /*swap file path pointer*/
  int  si_swplo;       /*start block*/
  int  si_nblks;       /*swap size*/
}
```

Note that the add and delete options of the command may only be exercised by the super-user.

Typically, a swap area is added by a single call to *sysmips*. First, the swap buffer is primed with appropriate entries for the structure members. Then *sysmips* is invoked.

```
#include <sys/sysmips.h>
#include <sys/swap.h>

struct swapint swapbuf;          /*swap into buffer ptr*/

sysmips(SMIPSSWPI, &swapbuf);
```

If this command succeeds, it returns 0 to the calling process. This command fails, returning -1, if one or more of the following is true:

[EFAULT]	*Swapbuf* points to an invalid address
[EFAULT]	*Swapbuf.si_buf* points to an invalid address
[ENOTBLK]	Swap area specified is not a block special device
[EEXIST]	Swap area specified has already been added
[ENOSPC]	Too many swap areas in use (if adding)
[ENOMEM]	Tried to delete last remaining swap area
[EINVAL]	Bad arguments
[EINVAL]	Specified a non-existent cpu
[ENOMEM]	No place to put swapped pages when deleting a swap area

Command STIME

When *cmd* is STIME, an argument of type long is expected. This function sets the system time and date. The argument contains the time as measured in seconds from 00:00:00 GMT January 1, 1970. Note that this command is only available to the super-user.

SEE ALSO

sync(2), a.out(4).

swap(1M) in the *System* Administrator's*Reference*Manual.

DIAGNOSTICS

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

SBOOT_CPU: returns the cpu id of the boot processor

WHICHPROC: returns the cpu id of the current processor

PROCPRESENT: returns a bit mask of present and enabled processors

all others: return 0

Otherwise, a value of –1 is returned and *errno* is set to indicate the error. When *cmd* is invalid, *errno* is set to EINVAL on return.

NAME

thread – create, terminate, suspend, or resume threads in a process

SYNTAX

int thread(enum thread_functions, int);

DESCRIPTION

thread is used to control the allocation and execution of threads. The *thread_function* argument describes the desired action. *int* is mandatory and its value and its use are dependent upon the specified *thread_function* argument. The *thread_function* arguments are:

THREAD_CREATE	Creates a number of threads, chosen by the system for best performance. Typically, this number is equal to the number of processors available. *int* argument is ignored. The return value for each thread is the number created. Returns –1 on an error.
THREAD_CREATEN	Creates *int* number of arguments. Typically, this number is equal to the number of processors available. The return value for each thread is the number created. Returns –1 on an error.
THREAD_JOIN	Forces all threads to be terminated and the parent revives. *int* argument is ignored. Returns zero on completion.
THREAD_SUSPEND	Halts all peer threads, but preserves their state. Leaves only one thread running. *int* argument is ignored.
THREAD_RESUME	Resumes execution of halted peer threads. *int* argument is ignored.
THREAD_NPROC	Returns as the value of the function the number of processors available. This is the number of processors that would be assigned with use of THREAD_CREATE. *int* argument is ignored.

Associated Functions

The following functions are used in the manipulation of threads.

void threadserial()	Forces all other threads to become idle. After the function is executed, the process is serial.
thread_new(p,n,TYPE)	A macro that assigns to *p*, a pointer, new storage sufficient to hold *n* objects of type *TYPE*.
thread_free(p)	A macro that frees storage allocated by *thread_new*. *p* is a pointer to the storage.
void parbegin(n)	Marks the beginning of a parallel section of code that splits into *n* microtasks. This can also be used, with low overhead, to redispatch idled threads.
int parnext()	Initiates a microtask in the parallel section of code. It returns the microtask number from *n* to *1* When there are no further microtasks, *parnext* idles all threads but one. When all other threads complete their tasks, *parnext* returns zero.

void parstack(void (*)(), int, int)
: The first argument is a *function*, the second argument is a *stack_size*, and the third argument is some *number* of threads. *parstack* requests that some *number* of available threads each independently execute a call to a *function*. Each call to the *function* is done on a separate stack of size *stack_size*. If *number* is zero, the natural number of processors is used. When all other threads complete their tasks, *parstack* returns zero.

Variables and Macros

The following variables and macros are also available.

extern int _procno	Variable used with *parbegin*, *parnext*, and *parstack* that allows you to keep track of the thread that is currently executing.
extern int _taskno	Variable used only with *parstack*. Its value ranges from zero to *n*−*1*, where *n* is the *parstack* argument.
extern volatile int thread_system	Variable used as a semaphore that gates system calls. This allows only one system call at a time per process.
THREAD_SYS(S)	This macro synchronizes statement *S* with the *thread_system* semaphore.
LOAD_SYNCH(v)	This macro expands into a *load-and-synch* operation on location *v*.
PROTECT(v,S)	This macro specifies that statement *S* is only done after the thread owns the semaphore *v*. *v* must be initialized to 1.

FILES

/usr/include/thread.h

NAME

time – get time

SYNOPSIS

#include <sys/types.h>

time_t time (tloc)
long *tloc;

DESCRIPTION

time 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.

SEE ALSO

stime(2).

WARNING

time fails and its actions are undefined if *tloc* points to an illegal address.

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

times 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.

[EFAULT] *times* will fail if *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 (e.g., system start-up time). This point does not change from one invocation of *times* to another. If *times* fails, a –1 is returned and *errno* is set to indicate the error.

NAME

truncate – truncate a file to a specified length

SYNOPSIS

truncate(path, length)
char *path;
off_t length;

ftruncate(fd, length)
int fd;
off_t length;

DESCRIPTION

truncate causes the file named by *path* or referenced by *fd* to be truncated to at most *length* bytes in size. If the file previously was larger than this size, the extra data is lost. With *ftruncate*, the file must be open for writing.

RETURN VALUES

A value of 0 is returned if the call succeeds. If the call fails a –1 is returned, and the global variable *errno* specifies the error.

ERRORS

truncate succeeds unless:

[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]	The named file is not writable by the user.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EISDIR]	The named file is a directory.
[EROFS]	The named file resides on a read-only file system.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed.
[EIO]	An I/O error occurred updating the inode.
[EFAULT]	*path* points outside the process's allocated address space.

ftruncate succeeds unless:

[EBADF]	The *fd* is not a valid descriptor.
[EINVAL]	The *fd* is not open for writing.

SEE ALSO

open(2)

BUGS

These calls should be generalized to allow ranges of bytes in a file to be discarded.

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 argument *mdep* is provided for machine-dependent use and is not defined here.

As specified by *cmd*, the following commands are available:

A_SHUTDOWN	The system is shutdown. All user processes are killed, the buffer cache is flushed, and the root file system is unmounted. The action to be taken after the system has been shut down is specified by *fcn*. The functions are generic; the hardware capabilities vary on specific machines.

AD_HALT	Return to PROM monitor.
AD_BOOT	Reboot the system, using /unix.
AD_IBOOT	Interactive reboot; user is prompted for system name.

A_REBOOT	The system stops immediately without any further processing. The action to be taken next is specified by *fcn* as above.
A_REMOUNT	The root file system is mounted again after having been fixed. This should be used only during the startup process.

uadmin fails in the following case:

[EPERM]	The effective user ID is not super-user.

DIAGNOSTICS

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

A_SHUTDOWN	Never returns.
A_REBOOT	Never returns.
A_REMOUNT	0

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 available are:

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 maximum number of open files limit. This is a system-configured value.

SEE ALSO

brk(2), write(2).

WARNING

ulimit is effective in limiting the growth of regular files. Pipes are currently limited to 5,120 bytes.

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

umask – set and get file creation mask

SYNOPSIS

int umask (cmask)
int cmask;

DESCRIPTION

umask sets the process's file mode creation mask to *cmask* and returns the previous value of the mask. Only the low-order 9 bits of *cmask* and the file mode creation mask are used.

SEE ALSO

chmod(2), creat(2), mkdir(1), mknod(2), open(2), sh(1)

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

umount requests that a previously mounted file system contained on the block special device or directory identified by *file* be unmounted. *file* is a pointer to a path name. After unmounting the file system, the directory upon which the file system was mounted reverts to its ordinary interpretation.

umount may be invoked only by the super-user.

umount 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 UNIX system

SYNOPSIS

#include <sys/utsname.h>

int uname (name)
struct utsname *name;

DESCRIPTION

uname stores information identifying the current UNIX system in the structure pointed to by *name*.

uname uses the structure defined in **<sys/utsname.h>** whose members are:

```
char    sysname[9];
char    nodename[9];
char    release[9];
char    version[9];
char    machine[9];
```

uname returns a null-terminated character string naming the current UNIX system in the character array *sysname*. Similarly, *nodename* contains the name that the system is known by on a communications network. *Release* and *version* further identify the operating system. *Machine* contains a standard name that identifies the hardware that the UNIX system is running on.

[EFAULT] *uname* fails if *name* points to an invalid address.

SEE ALSO

uname(1)

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.
[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

close(2), link(2), open(2), rm(1)

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

ustat returns information about a mounted file system. *dev* is a device number identifying a device containing a mounted file system. *buf* is a pointer to a *ustat* structure that includes 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 */
```

ustat 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), 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

utime – set file access and modification times

SYNOPSIS

```
#include <sys/types.h>
int utime (path, times)
char *path;
struct utimbuf *times;
```

DESCRIPTION

path points to a path name naming a file. *utime* sets the access and modification times of the named file.

If *times* is NULL, the access and modification times of the file are set to the current time. A process must be the owner of the file or have write permission to use *utime* in this manner.

If *times* is not NULL, *times* is interpreted as a pointer to a *utimbuf* structure and the access and modification times are set to the values contained in the designated structure. Only the owner of the file or the super-user may use *utime* this way.

The times in the following structure are measured in seconds since 00:00:00 GMT, Jan. 1, 1970.

```
struct  utimbuf{
        time_t  actime;     /* access time */
        time_t  modtime;    /* modification time */
};
```

utime 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

wait 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. 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 (i.e., bit 200) is set, a "core image" is produced [see *signal*(2)].

If a parent process terminates without waiting for its child processes to terminate, the parent process ID of each child process is set to 1. This means the initialization process inherits the child processes [see *intro*(2)].

wait fails and returns immediately in the following case:

[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).

WARNING

wait fails and its actions are undefined if *stat_loc* points to an invalid address.

See *WARNING* in *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.

NAME

write – write on a file

SYNOPSIS

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

DESCRIPTION

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

write attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the *fildes*.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from *write*, the file pointer is incremented by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

If the O_APPEND flag of the file status flags is set, the file pointer is 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 does 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 does not return until the data has been physically updated.

A write to a regular file is 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 are written. If *nbyte* does not fall within the range and the minimum packet size value is zero, *write* 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* fails 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* blocks until data can be accepted. O_NDELAY prevents a process from blocking due to flow control conditions. If O_NDELAY is set and the *stream* can not accept data, *write* fails. 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* terminates and returns the number of bytes written.

write fails and the file pointer does not change 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.

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 are written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512-bytes returns 20. The next write of a non-zero number of bytes gives 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) returns a count of 0. Otherwise (O_NDELAY clear), writes to a full pipe (or FIFO) blocks 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 UNIX system primitives, which are described in Section 2 of this volume. Certain major collections are identified by a letter after the section number:

(3C) These functions, together with those of Section 2 and those marked (3S), constitute the Standard C Library *libc*, which is automatically loaded by the C compiler, *cc*(1). (For this reason the (3C) and (3S) sections together comprise one section of this manual.) The link editor *ld*(1) searches this library under the **–lc** option. Declarations for some of these functions may be obtained from **#include** files indicated on the appropriate pages.

(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.

The function sets contained in the library are:

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.

(3F) These functions constitute the FORTRAN intrinsic function library, *libF77* available as *libmF77.a*, *libuF77.a*, *libiF77.a*. These functions are automatically available to the FORTRAN programmer and require no special invocation of the compiler.

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>**.

Many groups of FORTRAN intrinsic functions have *generic* function names that do not require explicit or implicit type declaration. The type of the function will be determined by the type of its argument(s). For example, the generic function *max* will return an integer value if given integer arguments (*max0*), a real value if given real arguments (*amax1*), or a double-precision value if given double-precision arguments (*dmax1*).

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.

FILES

LIBDIR usually /lib
LIBDIR/libc.a
LIBDIR/libm.a
LIBDIR/lib77.a
/usr/lib/libnsl_s.a (3N)
/usr/lib/libiF77.a
/usr/lib/libmF77.a
/usr/lib/libuF77.a

SEE ALSO

ar(1), cc(1), ld(1), nm(1), intro(2), stdio(3S), math(5).
f77(1) in the *FORTRAN Programming Language Manual*.

DIAGNOSTICS

Functions in the C and Math Libraries (3C and 3M) may return ±**Infinity** or ±**Nan** when a function is undefined for the given arguments or when the value is not representable. At this point, *errno* is **not** set in these cases because the hardware provides correct representation for the answers. Note that this behavior differs from many UNIX implementations which will return a value of ±HUGE (the largest-magnitude single-precision floating-point numbers; HUGE is defined in the *<math.h>* header file).

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.)

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* uses 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 an IOT fault

SYNOPSIS

int abort ( )

DESCRIPTION

abort 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.

abort 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

abs 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.

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

bsearch is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table indicating where a datum may be found. The table must be previously sorted in increasing order according to a provided comparison function. *key* points to 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 out 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

clock – report CPU time used

SYNOPSIS

long clock ( )

DESCRIPTION

clock returns the amount of CPU time (in microseconds) used since the first call to *clock*. The time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed *wait*(2), *pclose*(3S), or *system*(3S).

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).

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

toupper and *tolower* have as domain the range of *getc*(3S): the integers from –1 through 255. If the argument of *toupper* represents a lower-case letter, the result is the corresponding upper-case letter. If the argument of *tolower* represents an upper-case letter, the result is the corresponding lower-case letter. All other arguments in the domain are returned unchanged.

The macros *_toupper* and *_tolower*, are macros that accomplish the same thing as *toupper* and *tolower* but have restricted domains and are faster. *_toupper* requires a lower-case letter as its argument; its result is the corresponding upper-case letter. The macro *_tolower* requires an upper-case letter as its argument; its result is the corresponding lower-case letter. Arguments outside the domain cause undefined results.

toascii yields its argument with all bits turned off that are not part of a standard ASCII character; it is intended for compatibility with other systems.

SEE ALSO

ctype(3C), getc(3S)

NAME

cputim, systim, secnds – timing functions

SYNOPSIS

float cputim(oldtime)

float oldtime;

float systim(oldtime)

float oldtime;

float secnds(oldtime)

float oldtime;

DESCRIPTION

cputim returns the elapsed CPU time used by the current process minus the value of its argument *oldtime*.

systim returns the elapsed system time used by this process minus the value of its argument *oldtime*.

secnds returns the elapsed time since midnight minus the value of its argument *oldtime*.

All returned time is expressed in seconds and is accurate to 1/100th of a second.

If the argument to *cputim* or *systim* is zero, the value returned is the time used by this process. If the argument to *secnds* is zero, the value returned is the time since midnight. When the argument to any of these functions is non-zero, the function can be used as a split timer.

FILES

/usr/lib/libuF77.a

NOTES

secnds is a VMS built-in function, so it gets mapped to a different name by the Fortran compiler. Be aware of this implication if you are a C programmer.

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

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

key is a user's typed password. *salt* is a two-character string chosen from the set [**a-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 *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 is 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

getpass(3C), login(1), passwd(1), passwd(4)

CAVEAT

The return value points to static data that are overwritten by each call.

NAME

ctermid – generate file name for terminal

SYNOPSIS

#include <stdio.h>
char *ctermid (s)
char *s;

DESCRIPTION

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

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

NOTES

The difference between *ctermid* and *ttyname*(3C) is that *ttyname* must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while *ctermid* returns a string (**/dev/tty**) that refers to the terminal if used as a file name. Thus *ttyname* is useful only if the process already has at least one file open to a terminal.

SEE ALSO

ttyname(3C)

NAME

ctime, localtime, gmtime, asctime, tzset, tzsetwall – convert date and time to ASCII

SYNOPSIS

extern char *tzname[2];

void tzset()

void tzsetwall()

char *ctime(clock)
long *clock;

#include <time.h>

char *asctime(tm)
struct tm *tm;

struct tm *localtime(clock)
long *clock;

struct tm *gmtime(clock)
long *clock;

cc ... -lz

DESCRIPTION

Tzset uses the value of the environment variable **TZ** to set time conversion information used by *localtime*. If **TZ** does not appear in the environment, the best available approximation to local wall clock time is used by *localtime*. If **TZ** appears in the environment but its value is a null string, Greenwich Mean Time is used (without leap second correction); if **TZ** appears and begins with a slash, it is used as the absolute pathname of the *tzfile*(5)-format file from which to read the time conversion information; if **TZ** appears and begins with a character other than a slash, it's used as a pathname relative to a system time conversion information directory.

Tzsetwall sets things up so that *localtime* returns the best available approximation of local wall clock time.

Ctime converts a long integer, pointed to by *clock*, representing the time in seconds since 00:00:00 GMT, January 1, 1970, and returns a pointer to a 26-character string of the form

```
Thu Nov 24 18:22:48 1986\n\0
```

All the fields have constant width.

Localtime and *gmtime* return pointers to "tm" structures, described below. *Localtime* corrects for the time zone and any time zone adjustments (such as Daylight Saving Time in the U.S.A.). Before doing so, *localtime* calls *tzset* (if *tzset* has not been called in the current process). After filling in the "tm" structure, *localtime* sets the **tm_isdst**'th element of **tzname** to a pointer to an ASCII string that's the time zone abbreviation to be used with *localtime*'s return value.

Gmtime converts to Greenwich Mean Time (GMT).

Asctime converts a time value contained in 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 (of type) **struct tm** includes the following fields:

```
int tm_sec;        /* seconds (0 - 60) */
int tm_min;        /* minutes (0 - 59) */
int tm_hour;       /* hours (0 - 23) */
int tm_mday;       /* day of month (1 - 31) */
int tm_mon;        /* month of year (0 - 11) */
int tm_year;       /* year – 1900 */
int tm_wday;       /* day of week (Sunday = 0) */
int tm_yday;       /* day of year (0 - 365) */
int tm_isdst;      /* is DST in effect? */
char *tm_zone;     /* abbreviation of timezone name */
long tm_gmtoff;    /* offset from GMT in seconds */
```

The *tm_zone* and *tm_gmtoff* fields exist, and are filled in, only if arrangements to do so were made when the library containing these functions was created. There is no guarantee that these fields will continue to exist in this form in future releases of this code.

Tm_isdst is non-zero if a time zone adjustment such as Daylight Saving Time is in effect.

Tm_gmtoff is the offset (in seconds) of the time represented from GMT, with positive values indicating East of Greenwich.

FILES

/etc/zoneinfo	time zone information directory
/etc/zoneinfo/localtime	local time zone file
/etc/zoneinfo/GMT	GMT file (needed for leap seconds)

SEE ALSO

tzfile(5), getenv(3), time(2)

NOTE

The return values point to static data whose content is overwritten by each call. The **tm_zone** field of a returned **struct tm** points to a static array of characters, which will also be overwritten at the next call (and by calls to *tzset* or *tzsetwall*).

NAME

ctype: isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii – classify characters

SYNOPSIS

```
#include <ctype.h>

int isalpha (c)
int c;

. . .
```

DESCRIPTION

These macros classify character-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. *isascii* is defined on all integer values; the rest are defined only where *isascii* is true and on the single non-ASCII value **EOF** [–1; see *stdio*(3S)].

isalpha	*c* is a letter.
isupper	*c* is an upper-case letter.
islower	*c* is a lower-case letter.
isdigit	*c* is a digit [0-9].
isxdigit	*c* is a hexadecimal digit [0-9], [A-F] or [a-f].
isalnum	*c* is an alphanumeric (letter or digit).
isspace	*c* is a space, tab, carriage return, newline, vertical tab, or form-feed.
ispunct	*c* is a punctuation character (neither control nor alphanumeric).
isprint	*c* is a printing character, code 040 (space) through 0176 (tilde).
isgraph	*c* is a printing character, like *isprint* except false for space.
iscntrl	*c* is a delete character (0177) or an ordinary control character (less than 040).
isascii	*c* is an ASCII character, code less than 0200.

SEE ALSO

stdio(3S), ascii(5)

DIAGNOSTICS

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

NAME

cuserid – get character login name of the user

SYNOPSIS

#include <stdio.h>

char *cuserid (s)
char *s;

DESCRIPTION

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

DIAGNOSTICS

If the login name cannot be found, *cuserid* returns a NULL pointer; if *s* is not a NULL pointer, a null character (\0) is placed at *s[0]*.

SEE ALSO

getlogin(3C), getpwent(3C)

NAME

dial – establish an out-going terminal line connection

SYNOPSIS

#include <dial.h>

int dial (call)
CALL call;

void undial (fd)
int fd;

DESCRIPTION

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

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

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

```
typedef struct {
        struct termio *attr;     /* pointer to termio attribute struct */
        int           baud;      /* transmission data rate */
        int           speed;     /* 212A modem: low=300, high=1200 */
        char          *line;     /* device name for 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 */
        int           dev_len;   /* The length of the device used to make
                                 connection */
} CALL;
```

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

If the desired terminal line is a direct line, a string pointer to its device-name should be placed in the *line* element in the CALL structure. Legal values for such terminal device names are kept in the *L-devices* file. In this case, the value of the *baud* element need not be specified since it is determined from the *L-devices* file.

The *telno* element is for a pointer to a character string representing the telephone number to be dialed. Such numbers may consist only of symbols described on the *acu*(7). The termination symbol is supplied by the *dial* function, and should not be included in the *telno* string passed to *dial* in the CALL structure.

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

The **CALL** element *device* is used to hold the device name (cul..) that establishes the connection.

The **CALL** element *dev_len* is the length of the device name that is copied into the array device.

FILES

/usr/lib/uucp/L-devices
/usr/spool/uucp/LCK..*tty-device*

SEE ALSO

acu(7), alarm(2), read(2), termio(7), uucp(1C), write(2)

DIAGNOSTICS

On failure, a negative value is returned indicating the reason for the failure. The mnemonics listed here for these negative indices 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

The *dial* (3C) library function is not compatible with Basic Networking Utilities on UNIX System V Release 2.0.

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

This routine uses <*stdio.h*>, causing a greater than expected increase in the size of programs not otherwise using standard I/O.

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

dials – subroutines for accessing the dials

SYNOPSIS

fd = OpenDials();
nblocks = ReadDials(block, combine_flag)
int block[2];

DESCRIPTION

These routines should be used to open and read the dial box. *OpenDials* returns a file descriptor appropriate for *poll*(2). Error return is indicated by a –1 value, and *errno* will be valid.

Each call to *ReadDials* returns a count of the number of valid dial blocks returned from the dial box. It will wait for one block to arrive if there are none waiting at the time of the call. A zero return is possible if the dials get out of synchronization with the System.

The *block* argument is a two integer array: the first returned integer is a number between zero and seven indicating which dial was rotated; the second returned integer is the signed delta change for that dial. A positive number is clockwise. A full rotation of a dial is 256 ticks.

The *combine_flag* argument directs *ReadDials* to look ahead and combine multiple blocks (if True), or to simply return one sample (if False). For most graphics applications, the *combine_flag* should be true.

SEE ALSO

poll(2), tablet(3)

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

delim $$ 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 $[0,~2 sup 31).$

Functions *mrand48* and *jrand48* return signed long integers uniformly distributed over the interval $[-2 sup 31 ,~2 sup 31).$

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, $X sub i ,$ according to the linear congruential formula

X sub{n+1}~=~(aX sub n^+^c) sub{roman mod~m}~~~~~~~~n>=0.

The parameter $m^=^2 sup 48$; 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~mark =~roman 5DEECE66D^sub 16~=~roman 273673163155^sub 8
c~lineup =~roman B^sub 16~=~roman 13^sub 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 $X sub i$ 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 $X sub i$ and transformed into the returned value.

The functions *drand48, lrand48* and *mrand48* store the last 48-bit $X sub i$ 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 $X sub i$ 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 $X sub i$ into the array and pass it as an argument. By using different arguments, functions *erand48, nrand48* and *jrand48* allow separate modules of a large program to generate several *independent* streams of pseudo-random numbers, i.e., the sequence of numbers in each stream will *not* depend upon how many times the routines have been called to generate numbers for the other streams.

The initializer function *srand48* sets the high-order 32 bits of $X sub i$ to the 32 bits contained in its argument. The low-order 16 bits of $X sub i$ are set to the arbitrary value $roman 330E sub 16 .$

The initializer function *seed48* sets the value of $X sub i$ to the 48-bit value specified in the argument array. In addition, the previous value of $X sub i$ 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 $X sub i$ 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 $X sub i ,$ the multiplier value *a*, and the addend value *c*. Argument array elements *param[0-2]* specify $X sub i ,$ *param[3-5]* specify the multiplier *a*, and *param[6]* specifies the 16-bit addend *c*. After *lcong48* has been called, a subsequent call to either *srand48* or *seed48* will restore the "standard" multiplier and addend values, *a* and *c*, specified on the previous page.

NOTES

The source code for the portable version can be used on computers which do not have floating-point arithmetic. In such a situation, functions *drand48* and *erand48* are replaced by the two new functions below.

long irand48 (m)
unsigned short m;

long krand48 (xsubi, m)
unsigned short xsubi[3], m;

Functions *irand48* and *krand48* return non-negative long integers uniformly distributed over the interval $[0,~m-1]$. delim off

SEE ALSO

rand(3C).

NAME

dup2 – duplicate an open file descriptor

SYNOPSIS

int dup2 (fildes, fildes2)
int fildes, fildes2;

DESCRIPTION

fildes is a file descriptor referring to an open file, and *fildes2* is a non-negative integer less than NOFILES. *dup2* causes *fildes2* to refer to the same file as *fildes*. If *fildes2* already referred to an open file, it is closed first.

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

ecvt converts *value* to a null-terminated string of *ndigit* digits and returns a pointer thereto. The high-order digit is non-zero, unless the value is zero. The low-order digit is rounded. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). The decimal point is not included in the returned string. If the sign of the result is negative, the word pointed to by *sign* is non-zero, otherwise it is zero.

fcvt the same as *ecvt*, except that the correct digit has been rounded for printf "%f" (FORTRAN F-format) output of the number of digits specified by *ndigit*.

gcvt converts the *value* to a null-terminated string in the array pointed to by *buf* and returns *buf*. It attempts to produce *ndigit* significant digits in FORTRAN F-format if possible, otherwise E-format, ready for printing. A minus sign, if there is one, or a decimal point is included as part of the returned string. Trailing zeros are suppressed.

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

fclose, fflush – close or flush a stream

SYNOPSIS

#include <stdio.h>

int fclose (stream)
FILE ***stream;**

int fflush (stream)
FILE ***stream;**

DESCRIPTION

fclose causes any buffered data for the named *stream* to be written out, and the *stream* to be closed.

fclose is performed automatically for all open files upon calling *exit*(2).

Fflush causes any buffered data for the named *stream* to be written to that file. The *stream* remains open.

SEE ALSO

close(2), exit(2), fopen(3S), setbuf(3S), stdio(3S)

DIAGNOSTICS

These functions return 0 for success, and **EOF** if any error (such as trying to write to a file that has not been opened for writing) was detected.

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

ferror returns non-zero when an I/O error has previously occurred reading from or writing to the named *stream*, otherwise zero.

feof returns non-zero when EOF has previously been detected reading the named input *stream*, otherwise zero.

clearerr resets the error indicator and EOF indicator to zero on the named *stream*.

fileno returns the integer file descriptor associated with the named *stream*; see *open*(2).

NOTES

All these functions are implemented as macros; they cannot be declared or redeclared.

SEE ALSO

open(2), fopen(3S), stdio(3S)

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

fopen opens the file named by *filename* and associates a *stream* with it. *fopen* returns a pointer to the FILE structure associated with the *stream*.

filename points to a character string that contains the name of the file to be opened.

type is a character string having one of the following values:

"r"	open for reading
"w"	truncate or create for writing
"a"	append; open for writing at end of file, or create for writing
"r+"	open for update (reading and writing)
"w+"	truncate or create for update
"a+"	append; open or create for update at end-of-file

freopen substitutes the named file in place of the open *stream*. The original *stream* is closed, regardless of whether the open ultimately succeeds. *freopen* returns a pointer to the FILE structure associated with *stream*.

freopen is typically used to attach the preopened *streams* associated with **stdin**, **stdout**, and **stderr** to other files.

fdopen associates a *stream* with a file descriptor. File descriptors are obtained from *open*(2), *dup*(2), *creat*(2), 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 may be done on the resulting *stream*. However, output may not be directly followed by input without an intervening *fseek* or *rewind*, and input may not be directly followed by output without an intervening *fseek*, *rewind*, or an input operation which encounters end-of-file.

When a file is opened for append (i.e., when *type* is "a" or "a+"), it is impossible to overwrite information already in the file. *fseek* may be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output. If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written by the other. The output from the two processes is intermixed in the file in the order in which it is written.

SEE ALSO

creat(2), dup(2), fclose(3S), fseek(3S), open(2), pipe(2), stdio(3S)

DIAGNOSTICS

fopen, *fdopen*, and *freopen* return a **NULL** pointer on failure.

NAME

fputim – elapsed floating point processor time

SYNOPSIS

double fputim(oldtime)

double oldtime;

DESCRIPTION

fputim returns the elasped floating point processor time used by the current process minus the value of its argument *oldtime*. The time is expressed in seconds and is accurate to 16MHz.

If the argument to *fputim* is zero, the value returned is the time used by this process. If the argument is non-zero, the function can be used as a split timer.

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

fread 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*. *fread* stops appending bytes if an end-of-file or error condition is encountered while reading *stream*, or if *nitems* items have been read. *fread* leaves the file pointer in *stream*, if defined, pointing to the byte following the last byte read if there is one. *fread* does not change the contents of *stream*.

fwrite appends at most *nitems* items of data from the array pointed to by *ptr* to the named output *stream*. *fwrite* stops appending when it has appended *nitems* items of data or if an error condition is encountered on *stream*. *fwrite* does not change the contents of the array pointed to by *ptr*.

The argument *size* is typically *sizeof(*ptr)* where the pseudo-function *sizeof* specifies the length of an item pointed to by *ptr*. If *ptr* points to a data type other than *char* it should be cast into a pointer to *char*.

SEE ALSO

fopen(3S), getc(3S), gets(3S), printf(3S), putc(3S), puts(3S), read(2), scanf(3S), stdio(3S), write(2)

DIAGNOSTICS

fread and *fwrite* return the number of items read or written. If *nitems* is non-positive, no characters are read or written and 0 is returned by both *fread* and *fwrite*.

NAME

frexp, ldexp, modf – manipulate parts of floating-point numbers

SYNOPSIS

```
double frexp (value, eptr)
double value;
int *eptr;

double ldexp (value, exp)
double value;
int exp;

double modf (value, iptr)
double value, *iptr;
```

DESCRIPTION

Every non-zero number can be written uniquely as $x * 2^n$, where the "mantissa" (fraction) x is in the range $0.5 \leq |x| < 1.0$, and the "exponent" n is an integer. *frexp* returns the mantissa of a double *value*, and stores the exponent indirectly in the location pointed to by *eptr*. If *value* is zero, both results returned by *frexp* are zero.

ldexp returns the quantity $value * 2^{exp}$.

modf returns the signed fractional part of *value* and stores the integral part indirectly in the location pointed to by *iptr*.

DIAGNOSTICS

If *ldexp* would cause overflow, ±HUGE (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

fseek sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, from the current position, or from the end of the file, according as *ptrname* has the value 0, 1, or 2.

rewind(*stream*) is equivalent to *fseek*(*stream*, 0L, 0), except that no value is returned.

fseek and *rewind* undo any effects of *ungetc*(3S).

After *fseek* or *rewind*, the next operation on a file opened for update may be either input or output.

ftell returns the offset of the current byte relative to the beginning of the file associated with the named *stream*.

SEE ALSO

fopen(3S), lseek(2), popen(3S), stdio(3S), ungetc(3S)

DIAGNOSTICS

fseek returns non-zero for improper seeks, otherwise zero. An improper seek can be, for example, an *fseek* done on a file that has not been opened via *fopen;* in particular, *fseek* may not be used on a terminal, or on a file opened via *popen*(3S).

WARNING

Although on a 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

ftw recursively descends the directory hierarchy rooted in *path*. For each object in the hierarchy, *ftw* calls *fn*, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a **stat** structure [see *stat*(2)] containing information about the object, and an integer. Possible values of the integer, defined in the <*ftw.h*> header file, are FTW_F for a file, FTW_D for a directory, FTW_DNR for a directory that cannot be read, and FTW_NS for an object for which *stat* could not successfully be executed. If the integer is FTW_DNR, descendants of that directory are not processed. If the integer is FTW_NS, the **stat** structure contains garbage. An example of an object that causes FTW_NS to be passed to *fn* is a file in a directory with read but without execute (search) permission.

ftw visits a directory before visiting any of its descendants.

The tree traversal continues until the tree is exhausted, an invocation of *fn* returns a nonzero value, or some error is detected within *ftw* (such as an I/O error). If the tree is exhausted, *ftw* returns zero. If *fn* returns a nonzero value, *ftw* stops its tree traversal and returns whatever value was returned by *fn*. If *ftw* detects an error, it returns –1, and sets the error type in *errno*.

ftw uses one file descriptor for each level in the tree. The *depth* argument limits the number of file descriptors so used. If *depth* is zero or negative, the effect is the same as if it were 1. *depth* must not be greater than the number of file descriptors currently available for use. *ftw* runs more quickly if *depth* is at least as large as the number of levels in the tree.

SEE ALSO

stat(2), malloc(3C)

BUGS

Because *ftw* is recursive, it is possible for it to terminate with a memory fault when applied to very deep file structures.

CAVEAT

ftw uses *malloc*(3C) to allocate dynamic storage during its operation. If *ftw* is forcibly terminated, such as by *longjmp* being executed by *fn* or an interrupt routine, *ftw* does not have a chance to free that storage, so it remains permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have *fn* return a nonzero value at its next invocation.

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

getc returns the next character (i.e., byte) from the named input *stream*, as an integer. It also moves the file pointer, if defined, ahead one character in *stream*. *getchar* is defined as *getc(stdin)*. *getc* and *getchar* are macros.

fgetc behaves like *getc*, but is a function rather than a macro. *fgetc* runs more slowly than *getc*, but it takes less space per invocation and its name can be passed as an argument to a function.

getw returns the next word (i.e., integer) from the named input *stream*. *getw* increments the associated file pointer, if defined, to point to the next word. The size of a word is the size of an integer and varies from machine to machine. *getw* assumes no special alignment in the file.

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.

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.

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 may not be read using *getw* on a different processor.

NAME

getcwd – get path-name of current working directory

SYNOPSIS

char *getcwd (buf, size)
char *buf;
int size;

DESCRIPTION

getcwd returns a pointer to the current directory path name. The value of *size* must be at least two greater than the length of the path-name to be returned.

If *buf* is a **NULL** pointer, *getcwd* obtains *size* bytes of space using *malloc*(3C). In this case, the pointer returned by *getcwd* may be used as the argument in a subsequent call to *free.*

The function is implemented by using *popen*(3S) to pipe the output of the *pwd*(1) command into the specified string space.

EXAMPLE

```
void exit(), perror();
.
.
.
if ((cwd = getcwd((char *)NULL, 64)) == NULL) {
      perror("pwd");
      exit(2);
}
printf("%s\n", cwd);
```

SEE ALSO

malloc(3C), popen(3S), pwd(1)

DIAGNOSTICS

Returns **NULL** with *errno* set if *size* is not large enough, or if an error occurs in a lower-level function.

NAME

getenv – return value for environment name

SYNOPSIS

char *getenv (name)
char *name;

DESCRIPTION

getenv searches the environment list [see *environ*(5)] for a string of the form *name=value,* and returns a pointer to the *value* in the current environment if such a string is present, otherwise a NULL pointer.

SEE ALSO

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

getgrent, *getgrgid* and *getgrnam* each return pointers to an object with the following structure containing the broken-out fields of a line in the */etc/group* file. Each line contains a "group" structure, defined in the <*grp.h*> header file.

```
struct  group {
        char    *gr_name;       /* the name of the group */
        char    *gr_passwd;     /* the encrypted group password */
        int     gr_gid;         /* the numerical group ID */
        char    **gr_mem;       /* vector of pointers to member names */
};
```

getgrent when first called returns a pointer to the first group structure in the file; thereafter, it returns a pointer to the next group structure in the file; so, successive calls may be used to search the entire file. *getgrgid* searches from the beginning of the file until a numerical group id matching *gid* is found and returns a pointer to the particular structure in which it was found. *getgrnam* searches from the beginning of the file until a group name matching *name* is found and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setgrent* has the effect of rewinding the group file to allow repeated searches. *endgrent* may be called to close the group file when processing is complete.

fgetgrent 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.

WARNING

These routines use <*stdio.h*>, causing a greater than expected increase in the size of programs not otherwise using standard I/O.

CAVEAT

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

getlogin – get login name

SYNOPSIS

char *getlogin ( );

DESCRIPTION

getlogin returns a pointer to the login name as found in */etc/utmp*. It may be used in conjunction with *getpwnam* to locate the correct password file entry when the same user ID is shared by several login names.

If *getlogin* is called within a process that is not attached to a terminal, it returns a NULL pointer. The correct procedure for determining the login name is to call *cuserid*, or to call *getlogin* and if it fails to call *getpwuid*.

FILES

/etc/utmp

SEE ALSO

cuserid(3S), getgrent(3C), getpwent(3C), utmp(4)

DIAGNOSTICS

Returns the NULL pointer if *name* is not found.

CAVEAT

The return values point to static data whose contents are overwritten by each call.

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

getopt 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)). New commands will adhere to the command syntax standard, if they use *getopts*(1) or *getopt* to parse positional parameters and check for options that are legal for that command.

optstring must contain the option letters expected to be recognized by the command using *getopt*; if a letter is followed by a colon, the option is expected to have an argument, or group of arguments, which must be separated from it by white space.

optarg is set to point to the start of the option-argument on return from *getopt*.

getopt places in *optind* the *argv* index of the next argument to be processed. *optind* is external and initialized to 1 before the first call to *getopt*.

When all options have been processed (i.e., up to the first non-option argument), *getopt* returns –1. The special option "––" may be used to delimit the end of the options; when it is encountered, –1 is returned, and "––" skipped.

DIAGNOSTICS

getopt prints an error message on standard error and returns a question mark (?) when it encounters an option letter not included in *optstring* or no option-argument after an option that expects one. This error message may be disabled by setting to 0.

EXAMPLE

The following code fragment shows how to 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)) {
    :
}
```

WARNING

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 –aboxxx file (Rule 5 violation: options with option-arguments must not be grouped with other options)
cmd –ab –oxxx file (Rule 6 violation: there must be white space after an option that takes an option-argument)

SEE ALSO

getopts(1), intro(1)

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

getpass reads up to a newline or EOF from the file */dev/tty*, after prompting on the standard error output with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most 8 characters. If */dev/tty* cannot be opened, a NULL pointer is returned. An interrupt terminates input and sends an interrupt signal to the calling program before returning.

FILES

/dev/tty

WARNING

This routine uses <*stdio.h*>, causing a greater than expected increase in the size of programs not otherwise using standard I/O.

CAVEAT

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

NAME

getpw – get name from UID

SYNOPSIS

int getpw (uid, buf)
int uid;
char *buf;

DESCRIPTION

getpw searches the password file for a user id number that equals *uid*, copies the line of the password file in which *uid* was found into the array pointed to by *buf*, and returns 0. *getpw* returns non-zero if *uid* cannot be found.

This routine is included only for compatibility with prior systems and should not be used; see *getpwent*(3C) for routines to use instead.

FILES

/etc/passwd

SEE ALSO

getpwent(3C), passwd(4)

DIAGNOSTICS

getpw returns non-zero on error.

WARNING

This routine uses <*stdio.h*>, causing a greater than expected increase in the size of programs not otherwise using standard I/O.

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

getpwent, *getpwuid* and *getpwnam* each returns a pointer to an object with the following structure containing the broken-out fields of a line in the */etc/passwd* file. Each line in the file contains a "passwd" structure, declared in the *<pwd.h>* header file:

```
struct passwd {
        char    *pw_name;
        char    *pw_passwd;
        int     pw_uid;
        int     pw_gid;
        char    *pw_age;
        char    *pw_comment;
        char    *pw_gecos;
        char    *pw_dir;
        char    *pw_shell;
};
```

This structure is declared in *<pwd.h>* so it is not necessary to redeclare it.

The fields have meanings described in *passwd*(4).

getpwent when first called returns a pointer to the first passwd structure in the file; thereafter, it returns a pointer to the next passwd structure in the file; so successive calls can be used to search the entire file. *getpwuid* searches from the beginning of the file until a numerical user id matching *uid* is found and returns a pointer to the particular structure in which it was found. *getpwnam* searches from the beginning of the file until a login name matching *name* is found, and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setpwent* has the effect of rewinding the password file to allow repeated searches. *endpwent* may be called to close the password file when processing is complete.

fgetpwent returns a pointer to the next passwd structure in the stream *f*, which matches the format of */etc/passwd*.

FILES

/etc/passwd

SEE ALSO

getlogin(3C), getgrent(3C), passwd(4)

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

WARNING

These routines use *<stdio.h>*, causing a greater than expected increase in the size of programs not otherwise using standard I/O.

CAVEAT

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

gets, fgets – get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char *gets (s)
char *s;

char *fgets (s, n, stream)
char *s;
int n;
FILE *stream;
```

DESCRIPTION

gets reads characters from the standard input stream, *stdin,* into the array pointed to by *s*, until a new-line character is read or an end-of-file condition is encountered. The new-line character is discarded and the string is terminated with a null character.

fgets reads characters from the *stream* into the array pointed to by *s*, until *n*–1 characters are read, or a new-line character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null character.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), getc(3S), scanf(3S), 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

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

getutent, *getutid* and *getutline* each returns a pointer to a structure of the following type:

```
struct  utmp {
        char      ut_user[8];      /* User login name */
        char      ut_id[4];        /* /etc/inittab id (usually line #) */
        char      ut_line[12];     /* device name (console, lnxx) */
        short     ut_pid;          /* process id */
        short     ut_type;         /* type of entry */
        struct    exit_status {
          short     e_termination;     /* Process termination status */
          short     e_exit;        /* Process exit status */
        } ut_exit;                 /* The exit status of a process
                                    * marked as DEAD_PROCESS. */
        time_t    ut_time;         /* time entry was made */
};
```

getutent reads in the next entry from a *utmp*-like file. If the file is not already open, *getutent* opens it. If the routine reaches the end of the file, it fails.

getutid searches forward from the current point in the *utmp* file until it finds an entry with a *ut_type* matching *id->ut_type* if the type specified is RUN_LVL, BOOT_TIME, OLD_TIME or NEW_TIME. If the type specified in *id* is INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS or DEAD_PROCESS, then *getutid* 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, *getutid* fails.

getutline searches forward from the current point in the *utmp* file until it finds an entry of the type LOGIN_PROCESS or USER_PROCESS which also has a *ut_line* string matching the *line->ut_line* string. If the end of file is reached without a match, *getutline* fails.

pututline writes out the supplied *utmp* structure into the *utmp* file. It uses *getutid* to search forward for the proper place if it is not already at the proper place. It is expected that normally the user of *pututline* has searched for the proper entry using one of the *getut* routines. If so, *pututline* does not search. If *pututline* does not find a matching slot for the new entry, it adds a new entry to the end of the file.

setutent resets the input stream to the beginning of the file. This should be done before each search for a new entry if it is desired that the entire file be examined.

endutent closes the currently open file.

utmpname allows the user to change the name of the file examined, from */etc/utmp* to any other legal filename. It is most often expected that this is */etc/wtmp*. If */etc/wtmp* does not exist, this is not apparent until the first attempt to reference the file is made. *utmpname* does not open the file. It just closes the old file if it is currently open and saves the new file name.

FILES

/etc/utmp
/etc/wtmp

SEE ALSO

ttyslot(3C), utmp(4)

DIAGNOSTICS

A **NULL** pointer is returned upon failure to read, whether for permissions or having reached the end of file, or upon failure to write.

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, zero out the static structure after each success. Otherwise, *getutline* just returns the same pointer over and over again.

There is one exception to the rule about removing the structure before further reads are done. The implicit read done by *pututline* (if it finds that it is not already at the correct place in the file) does not hurt the contents of the static structure returned by the *getutent*, *getutid* or *getutline* routines, if the user has just modified those contents and passed the pointer back to *pututline*.

These routines use buffered standard I/O for input, but *pututline* uses an unbuffered non-standard write to avoid race conditions between processes trying to modify the *utmp* and *wtmp* files.

NAME

hsearch, hcreate, hdestroy – manage hash search tables

SYNOPSIS

#include <search.h>

ENTRY *hsearch (item, action)
ENTRY item;
ACTION action;

int hcreate (nel)
unsigned nel;

void hdestroy ( )

DESCRIPTION

hsearch is a hash-table search routine generalized from Knuth (6.4) Algorithm D. It returns a pointer into a hash table indicating the location at which an entry can be found. *item* is a structure of type ENTRY (defined in the <*search.h*> header file) containing two pointers: *item.key* points to the comparison key, and *item.data* points to any other data to be associated with that key. (Pointers to types other than character should be cast to pointer-to-character.) *action* is a member of an enumeration type ACTION indicating the disposition of the entry if it cannot be found in the table. ENTER indicates that the item should be inserted in the table at an appropriate point. FIND indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a NULL pointer.

hcreate allocates sufficient space for the table, and must be called before *hsearch* is used. *nel* is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

hdestroy destroys the search table, and may be followed by another call to *hcreate*.

NOTES

hsearch uses *open addressing* with a *multiplicative* hash function. However, its source code has many other options available which the user may select by compiling the *hsearch* source with the following symbols defined to the preprocessor:

DIV	Use the *remainder modulo table size* as the hash function instead of the multiplicative algorithm.
USCR	Use a User Supplied Comparison Routine for ascertaining table membership. The routine should be named *hcompar* and should behave in a mannner 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). Consult the source code for further details.

EXAMPLE

The following example reads in strings followed by two numbers and stores them in a hash table, discarding duplicates. It then reads in strings and finds the matching entry in the hash table and prints it out.

```
#include <stdio.h>
#include <search.h>
struct info {             /* this is the info stored in the table */
        int age, room;    /* 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

hsearch returns a NULL pointer either if the action is FIND and the item could not be found or the action is ENTER and the table is full.

hcreate returns zero if it cannot allocate sufficient space for the table.

WARNING

hsearch and *hcreate* use *malloc*(3C) to allocate space.

CAVEAT

Only one hash search table may be active at any given time.

NAME

IDAMAX – a set of routines to find the index in an array of the element having largest or smallest value

SYNOPSIS

int_IDAFMAX(count, array, stride)
double *array;
int count, stride;

DESCRIPTION

These functions are available for finding the index in an array of the element having the greatest or smallest value:

_IDAFMAX	_IIFFMAX
_IDAFMIN	_IIFMIN
_IDALMAX	_IILMAX
_IDALMIN	_IILMIN
_IDFMAX	_ISAFMAX
_IDFMIN	_ISAFMIN
_IDLMAX	_ISALMAX
_IDLMIN	_ISALMIN
_IIAFMAX	_ISFMAX
_IIAFMIN	_ISFMIN
_IIALMAX	_ISLMAX
_IIALMIN	_ISLMIN

The name of the function determines the type of array searched and which element is returned by the following convention:

(1) The third letter of each function name determines the type of array which is searched. A "d" indicates that the routine expects a *double* * as the first parameter; an "s" indicates that the routine expects a *float* *; and a "i" indicates that the routine expects a *int* *.

(2) If the next letter in the routine name is an "a", then the routine takes the absolute value of all the elements before searching.

(3) The next letter is always either an "f" or an "l." An "f" indicates that the routine returns the first occurence in the case of duplicates; an "l" indicates that the routine returns the last occurence.

(4) Finally, the last three letters of "min" or "max" indicates whether the routine searches for a maximum value or a minimum value.

The parameter *count* indicates the number of elements of the array to search, and the parameter *stride* indicates the stride to use in stepping through *array*.

The following calls should clarify the use of these routines. Assume the array contains the following double precision values:

1.0, 2.0, 3.0, -2.0, -1.0

(1) _IDAFMIN(5,a,1) returns 0 because both the first and last elements have the smallest absolute value, and the function returns the index of the first.

(2) _IDALMIN(5,a,1) returns 4 because the last element with the smallest absolute value is element 4.

(3) _IDFMIN(5,a,1) returns 3 because -2.0 is the smallest element.

(4) _IDFMIN(3,a,2) returns 2 because -1.0 is the smallest of elements 1, 3, and 5. Note that the return value does *not* consider the stride; it returns the offset within the passed-in array.

These files use the Stardent 1500/3000 vector hardware to perform the necessary computations very rapidly. Their unusual names result because calls to them are automatically generated by the compiler; however, there should be no problems with users calling them.

CAVEATS

Comparison against NaN is undefined on Stardent 1500/3000; hence calling these routines on a vector that contains NaN's may give unexpected results. Note that these are *not* vector-valued functions, as are most Stardent 1500/3000 math routines, so that they take the address of a vector in memory and not a vector register.

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

isnand and *isnanf* 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.

isnanf() 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

l3tol converts a list of *n* three-byte integers packed into a character string pointed to by *cp* into a list of long integers pointed to by *lp*.

ltol3 performs the reverse conversion from long integers (*lp*) to three-byte integers (*cp*).

These functions are useful for file-system maintenance where the block numbers are three bytes long.

SEE ALSO

fs(4)

CAVEAT

Because of possible differences in byte ordering, the numerical values of the long integers are machine-dependent.

NAME

lockf – record locking on files

SYNOPSIS

```
#include <unistd.h>

int lockf (fildes, function, size)
long size;
int fildes, function;
```

DESCRIPTION

The *lockf* command allows sections of a file to be locked; advisory or mandatory write locks depending on the mode bits of the file [see *chmod*(2)]. Locking calls from other processes which attempt to lock the locked file section either return an error value or are 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 which specifies the action to be taken. The permissible values for *function* are defined in **<unistd.h>** as follows:

```
#define F_ULOCK  0      /* Unlock a previously locked section */
#define F_LOCK   1      /* Lock a section for exclusive use */
#define F_TLOCK  2      /* Test and lock a section for exclusive use */
#define F_TEST   3      /* Test section for other processes locks */
```

All other values of *function* are reserved for future extensions and 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 (i.e., from the current offset through the present or any future end-of-file). An area need not be allocated to the file in order to be locked as such locks may exist past the end-of-file.

The sections locked with F_LOCK or F_TLOCK may, in whole or in part, contain or be contained by a previously locked section for the same process. When this occurs, or if adjacent sections occur, the sections are combined into a single section. If the request requires that a new element be added to the table of active locks and this table is already full, an error is returned, and the new section is not locked.

F_LOCK and F_TLOCK requests differ only by the action taken if the resource is not available. F_LOCK causes the calling process to sleep until the resource is available. F_TLOCK causes the function to return a –1 and set *errno* to [EACCES] 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*(2) scan for a deadlock prior to sleeping on a locked resource. An error return is made if sleeping on the locked resource would cause a deadlock.

Sleeping on a resource is interrupted with any signal. The *alarm*(2) command may be used to provide a timeout facility in applications which require this facility.

The *lockf* utility fails if one or more of the following are true:

[EBADF]
fildes is not a valid open descriptor.

[EACCES]
cmd is F_TLOCK or F_TEST and the section is already locked by another process.

[EDEADLK]
cmd is F_LOCK and a deadlock would occur. Also the *cmd* is either F_LOCK, F_TLOCK, or F_ULOCK and the number of entries in the lock table would exceed the number allocated on the system.

[ECOMM]
fildes is located 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 which is/was locked. The standard I/O package is the most common source of unexpected buffering.

Because in the future the variable *errno* will be set to EAGAIN rather than EACCES when a section of a file is already locked by another process, portable application programs should expect and test for either value.

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, nothing is 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 reads in less than TABSIZE strings of length less than ELSIZE and stores them in a table, eliminating duplicates.

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

#define TABSIZE 50
#define ELSIZE 120

        char line[ELSIZE], tab[TABSIZE][ELSIZE], *lsearch( );
        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 occur if the space assigned by *malloc* is overrun or if some random number is handed to *free*.

malloc allocates the first contiguous, sufficiently large 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 are 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* asks *malloc* to enlarge the arena by *size* bytes and then moves 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 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(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

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 uses 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

mktemp replaces the contents of the string pointed to by *template* by a unique file name, and returns the address of *template*. The string in *template* should look like a file name with six trailing Xs; *mktemp* 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

mktemp assigns to *template* the NULL string if it cannot create a unique name.

CAVEAT

If called more than 17,576 times in a single process, this function 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, 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);
```

Use *prof*(1) 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

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 is set to 0 unless the file was compiled with the –g option. If the name is not found, both entries are set to 0. See *a.out*(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.

NOTES

The <*nlist.h*> header file is automatically included by <*a.out.h*> for compatability. However, if the only information needed from <*a.out.h*> is for use of *nlist*, then including <*a.out.h*> is discouraged. If <*a.out.h*> is included, the line "#undef n_name" may need to follow it.

SEE ALSO

a.out(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.

NAME

perror, errno, sys_errlist, sys_nerr – system error messages

SYNOPSIS

void perror (s)
char *s;

extern int errno;

extern char *sys_errlist[ ];

extern int sys_nerr;

DESCRIPTION

perror produces a message on the standard error output, describing the last error encountered during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, then the message and a new-line. (However, if s="" the colon is not printed.) To be of most use, the argument string should include the name of the program that incurred the error. The error number is taken from the external variable *errno*, which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the array of message strings *sys_errlist* is provided; *errno* can be used as an index into this table to get the message string without the new-line. *Sys_nerr* is the number of messages in the table; it should be checked because new error codes may be added to the system before they are added to the table.

SEE ALSO

intro(2)

NAME

popen, pclose – initiate pipe to/from a process

SYNOPSIS

```
#include <stdio.h>

FILE *popen (command, type)
char *command, *type;

int pclose (stream)
FILE *stream;
```

DESCRIPTION

popen creates a pipe between the calling program and the command to be executed. The arguments to *popen* are pointers to null-terminated strings. *command* consists of a shell command line. *type* is an I/O mode, either **r** for reading or **w** for writing. The value returned is a stream pointer such that one can write to the standard input of the command, if the I/O mode is **w**, by writing to the file *stream*; and one can read from the standard output of the command, if the I/O mode is **r**, by reading from the file *stream*.

A stream opened by *popen* should be closed by *pclose*, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type **r** command may be used as an input filter and a type **w** as an output filter.

EXAMPLE

A typical call may be:

```
char *cmd = "ls *.c";
FILE *ptr;
if ((ptr = popen(cmd, "r")) != NULL)
    while (fgets(buf, n, ptr) != NULL)
        (void) printf("%s ",buf);
```

This prints in *stdout* [see *stdio* (3S)] all the file names in the current directory that have a ".c" suffix.

SEE ALSO

fclose(3S), fopen(3S), pipe(2), stdio(3S), system(3S), wait(2)

DIAGNOSTICS

popen returns a **NULL** pointer if files or processes cannot be created.

pclose 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 may be forestalled by careful buffer flushing, e.g. 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 is padded on the left (or right, if the left-adjustment flag '–', described below, has been given) to the field width. The padding is done with blanks unless the field width digit string starts with a zero, in which case the padding is done with zeros.
>
> A *precision* that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, or **X** conversions, the number of digits to appear after the decimal point for the **e**, **E**, and **f** conversions, the maximum number of significant digits for the **g** and **G** conversion, or the maximum number of characters to be printed from a string in **s** conversion. The precision takes the form of a period (.) followed by a decimal digit string; a null digit string is treated as zero. Padding specified by the precision overrides the padding specified by the field width.
>
> An optional **l** (ell) specifying that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion character applies to a long integer *arg*. An **l** before any other conversion character is ignored.
>
> A character that indicates the type of conversion to be applied.

A field width or precision or both may be indicated by an asterisk (*) instead of a digit string. In this case, an integer *arg* supplies the field width or precision. The *arg* that is actually converted is not fetched until the conversion letter is seen, so the *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 is changed to zero.

The flag characters and their meanings are:

–	The result of the conversion is left-justified within the field.
+	The result of a signed conversion always begins with a sign (+ or –).
blank	If the first character of a signed conversion is not a sign, a blank is prefixed to the result. This implies that if the blank and + flags both appear, the blank flag is ignored.
#	This flag specifies that the value is converted to an "alternate form." For **c**, **d**, **i**, **s**, and **u** conversions, the flag creates 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 is prefixed with **0x** or **0X**. For **e**, **E**, **f**, **g**, and **G** conversions, the result always contains a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For **g** and **G** conversions, trailing zeroes are *not* removed from the result (which they normally are).

The conversion characters and their meanings are:

d,i,o,u,x,X	The integer *arg* is converted to signed decimal (**d** or **i**), unsigned octal, (**o**), decimal (**u**), or hexadecimal notation (**x** or **X**), respectively; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeroes. The default precision is 1. The result of converting a zero value with a precision of zero is a null string.
f	The float or double *arg* is converted to decimal notation in the style "[–]ddd.ddd," where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, six digits are output; if the precision is explicitly 0, no decimal point appears.
e,E	The float or double *arg* is converted in the style "[–]d.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 produces a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits.
g,G	The float or double *arg* is printed in style **f** or **e** (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style **e** is used only if the exponent resulting from the conversion is less than –4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.
c	The character *arg* is printed.
s	The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (\0) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed. A **NULL** value for *arg* yields undefined results.

% Print a %; no argument is converted.

In printing floating point types (float and double), if the exponent is 0x7FF and the mantissa is not equal to zero, then the output is

    [–]NaN0xdddddddd

where 0xdddddddd is the hexadecimal representation of the leftmost 32 bits of the mantissa. If the mantissa is zero, the output is

    [±]inf.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by *printf* and *fprintf* are printed as if *putc*(3S) had been called.

EXAMPLES

To print a date and time in the form "Sunday, July 3, 10:02," where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %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 (i.e. integer) *w* to the output *stream* (at the position at which the file pointer, if defined, is pointing). The size of a word is the size of an integer and varies from machine to machine. *putw* neither assumes nor causes special alignment in the file.

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 occurs 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 may not 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 changes the environment. The space used by *string* is no longer used once a new string-defining *name* is passed to *putenv*.

SEE ALSO

exec(2), getenv(3C), malloc(3C), environ(5)

DIAGNOSTICS

putenv returns non-zero if it was unable to obtain enough space via *malloc* for an expanded environment, otherwise zero.

WARNINGS

putenv manipulates the environment pointed to by *environ*, and can be used in conjunction with *getenv*. However, *envp* (the third argument to *main*) is not changed.
This routine uses *malloc*(3C) to enlarge the environment.
After *putenv* is called, environmental variables are not kept in alphabetical order.
A potential error is to call *putenv* with an automatic variable as the argument, then exit the calling function while *string* is still part of the environment.

NAME

putpwent – write password file entry

SYNOPSIS

```
#include <pwd.h>

int putpwent (p, f)
struct passwd *p;
FILE *f;
```

DESCRIPTION

putpwent is the inverse of *getpwent*(3C). Given a pointer to a passwd structure created by *getpwent* (or *getpwuid* or *getpwnam*), *putpwent* writes a line on the stream *f*, which matches the format of */etc/passwd*.

SEE ALSO

getpwent(3C)

DIAGNOSTICS

putpwent returns non-zero if an error was detected during its operation, otherwise zero.

WARNING

This routine uses <*stdio.h*>, causing a greater than expected increase in the size of programs not otherwise using standard I/O.

NAME

puts, fputs – put a string on a stream

SYNOPSIS

#include <stdio.h>

int puts (s)
char *s;

int fputs (s, stream)
char *s;
FILE *stream;

DESCRIPTION

puts writes the null-terminated string pointed to by *s* ,followed by a new-line character, to the standard output stream *stdout.*

fputs writes the null-terminated string pointed to by *s* to the named output *stream*.

Neither function writes the terminating null character.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), printf(3S), putc(3S), stdio(3S).

DIAGNOSTICS

Both routines return EOF on error. This happens if the routines try to write on a file that has not been opened for writing.

NOTES

puts appends a new-line character while *fputs* does not.

NAME

qsort – quicker sort

SYNOPSIS

void qsort ((char *) base, nel, sizeof (*base), compar)
unsigned 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. As the function must return an integer less than, equal to, or greater than zero, so must the first argument to be considered be less than, equal to, or greater than the second.

NOTES

The pointer to the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.
The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.
The order in the output of two items which compare as equal is unpredictable.

SEE ALSO

bsearch(3C), lsearch(3C), sort(1), string(3C)

NAME

rand, srand – simple random-number generator

SYNOPSIS

int rand ( )

void srand (seed)
unsigned seed;

DESCRIPTION

rand uses a multiplicative congruential random-number generator with period 2^{32} that returns successive pseudo-random numbers in the range from 0 to $2^{15}-1$.

srand can be called at any time to reset the random-number generator to a random starting point. The generator is initially seeded with a value of 1.

NOTES

The spectral properties of *rand* are limited. *Drand48*(3C) provides a much better, though more elaborate, random-number generator.

SEE ALSO

drand48(3C)

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 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.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. White-space characters (blanks, tabs, new-lines, or form-feeds) which, except in two cases described below, cause input to be read up to the next non-white-space character.
2. An ordinary character (not %), which must match the next character of the input stream.
3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width, an optional **l** (ell) or **h** indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. The suppression of assignment provides a way of describing an input field which is to be skipped. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted. For all descriptors except "[" and "c", white space leading an input field is ignored.

The conversion code indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. For a suppressed field, no pointer argument is given. The following conversion codes are legal:

%	a single % is expected in the input at this point; no assignment is done.
d	a decimal integer is expected; the corresponding argument should be an integer pointer.
u	an unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.
o	an octal integer is expected; the corresponding argument should be an integer pointer.

x a hexadecimal integer is expected; the corresponding argument should be an integer pointer.

i an integer is expected; the corresponding argument should be an integer pointer. It 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 is added automatically. The input field is terminated by a white-space character.

c a character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-space character, use **%1s**. If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read.

[indicates string data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, 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 may be represented by the construct *first–last*, thus [0123456789] may be expressed [0–9]. Using this convention, *first* must be lexically less than or equal to *last*, or else the dash stands for itself. The dash also stands for itself whenever it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset, and in this case it is not syntactically interpreted as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating **\0**, which is added automatically. At least one character must match for this conversion to be considered successful.

The conversion characters **d**, **u**, **o**, **x**, and **i** may be preceded by **l** or **h** to indicate that a pointer to **long** or to **short** rather than to **int** is in the argument list. Similarly, the conversion characters **e**, **f**, and **g** may be preceded by **l** to indicate that a pointer to **double** rather than to **float** is in the argument list. The **l** or **h** modifier is ignored for other conversion characters.

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
```

assigns to *n* the value 3, to *i* the value **25**, to *x* the value **5.432**, and *name* contains **thompson\0**. Or:

```
int i, j; float x; char name[50];
(void) scanf("%i%2d%f%*d %[0–9] ", &j, &i, &x, name);
```

with input:

```
011 56789 0123 56a72
```

assigns **9** to *j*, **56** to *i*, **789.0** to *x*, skips **0123**, and places the string **56\0** in *name*. The next call to *getchar* [see *getc*(3S)] returns **a**. Or:

```
int i, j, s, e; char name[50];
(void) scanf("%i %i %n%s%n", &i, &j, &s, name, &e);
```

with input:

```
0x11 0xy johnson
```

assigns **17** to *i*, **0** to *j*, **6** to *s*, places the string **xy\0** in *name*, and assigns **8** to *e*. Thus, the length of *name* is *e* – 2 . The next call to *getchar* [see *getc*(3S)] returns 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 new-line) 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 may be used after a stream has been opened but before it is read or written. It causes the array pointed to by *buf* to be used instead of an automatically allocated buffer. If *buf* is the NULL pointer, I/O is completely unbuffered.

A constant BUFSIZ, defined in the *<stdio.h>* header file, tells how big an array is needed:

    char buf[BUFSIZ];

setvbuf may be used after a stream has been opened but before it is read or written. *type* determines how *stream* is buffered. Legal values for *type* (defined in stdio.h) are:

_IOFBF	causes input/output to be fully buffered.
_IOLBF	causes output to be line buffered; the buffer is flushed when a newline is written, the buffer is full, or input is requested.
_IONBF	causes I/O to be completely unbuffered.

If *buf* is not the NULL pointer, the array it points to is used for buffering, instead of an automatically allocated buffer. *size* specifies the size of the buffer to be used. The constant BUFSIZ in *<stdio.h>* is suggested as a good buffer size. If I/O is unbuffered, *buf* and *size* are ignored.

By default, output to a terminal is line buffered and all other I/O 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 is 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* returns 1. At the time of the second return from *setjmp*, all variables (local and global) contain the last values they had before the call of *longjmp* (see example).

This behavior is enforced by a pragma hidden in the definition of *setjmp* in the file *setjmp.h* Accordingly, *setjmp* is really a macro, and (for example) its address cannot be passed to another routine, and it cannot safely be invoked from an assembly language routine.

EXAMPLE

```
#include <setjmp.h>

jmp_buf env;
int i = 0;
main ()
{
        void exit();

        if(setjmp(env) != 0) {
                (void) printf("value of i on 2nd return from setjmp: %d\n", i);
                exit(0);
        }
        (void) printf("value of i on 1st return from setjmp: %d\n", i);
        i = 1;
        g();
        /*NOTREACHED*/
}

g()
{
        longjmp(env, 1);
        /*NOTREACHED*/
}
```

If the *a.out* resulting from this C language code is run, the output is:

```
value of i on 1st return from setjmp:0

value of i on 2nd return from setjmp:1
```

SEE ALSO

signal(2)

WARNING

Absolute chaos is guaranteed in either of the following cases:

1. If *longjmp* is called even though *env* was never primed by a call to *setjmp*.
2. If the last such call was in a function which has since returned.

NAME

sleep – suspend execution for interval

SYNOPSIS

unsigned sleep (seconds)
unsigned seconds;

DESCRIPTION

The current process is suspended from execution for the number of *seconds* specified by the argument. The actual suspension time may be less than that requested for two reasons: (1) Because scheduled wakeups occur at fixed 1-second intervals, (on the second, according to an internal clock) and (2) because any caught signal terminates the *sleep* following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount due to the scheduling of other activity in the system. The value returned by *sleep* is the "unslept" amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested *sleep* time, or premature arousal due to another caught signal.

The routine is implemented by setting an alarm signal and pausing until it (or some other signal) occurs. The previous state of the alarm signal is saved and restored. The calling program may have set up an alarm signal before calling *sleep*. If the *sleep* time exceeds the time till such alarm signal, the process sleeps only until the alarm signal would have occurred. The caller's alarm catch routine is executed just before the *sleep* routine returns. But if the *sleep* time is less than the time till such alarm, the prior alarm time is reset to go off at the same time it would have without the intervening *sleep*.

SEE ALSO

alarm(2), pause(2), signal(2)

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* contains 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 freely intermixed.

A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type FILE. *fopen*(3S) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the <*stdio.h*> header file and associated with the standard open files:

stdin	standard input file
stdout	standard output file
stderr	standard error file

A constant NULL (0) designates a nonexistent pointer.

An integer-constant 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:

```
#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)] causes it to become buffered or line-buffered. When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as written; when it is buffered, many characters are saved up and written as a block. When it is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new-line character is written or terminal input is requested). *setbuf*(3S) or *setvbuf*(3S) in *setbuf*(3S) may be used to change the stream's buffering strategy.

SEE ALSO

close(2), cuserid(3S), ctermid(3S), fclose(3S), ferror(3S), fopen(3S), fread(3S), fseek(3S), getc(3S), gets(3S), lseek(2), open(2), pipe(2), popen(3S), printf(3S), putc(3S), puts(3S), read(2), scanf(3S), setbuf(3S), system(3S), tmpfile(3S), tmpnam(3S), ungetc(3S), write(2)

DIAGNOSTICS

Invalid *stream* pointers 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 it is necessary for each system to define standards for forming them. If some standard is not adhered to, it is possible for unrelated processes unintentionally to interfere with each other's operation. Therefore, it is strongly suggested that the most significant byte of a key in some sense refer to a project so that keys do not conflict across a given system.

ftok returns a key based on *path* and *id* that is usable in subsequent *msgget*, *semget*, and *shmget* system calls. *path* must be the path name of an existing file that is accessible to the process. *id* is a character which uniquely identifies a project. Note that *ftok* returns the same key for linked files when called with the same *id* and that it returns 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* return an error. If the same file is recreated, then *ftok* is likely to return a different key than it did the original time it was called.

NAME

string: strcat, strdup, strncat, strcmp, strncmp, strcasecmp, strncasecmp, 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;

strcasecmp(s1, s2)
char *s1, *s2;

strncasecmp(s1, s2, count)
char *s1, *s2;
int count;

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*.

strcat appends a copy of string *s2* to the end of string *s1*.

strdup 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.

strncat appends at most *n* characters. Each returns a pointer to the null-terminated result.

strcmp compares its arguments and returns an integer less than, equal to, or greater than 0, according as *s1* is lexicographically less than, equal to, or greater than *s2*. *strncmp* makes the same comparison but looks at at most *n* characters. *strcasecmp* and *strncasecmp* are identical in function, but are case insensitive. The returned lexicographic difference reflects a conversion to lower-case.

strcpy copies string *s2* to *s1*, stopping after the null character has been copied. *strncpy* copies exactly *n* characters, truncating *s2* or adding null characters to *s1* if necessary. The result is not null-terminated if the length of *s2* is *n* or more. Each function returns *s1*.

strlen returns the number of characters in *s*, not including the terminating null character.

strchr (*strrchr*) returns a pointer to the first (last) occurrence of character *c* in string *s*, or a NULL pointer if *c* does not occur in the string. The null character terminating a string is considered to be part of the string.

strpbrk returns a pointer to the first occurrence in string *s1* of any character from string *s2*, or a NULL pointer if no character from *s2* exists in *s1*.

strspn (*strcspn*) returns the length of the initial segment of string *s1* which consists entirely of characters from (not from) string *s2*.

strtok considers the string *s1* to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *s2*. The first call (with pointer *s1* specified) returns a pointer to the first character of the first token, and will have written a null character into *s1* immediately following the returned token. The function keeps track of its position in the string between separate calls, so that subsequent calls (which must be made with the first argument a NULL pointer) works through the string *s1* immediately following that token. In this way subsequent calls work through the string *s1* until no tokens remain. The separator string *s2* may be different from call to call. When no token remains in *s1*, a NULL pointer is returned.

For user convenience, all these functions are declared in the optional *<string.h>* header file.

SEE ALSO

malloc(3C), malloc(3X)

CAVEATS

strcmp and *strncmp* are implemented by using the most natural character comparison on the machine. Thus the sign of the value returned when one of the characters has its high-order bit set is not the same in all implementations and should not be relied upon.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

NAME

strtod, atof – convert string to double-precision number

SYNOPSIS

double strtod (str, ptr)
char *str, **ptr;

double atof (str)
char *str;

DESCRIPTION

strtod returns as a double-precision floating-point number the value represented by the character string pointed to by *str*. The string is scanned up to the first unrecognized character.

strtod recognizes an optional string of "white-space" characters [as defined by *isspace* in *ctype*(3C)], then an optional sign, then a string of digits optionally containing a decimal point, then an optional **e** or **E** followed by an optional sign or space, followed by an integer.

If the value of *ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no number can be formed, **ptr* is set to *str*, and zero is returned.

atof(str) is equivalent to *strtod(str, (char **)NULL)*.

SEE ALSO

ctype(3C), scanf(3S), strtol(3C)

DIAGNOSTICS

If the correct value would cause overflow, ±HUGE (as defined in <*math.h*>) is returned (according to the sign of the value), and *errno* is set to ERANGE*S.
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

strtol 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 thusly: After an optional leading sign a leading zero indicates octal conversion, and a leading "0x" or "0X" hexadecimal conversion. Otherwise, decimal conversion is used.

Truncation from long to int can, of course, take place upon assignment or by an explicit cast.

atol(str) is equivalent to *strtol(str, (char **)NULL, 10)*.

atoi(str) is equivalent to *(int) strtol(str, (char **)NULL, 10)*.

SEE ALSO

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

swab copies *nbytes* bytes pointed to by *from* to the array pointed to by *to*, exchanging adjacent even and odd bytes. *nbytes* should be even and non-negative. If *nbytes* is odd and positive *swab* uses *nbytes*–1 instead. If *nbytes* is negative, *swab* does nothing.

NAME

system – issue a shell command

SYNOPSIS

#include <stdio.h>

int system (string)
char *string;

DESCRIPTION

system causes the *string* to be given to *sh*(1) as input, as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status of the shell.

FILES

/bin/sh

SEE ALSO

exec(2), sh(1)

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

tablet – subroutines for accessing the tablet

SYNOPSIS

fd = OpenTablet();
success = ReadTablet(block, last_flag)
int block[3];

DESCRIPTION

These routines should be used to open and read the tablet. *OpenTablet* returns a file descriptor appropriate for *poll*(2). Error return is indicated by a –1 value, and *errno* will be valid.

Each call to *ReadTablet* returns a one on successful read of the tablet. It will wait for one sample to arrive if there are none waiting at the time of the call. A zero return is possible if the tablet gets out of synchronization with the system.

The *block* argument is a three integer array: the first returned integer is zero or one, if the pen is up or down respectively; the second returned integer is the *X* position of the pen; and, the third returned integer is the *Y* position of the pen. The *X* and *Y* positions are in units of .001 inch, and range up to about 11500 in *X* and 8500 in *Y*.

The *last_flag* argument directs *ReadTablet* to look ahead and return the last sample (if True), or to simply return the next sample (if False). If *last_flag* is false, the application should read all pending samples itself (use poll(2)) quickly otherwise the tablet input will overrun the available input buffer and samples may be lost.

The tablet is set up in incremental run mode where samples are generated continuously while the sample data is changing (either from stylus motion or up/down changes). Samples are generated at a maximum rate of 50 per second.

SEE ALSO

poll(2), dials(3)

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 is deleted automatically when the process using it terminates. The file is opened for update ("w+").

SEE ALSO

creat(2), fopen(3S), mktemp(3C), perror(3C), stdio(3S), tmpnam(3S), unlink(2)

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* destroys the contents of the area. If *s* is not NULL, it is assumed to be the address of an array of at least **L_tmpnam** bytes, where **L_tmpnam** is a constant defined in *<stdio.h>*; *tmpnam* places its result in that array and returns *s*.

tempnam allows 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** is used as a last resort. This entire sequence can be up-staged by providing an environment variable TMPDIR in the user's environment, whose value is the name of the desired temporary-file directory.

Many applications prefer their temporary files to have certain favorite initial letter sequences in their names. Use the *pfx* argument for this. This argument may be NULL or point to a string of up to five characters to be used as the first few characters of the temporary-file name.

tempnam uses *malloc*(3C) to get space for the constructed file name, and returns a pointer to this area. Thus, any pointer value returned from *tempnam* may serve as an argument to *free* [see *malloc*(3C)]. If *tempnam* cannot return the expected result for any reason, i.e. *malloc*(3C) failed, or none of the above mentioned attempts to find an appropriate directory was successful, a NULL pointer is returned.

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) to remove the file when its use is ended.

SEE ALSO

creat(2), fopen(3S), malloc(3C), mktemp(3C), tmpfile(3S), unlink(2)

CAVEATS

If called more than 17,576 times in a single process, these functions starts 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.

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 may be contained in the elements in addition to the values being compared.

tsearch is used to build and access the tree. *key* is a pointer to a datum to be accessed or stored. If there is a datum in the tree equal to *key (the value pointed to by key), a pointer to this found datum is returned. Otherwise, *key is inserted, and a pointer to it returned. Only pointers are copied, so the calling routine must store the data. *rootp* points to a variable that points to the root of the tree. A **NULL** value for the variable pointed to by *rootp* denotes an empty tree; in this case, the variable is set to point to the datum at the root of the new tree.

Like *tsearch, tfind* searches for a datum in the tree, returning a pointer to it if found. However, if the datum is not found, *tfind* returns a **NULL** pointer. The arguments for *tfind* are the same as for *tsearch*.

tdelete deletes a node from a binary search tree. The arguments are the same as for *tsearch*. The variable pointed to by *rootp* are changed if the deleted node was the root of the tree. *tdelete* returns a pointer to the parent of the deleted node, or a **NULL** pointer if the node is not found.

twalk traverses a binary search tree. *root* is the root of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) *action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The second argument is a value from an enumeration data type *typedef enum { preorder, postorder, endorder, leaf } VISIT;* (defined in the *<search.h>* header file), depending on whether this is the first, second or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level zero.

The pointers to the key and the root of the tree should be of type pointer-to-element, and cast to type pointer-to-character. Similarly, although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

EXAMPLE

The following code reads in strings and stores structures containing a pointer to each string and a count of its length. It then walks the tree, printing out the stored strings and their lengths in alphabetical order.

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

struct node {           /* pointers to these are stored in the tree */
        char *string;
        int length;
};
char string_space[10000];       /* space to store strings */
struct node nodes[500]; /* nodes to store */
struct node *root = NULL;       /* this points to the root */

main( )
{
        char *strptr = string_space;
        struct node *nodeptr = nodes;
        void print_node( ), twalk( );
        int i = 0, node_compare( );

        while (gets(strptr) != NULL && i++ < 500)  {
                /* set node */
                nodeptr->string = strptr;
                nodeptr->length = strlen(strptr);
                /* put node into the tree */
                (void) tsearch((char *)nodeptr, (char **) &root,
                        node_compare);
                /* adjust pointers, so we don't overwrite tree */
                strptr += nodeptr->length + 1;
                nodeptr++;
        }
        twalk((char *)root, print_node);
}
/*
        This routine compares two nodes, based on an
        alphabetical ordering of the string field.
*/
int
node_compare(node1, node2)
char *node1, *node2;
{
        return strcmp(((struct node *)node1)->string,
        ((struct node *) node2)->string);
}
/*
        This routine prints out a node, the first time
        twalk encounters it.
*/
void
print_node(node, order, level)
char **node;
VISIT order;
```

```
int level;
{
        if (order == preorder || order == leaf) {
                (void)printf("string = %20s,  length = %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.

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 respectively 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.

CAVEAT

If the calling function alters the pointer to the root, results are unpredictable.

NAME

ttyname, isatty – find name of a terminal

SYNOPSIS

char *ttyname (fildes)
int fildes;

int isatty (fildes)
int fildes;

DESCRIPTION

ttyname returns a pointer to a string containing the null-terminated path name of the terminal device associated with file descriptor *fildes*.

isatty returns 1 if *fildes* is associated with a terminal device, 0 otherwise.

FILES

/dev/*

DIAGNOSTICS

ttyname returns a NULL pointer if *fildes* does not describe a terminal device in directory */dev*.

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).

NAME

asinh, acosh, atanh – inverse hyperbolic functions

SYNOPSIS

```
#include <math.h>

double asinh(x)
double x;

double acosh(x)
double x;

double atanh(x)
double x;
```

DESCRIPTION

These functions compute the designated inverse hyperbolic functions for real arguments.

ERROR (due to Roundoff etc.)

On a VAX, acosh is accurate to about 3 *ulp*s, asinh and atanh to about 2 *ulp*s. An *ulp* is one *U*nit in the *L*ast *P*lace carried.

DIAGNOSTICS

acosh returns the reserved operand on a VAX if the argument is less than 1.

atanh returns the reserved operand on a VAX if the argument has absolute value bigger than or equal to 1.

SEE ALSO

exp(3M)

AUTHOR

W. Kahan, Kwok–Choi Ng

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*.

j0 and *y1* return Bessel functions of *x* of the second kind of orders 0 and 1 respectively. *jn* returns the Bessel function of *x* of the second kind of order *n*. The value of *x* must be positive.

DIAGNOSTICS

Non-positive arguments cause *y0*, *y1* and *yn* to return the value **–Infinity**.

Arguments too large in magnitude cause *j0*, *j1*, *y0* and *y1* to return zero.

NAME

erf, erfc – error function and complementary error function

SYNOPSIS

```
#include <math.h>

double erf (x)
double x;

double erfc (x)
double x;
```

DESCRIPTION

erf returns the error function of x, defined as $\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt.$

erfc, which returns 1.0 – *erf(x)*, is provided because of the extreme loss of relative accuracy if *erf(x)* is called for large x and the result subtracted from 1.0. For example, if $x = 5$, 12 places are lost.

SEE ALSO

exp(3M)

NAME

exp, log, log10, pow, sqrt, cbrt – 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;

double cbrt (x)
double x;

DESCRIPTION

exp returns e^x.

log returns the natural logarithm of x. The value of x must be positive.

log10 returns the logarithm base ten of x. The value of x must be positive.

pow returns x^y. If x is zero, y must be positive. If x is negative, y must be an integer.

sqrt returns the non-negative square root of x. The value of x may not be negative.

cbrt returns the cube root of x.

SEE ALSO

hypot(3M), sinh(3M)

DIAGNOSTICS

exp returns *Infinity* when the correct value would overflow, or 0 when the correct value would underflow.

log and *log10* return *NaN* when x is non-positive.

pow returns *NaN* when x is 0 and y is non-positive, or when x is negative and y is not an integer. When the correct value for *pow* would overflow or underflow, *pow* returns *Infinity* or 0 respectively

sqrt and *cbrt* return *NaN*.

NAME

floor, ceil, fmod, fabs – floor, ceiling, remainder, absolute value functions

SYNOPSIS

#include <math.h>

double floor (x)
double x;

double ceil (x)
double x;

double fmod (x, y)
double x, y;

double fabs (x)
double x;

DESCRIPTION

floor returns the largest integer (as a double-precision number) not greater than x.

ceil returns the smallest integer not less than x.

fmod returns the floating-point remainder of the division of x by y: zero 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 $| f | < | y |$.

fabs returns the absolute value of x, $| x |$.

SEE ALSO

abs(3C)

NAME

gamma – log gamma function

SYNOPSIS

#include <math.h>

double gamma (x)
double x;

extern int signgam;

DESCRIPTION

gamma returns ln(| Γ(x) |), where Γ(x) is defined as $\int_0^\infty e^{-t} t^{x-1} dt$. The sign of Γ(x) is returned in the external integer *signgam*. The argument x may not be a non-positive integer.

The following C program fragment might be used to calculate Γ:

```
if ((y = gamma(x)) > LN_MAXDOUBLE)
        error( );
y = signgam * exp(y);
```

where LN_MAXDOUBLE is the least value that causes *exp*(3M) to return a range error, and is defined in the <*values.h*> header file.

SEE ALSO

exp(3M), values(5)

DIAGNOSTICS

For non-negative integer arguments *Infinity* is returned.

If the correct value would overflow, *gamma* returns *Infinity*

NAME

hypot, cabs – Euclidean distance, complex absolute value

SYNOPSIS

#include <math.h>

double hypot(x,y)
double x,y;

double cabs(z)
struct {double x,y;} z;

DESCRIPTION

hypot(x,y) and *cabs*(x,y) return *sqrt(x*x+y*y)* computed in such a way that underflow will not happen, and overflow occurs only if the final result deserves it.

hypot(∞,v) = hypot(v,∞) = +∞ for all v, including *NaN*.

ERROR (due to Roundoff, etc.)

Below 0.97 *ulp*s. Consequently *hypot*(5.0,12.0) = 13.0 exactly; in general, *hypot* and *cabs* return an integer whenever an integer might be expected.

The same cannot be said for the shorter and faster version of *hypot* and *cabs* that is provided in the comments in cabs.c; its error can exceed 1.2 *ulp*s.

NOTES

As might be expected, *hypot*(v,*NaN*) and *hypot(NaN*,v) are *NaN* for all *finite* v; with "reserved operand" in place of "*NaN*", the same is true on a VAX. But programmers on machines other than a VAX (it has no ∞) might be surprised at first to discover that *hypot*(±∞,*NaN*) = +∞. This is intentional; it happens because *hypot*(∞,v) = +∞ for *all* v, finite or infinite. Hence *hypot*(∞,v) is independent of v. Unlike the reserved operand on a VAX, the IEEE *NaN* is designed to disappear when it turns out to be irrelevant, as it does in *hypot*(∞,*NaN*).

SEE ALSO

trig(3M)

AUTHOR

W. Kahan

NAME

lgamma – log gamma function

SYNOPSIS

#include <math.h>

double lgamma(x)
double x;

DESCRIPTION

lgamma
returns ln $|\Gamma(x)|$ where $\Gamma(x) = \int_0^\infty t^{x-1} e^{-t}\,dt$ for $x > 0$ and
$\Gamma(x) = \pi/(\Gamma(1-x)\sin(\pi x))$ for $x < 1$.

The external integer signgam returns the sign of $\Gamma(x)$.

IDIOSYNCRASIES

Do **not** use the expression signgam*exp(lgamma(x)) to compute g := $\Gamma(x)$. Instead use a program like this (in C): lg = lgamma(x); g = signgam*exp(lg);

Only after lgamma has returned can signgam be correct. Note too that $\Gamma(x)$ must overflow when x is large enough, underflow when –x is large enough, and spawn a division by zero when x is a nonpositive integer.

Only in the UNIX math library for C was the name gamma ever attached to $\ln\Gamma$. Elsewhere, for instance in IBM's FORTRAN library, the name GAMMA belongs to Γ and the name ALGAMA to $\ln\Gamma$ in single precision; in double the names are DGAMMA and DLGAMA. Why should C be different?

Archaeological records suggest that C's gamma originally delivered $\ln(\Gamma(|x|))$. Later, the program gamma was changed to cope with negative arguments x in a more conventional way, but the documentation did not reflect that change correctly. The most recent change corrects inaccurate values when x is almost a negative integer, and lets $\Gamma(x)$ be computed without conditional expressions. Programmers should not assume that lgamma has settled down.

At some time in the future, the name *gamma* will be rehabilitated and used for the gamma function, just as is done in FORTRAN. The reason for this is not so much compatibility with FORTRAN as a desire to achieve greater speed for smaller values of $|x|$ and greater accuracy for larger values.

DIAGNOSTICS

The reserved operand is returned on a VAX for negative integer arguments, *errno* is set to ERANGE; for very large arguments over/underflows will occur inside the *lgamma* routine.

SEE ALSO

gamma(3M)

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 arguments.

DIAGNOSTICS

sinh and *cosh* return *Infinity* (and *sinh* may return *–Infinity* for negative *x*) when the correct value would overflow.

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.

DIAGNOSTICS

sin, *cos*, and *tan* lose accuracy when their argument is far from zero.

If the magnitude of the argument of *asin* or *acos* is greater than one, or if both arguments of *atan2* are zero, *NaN* is returned.

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, stops 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

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

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

key is a user's typed password. *salt* is a two-character string chosen from the set [**a-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 *edflag* is zero, the argument is encrypted; if non-zero, it is decrypted.

Note that decryption is not provided in the international version of *crypt*(3X). The international version is part of the *C Programming Language Utilities*, and the domestic version is part of the *Security Administration Utilities*. If decryption is attempted with the international version of *des_encrypt*, an error message is printed.

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

The routines *run_setkey* and *run_crypt* are designed for use by applications that need cryptographic capabilities [such as *ed*(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. *run_crypt* takes a block of characters and transforms the cleartext or ciphertext into their ciphertext or cleartext using *crypt*(1). *offset* 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).

run_setkey returns -1 if a connection with *crypt*(1) cannot be established. This will occur on international versions of UNIX where *crypt*(1) is not available. If a null key is passed to *run_setkey*, 0 is returned. Otherwise, 1 is returned. *run_crypt* returns –1 if it cannot write output or read input from the pipe attached to *crypt*. Otherwise it returns 0.

DIAGNOSTICS

In the international version of *crypt*(3X), a flag argument of 1 to *des_encrypt* is not accepted, and an error message is printed.

SEE ALSO

crypt(1), getpass(3C), login(1), passwd(1), passwd(4)

CAVEAT

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

NAME

curses – terminal screen handling and optimization package

SYNOPSIS

The *curses* manual page is organized as follows:

In SYNOPSIS
- compiling information
- summary of parameters used by *curses* routines
- alphabetical list of curses routines, showing their parameters

In DESCRIPTION:
- An overview of how *curses* routines should be used

In ROUTINES, descriptions of each *curses* routines, are grouped under the appropriate topics:
- Overall Screen Manipulation
- Window and Pad Manipulation
- Output
- Input
- Output Options Setting
- Input Options Setting
- Environment Queries
- Soft Labels
- Low-level Curses Access
- Terminfo-Level Manipulations
- Termcap Emulation
- Miscellaneous
- Use of **curscr**

Then come sections on:
- ATTRIBUTES
- FUNCTION CALLS
- LINE GRAPHICS

cc [flag ...] file ... **–lcurses** [library ...]

#include <curses.h> (automatically includes <*stdio.h*>, <*termio.h*>, and <*unctrl.h*>).

The parameters in the following list are not global variables, but rather this is a summary of the parameters used by the *curses* library routines. All routines return the **int** values **ERR** or **OK** unless otherwise noted. Routines that return pointers always return **NULL** on error. (**ERR**, **OK**, and **NULL** are all defined in <*curses.h*>.) Routines that return integers are not listed in the parameter list below.

bool bf

char **area,*boolnames[ ], *boolcodes[ ], *boolfnames[ ], *bp
char *cap, *capname, codename[2], erasechar, *filename, *fmt
char *keyname, killchar, *label, *longname
char *name, *numnames[ ], *numcodes[ ], *numfnames[ ]
char *slk_label, *str, *strnames[ ], *strcodes[ ], *strfnames[ ]
char *term, *tgetstr, *tigetstr, *tgoto, *tparm, *type

chtype attrs, ch, horch, vertch

FILE *infd, *outfd

int begin_x, begin_y, begline, bot, c, col, count
int dmaxcol, dmaxrow, dmincol, dminrow, *errret, fildes
int (*init( )), labfmt, labnum, line
int ms, ncols, new, newcol, newrow, nlines, numlines
int oldcol, oldrow, overlay
int p1, p2, p9, pmincol, pminrow, (*putc( )), row
int smaxcol, smaxrow, smincol, sminrow, start
int tenths, top, visibility, x, y

SCREEN *new, *newterm, *set_term

TERMINAL *cur_term, *nterm, *oterm

va_list varglist

WINDOW *curscr, *dstwin, *initscr, *newpad, *newwin, *orig
WINDOW *pad, *srcwin, *stdscr, *subpad, *subwin, *win

addch(ch)
addstr(str)
attroff(attrs)
attron(attrs)
attrset(attrs)
baudrate( )
beep( )
box(win, vertch, horch)
cbreak( )
clear( )
clearok(win, bf)
clrtobot( )
clrtoeol( )
copywin(srcwin, dstwin, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol, overlay)
curs_set(visibility)
def_prog_mode( )
def_shell_mode( )
del_curterm(oterm)
delay_output(ms)
delch( )
deleteln( )
delwin(win)
doupdate( )
draino(ms)
echo( )
echochar(ch)
endwin( )
erase( )
erasechar( )
filter( )
flash( )
flushinp( )
garbagedlines(win, begline, numlines)
getbegyx(win, y, x)
getch( )
getmaxyx(win, y, x)
getstr(str)
getsyx(y, x)

getyx(win, y, x)
halfdelay(tenths)
has_ic( )
has_il( )
idlok(win, bf)
inch( )
initscr( )
insch(ch)
insertln( )
intrflush(win, bf)
isendwin( )
keyname(c)
keypad(win, bf)
killchar( )
leaveok(win, bf)
longname( )
meta(win, bf)
move(y, x)
mvaddch(y, x, ch)
mvaddstr(y, x, str)
mvcur(oldrow, oldcol, newrow, newcol)
mvdelch(y, x)
mvgetch(y, x)
mvgetstr(y, x, str)
mvinch(y, x)
mvinsch(y, x, ch)
mvprintw(y, x, fmt [, arg . . .])
mvscanw(y, x, fmt [, arg . . .])
mvwaddch(win, y, x, ch)
mvwaddstr(win, y, x, str)
mvwdelch(win, y, x)
mvwgetch(win, y, x)
mvwgetstr(win, y, x, str)
mvwin(win, y, x)
mvwinch(win, y, x)
mvwinsch(win, y, x, ch)
mvwprintw(win, y, x, fmt [, arg . . .])
mvwscanw(win, y, x, fmt [, arg . . .])
napms(ms)
newpad(nlines, ncols)
newterm(type, outfd, infd)
newwin(nlines, ncols, begin_y, begin_x)
nl( )
nocbreak( )
nodelay(win, bf)
noecho( )
nonl( )
noraw( )
notimeout(win, bf)
overlay(srcwin, dstwin)
overwrite(srcwin, dstwin)
pechochar(pad, ch)
pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, 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)
slk_clear( )
slk_init(fmt)
slk_label(labnum)
slk_noutrefresh( )
slk_refresh( )
slk_restore( )
slk_set(labnum, label, fmt)
slk_touch( )
standend( )
standout( )
subpad(orig, nlines, ncols, begin_y, begin_x)
subwin(orig, nlines, ncols, begin_y, begin_x)
tgetent(bp, name)
tgetflag(codename)
tgetnum(codename)
tgetstr(codename, area)
tgoto(cap, col, row)
tigetflag(capname)
tigetnum(capname)
tigetstr(capname)
touchline(win, start, count)
touchwin(win)
tparm(str, p1, p2, . . ., p9)
tputs(str, count, putc)
traceoff( )
traceon( )
typeahead(fildes)
unctrl(c)
ungetch(c)
vidattr(attrs)
vidputs(attrs, putc)
vwprintw(win, fmt, varglist)
vwscanw(win, fmt, varglist)

waddch(win, ch)
waddstr(win, str)
wattroff(win, attrs)
wattron(win, attrs)
wattrset(win, attrs)
wclear(win)
wclrtobot(win)
wclrtoeol(win)
wdelch(win)
wdeleteln(win)
wechochar(win, ch)
werase(win)
wgetch(win)
wgetstr(win, str)
winch(win)
winsch(win, ch)
winsertln(win)
wmove(win, y, x)
wnoutrefresh(win)
wprintw(win, fmt [, arg...])
wrefresh(win)
wscanw(win, fmt [, arg...])
wsetscrreg(win, top, bot)
wstandend(win)
wstandout(win)

DESCRIPTION

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

In order to initialize the routines, the routine *initscr*( ) or *newterm*( ) must be called before any of the other routines that deal with windows and screens are used. (Three exceptions are noted where they apply.) The routine *endwin*( ) must be called before exiting. To get character-at-a-time input without echoing, (most interactive, screen oriented programs want this) after calling *initscr*( ) you should call *"cbreak*( ); *noecho*( );" Most programs would additionally call *"nonl*( ); *intrflush (stdscr, FALSE); keypad(stdscr, TRUE);"*.

Before a *curses* program is run, a terminal's tab stops should be set and its initialization strings, if defined, must be output. This can be done by executing the **tput init** command after the shell environment variable **TERM** has been exported. For further details, see *profile*(4), *tput*(1), and the "Tabs and Initialization" subsection of *terminfo*(4).

The *curses* library contains routines that manipulate data structures called *windows* that can be thought of as two-dimensional arrays of characters representing all or part of a terminal screen. A default window called **stdscr** is supplied, which is the size of the terminal screen. Others may be created with *newwin*( ). Windows are referred to by variables declared as **WINDOW** *; the type **WINDOW** is defined in <*curses.h*> to be a C structure. These data structures are manipulated with routines described below, among which the most basic are *move*( ) and *addch*( ). (More general versions of these routines are included with names beginning with **w**, allowing you to specify a window. The routines not beginning with **w** usually affect **stdscr**.) Then *refresh*( ) is called, telling the routines to make the user's terminal screen look like **stdscr**. The characters in a window are actually of type **chtype**, so that other information about the character may also be stored with each character.

Special windows called *pads* may also be manipulated. These are windows which are not constrained to the size of the screen and whose contents need not be displayed completely. See the description of *newpad*( ) under "Window and Pad Manipulation" for more information.

In addition to drawing characters on the screen, video attributes may be included which cause the characters to show up in modes such as underlined or in reverse video on terminals that support such display enhancements. Line drawing characters may be specified to be output. On input, *curses* is also able to translate arrow and function keys that transmit escape sequences into single values. The video attributes, line drawing characters, and input values use names, defined in <*curses.h*>, such as **A_REVERSE, ACS_HLINE,** and **KEY_LEFT.**

curses also defines the **WINDOW** * variable, **curscr**, which is used only for certain low-level operations like clearing and redrawing a garbaged screen. **curscr** can be used in only a few routines. If the window argument to *clearok*( ) is **curscr**, the next call to *wrefresh*( ) with any window causes the screen to be cleared and repainted from scratch. If the window argument to *wrefresh*( ) is **curscr**, the screen in immediately cleared and repainted from scratch. This is how most programs would implement a "repaint-screen" function. More information on using **curscr** is provided where its use is appropriate.

The environment variables **LINES** and **COLUMNS** may be set to override *terminfo*'s idea of how large a screen is. These may be used in an AT&T Teletype 5620 layer, for example, where the size of a screen is changeable.

If the environment variable **TERMINFO** is defined, any program using *curses* checks for a local terminal definition before checking in the standard place. For example, if the environment variable **TERM** is set to **att4425**, then the compiled terminal definition is found in */usr/lib/terminfo/a/att4425*. (The **a** is copied from the first letter of **att4425** to avoid creation of huge directories.) However, if **TERMINFO** is set to *$HOME/myterms*, *curses* first checks *$HOME/myterms/a/att4425*, and, if that fails, then checks */usr/lib/terminfo/a/att4425*. This is useful for developing experimental definitions or when write permission on */usr/lib/terminfo* is not available.

The integer variables **LINES** and **COLS** are defined in <*curses.h*>, and are filled in by *initscr*( ) with the size of the screen. (For more information, see the subsection "Terminfo-Level Manipulations".) The constants **TRUE** and **FALSE** hold 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 type **WINDOW** *.) Option-setting routines require a boolean flag *bf* with the value **TRUE** or **FALSE.** (*bf* is always type **bool.**) The types **WINDOW, bool,** and **chtype** are defined in <*curses.h*>. See the SYNOPSIS for a summary of variables types.

All routines return either the integer **ERR** or the integer **OK**, unless otherwise noted. Routines that return pointers always return **NULL** on error.

Overall Screen Manipulation

WINDOW ***initscr**()

The first routine called should almost always be *initscr*(). (The exceptions are *slk_init*(), *filter*(), and *ripoffline*().) This determines the terminal type and initializes all *curses* data structures. *initscr*() also arranges that the first call to *refresh*() clears the screen. If errors occur, *initscr*() writes an appropriate error message to standard error and exits; 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, move the cursor to the lower left corner of the screen and reset the terminal into the proper non-visual mode. To resume after a temporary escape, call *wrefresh*() or *doupdate*().

isendwin()

Returns TRUE if *endwin*() has been called without any subsequent calls to *wrefresh*().

SCREEN ***newterm**(type, outfd, infd)

A program that outputs to more than one terminal must use *newterm*() for each terminal instead of *initscr*(). A program that wants an indication of error conditions, so that it may continue to run in a line-oriented mode if the terminal cannot support a screen-oriented program, must also use this routine. *newterm*() should be called once for each terminal. It returns a variable of type SCREEN* that should be saved as a reference to that terminal. The arguments are the *type* of the terminal to be used in place of the environment variable TERM; *outfd*, a *stdio*(3S) file pointer for output to the terminal; and *infd*, another file pointer for input from the terminal. When it is done running, the program must also call *endwin*() for each terminal being used. If *newterm*() is called more than once for the same terminal, the first terminal referred to must be the last one for which *endwin*() is called.

SCREEN ***set_term**(new)

This routine is used to switch between different terminals. The screen reference *new* becomes the new current terminal. A pointer to the screen of the previous terminal is returned by the routine. This is the only routine which manipulates SCREEN pointers; all other routines affect only the current terminal.

Window and Pad Manipulation

refresh()
wrefresh (win)

These routines (or *prefresh*(), *pnoutrefresh*(), *wnoutrefresh*(), or *doupdate*()) must be called to write output to the terminal, as most other routines merely manipulate data structures. *wrefresh*() copies the named window to the physical terminal screen, taking into account what is already there in order to minimize the amount of information that's sent to the terminal (called optimization). *refresh*( ) does the same thing, except it uses **stdscr** as a default window. Unless *leaveok*( ) has been enabled, the physical cursor of the terminal is left at the location of the window's cursor. The number of characters output to the terminal is returned.

Note that *refresh*( ) is a macro.

wnoutrefresh(win)
doupdate( )

These two routines allow multiple updates to the physical terminal screen with more efficiency than *wrefresh*( ) alone.

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)

Create and return a pointer to a new window with the given number of lines (or rows), *nlines*, and columns, *ncols*. The upper left corner of the window is at line *begin_y*, column *begin_x*. If either *nlines* or *ncols* is **0**, they are set to the value of **lines**–*begin_y* and **cols**–*begin_x*. A new full-screen window is created by calling *newwin(0,0,0,0)*.

mvwin(win, y, x)

Move the window so that the upper left corner is positioned at (*y*, *x*). If the move would cause the window to be off the screen, it is an error and the window is not moved.

WINDOW ***subwin**(orig, nlines, ncols, begin_y, begin_x)

Create and return a pointer to a new window with the given number of lines (or rows), *nlines*, and columns, *ncols*. The window is at position (*begin_y*, *begin_x*) on the screen. (This position is relative to the screen, and not to the window *orig*.) The window is made in the middle of the window *orig*, so that changes made to one window affect both windows. When using this routine, often it is necessary to call *touchwin*( ) or *touchline*( ) on *orig* before calling *wrefresh*( ).

delwin(win)

Delete the named window, freeing up all memory associated with it. In the case of overlapping windows, subwindows should be deleted before the main window.

WINDOW ***newpad**(nlines, ncols)

Create and return a pointer to a new pad data structure with the given number of lines (or rows), *nlines*, and columns, *ncols*. A pad is a window that is not restricted by the screen size and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window appears on the screen at one time. Automatic refreshes of pads (e.g. from scrolling or echoing of input) do not occur. It is not legal to call *wrefresh*( ) with a pad as an argument; the routines *prefresh*( ) or *pnoutrefresh*( ) should be called instead. Note that these routines require additional parameters to specify the part of the pad to be displayed and the location on the screen to be used for display.

WINDOW ***subpad**(orig, nlines, ncols, begin_y, begin_x)

Create and return a pointer to a subwindow within a pad with the given number of lines (or rows), *nlines*, and columns, *ncols*. Unlike *subwin*( ), which uses screen coordinates, the window is at position (*begin_y, begin_x*) on the pad. The window is made in the middle of the window *orig*, so that changes made to one window affect both windows. When using this routine, often it is necessary to call *touchwin*( ) or *touchline*( ) on *orig* before calling *prefresh*( ).

prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)

These routines are analogous to *wrefresh*( ) and *wnoutrefresh*( ) except that pads, instead of windows, are involved. The additional parameters are needed to indicate what part of the pad and screen are involved. *pminrow* and *pmincol* specify the upper left corner, in the pad, of the rectangle to be displayed. *sminrow, smincol, smaxrow,* and *smaxcol* specify the edges, on the screen, of the rectangle to be displayed in. The lower right corner in the pad of the rectangle to be displayed is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of *pminrow, pmincol, sminrow,* or *smincol* are treated as if they were zero.

Output

These routines are used to "draw" text on windows.

addch(ch)
waddch(win, ch)
mvaddch(y, x, ch)
mvwaddch(win, y, x, ch)

The character *ch* is put into the window at the current cursor position of the window and the position of the window cursor is advanced. Its function is similar to that of *putchar* (see *putc*(3S)). At the right margin, an automatic newline is performed. At the bottom of the scrolling region, if *scrollok*( ) is enabled, the scrolling region 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 *clrtoeol*( ) before moving. Tabs are considered to be at every eighth column. If *ch* is another control character, it is drawn in the ^X notation. (Calling *winch*( ) after adding a control character does not return the control character, but instead returns the representation of the control character.)

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

Note that *ch* is actually of type **chtype**, not a character.

Note that *addch*(), *mvaddch*(), and *mvwaddch*(), are macros.

echochar(ch)
wechochar(win, ch)
pechochar(pad, ch)

These routines are functionally equivalent to a call to *addch*(ch) followed by a call to *refresh*(), a call to *waddch*(win, ch) followed by a call to *wrefresh*(win), or a call to *waddch*(pad, ch) followed by a call to *prefresh*(pad). The knowledge that only a single character is being output is taken into consideration and, for non-control characters, a considerable performance gain can be seen by using these routines instead of their equivalents. In the case of *pechochar*(), the last location of the pad on the screen is reused for the arguments to *prefresh*().

Note that *ch* is actually of type **chtype**, not a character.

Note that *echochar*() is a macro.

addstr(str)
waddstr(win, str)
mvwaddstr(win, y, x, str)
mvaddstr(y, x, str)

These routines write all the characters of the null-terminated character string *str* on the given window. This is equivalent to calling *waddch*() once for each character in the string. Note that *addstr*(), *mvaddstr*(), and *mvwaddstr*() are macros.

attroff(attrs)
wattroff(win, attrs)
attron(attrs)
wattron(win, attrs)
attrset(attrs)
wattrset(win, attrs)
standend()
wstandend(win)
standout()
wstandout(win)

These routines manipulate the current attributes of the named window. These attributes can be any combination of A_STANDOUT, A_REVERSE, A_BOLD, A_DIM, A_BLINK, A_UNDERLINE, and A_ALTCHARSET. These constants are defined in <**curses.h**> and can be combined with the C logical OR (|) operator.

The current attributes of a window are applied to all characters that are written into the window with *waddch*(). Attributes are a property of the character, and move with the character through any scrolling and insert/delete line/character operations. To the extent possible on the particular terminal, characters are displayed as the graphic rendition of the characters put on the screen.

attrset(attrs) sets the current attributes of the given window to *attrs*. *attroff*(attrs) turns off the named attributes without turning on or off any other attributes. *attron*(attrs) turns on the named attributes without affecting any others. *standout*() is the same as *attron*(*A_STANDOUT*). *standend*() is the same as *attrset (0),* that is, it turns off all attributes.

Note that *attrs* is actually of type **chtype**, not a character.

Note that *attroff*(), *attron*(), *attrset*(), *standend*(), and *standout*() are macros.

beep()
flash()

These routines are used to signal the terminal user. *beep*() 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 for the screen to be cleared completely on the next call to *wrefresh*() for that window, and repainted from scratch.

Note that *clear*() is a macro.

clrtobot()
wclrtobot(win)

All lines below the cursor in this window are erased. Also, the current line to the right of the cursor, inclusive, is erased.

Note that *clrtobot*() is a macro.

clrtoeol()
wclrtoeol(win)

The current line to the right of the cursor, inclusive, is erased.

Note that *clrtoeol*() is a macro.

delay_output(ms)

Insert a *ms* millisecond pause in the output. It is not recommended that this routine be used extensively, because padding characters are used rather than a processor pause.

delch()
wdelch(win)
mvdelch(y, x)

mvwdelch(win, y, x) The character under the cursor in the window is deleted. All characters to the right on the same line are moved to the left one position and the last character on the line is filled with a blank. The cursor position does not change (after moving to (*y*, *x*), if specified). (This does not imply use of the hardware "delete-character" feature.)

Note that *delch*( ), *mvdelch*( ), and *mvwdelch*( ) are macros.

deleteln( )
wdeleteln(win) The line under the cursor in the window is deleted. All lines below the current line are moved up one line. The bottom line of the window is cleared. The cursor position does not change. (This does not imply use of the hardware "delete-line" feature.)

Note that *deleteln*( ) is a macro.

getyx(win, y, x) The cursor position of the window is placed in the two integer variables *y* and *x*. This is implemented as a macro, so no "&" is necessary before the variables.

getbegyx(win, y, x)
getmaxyx(win, y, x) Like *getyx*( ), these routines store the current beginning coordinates and size of the specified window.

Note that *getbegyx*( ) and *getmaxyx*( ) are macros.

insch(ch)
winsch(win, ch)
mvwinsch(win, y, x, ch)
mvinsch(y, x, ch) The character *ch* is inserted before the character under the cursor. All characters to the right are moved one space to the right, possibly losing the rightmost character of the line. The cursor position does not change (after moving to (*y*, *x*), if specified). (This does not imply use of the hardware "insert-character" feature.)

Note that *ch* is actually of type **chtype**, not a character.

Note that *insch*( ), *mvinsch*( ), and *mvwinsch*( ) are macros.

insertln( )
winsertln(win) A blank line is inserted above the current line and the bottom line is lost. (This does not imply use of the hardware "insert-line" feature.)

Note that *insertln*( ) is a macro.

move(y, x)
wmove(win, y, x) The cursor associated with the window is moved to line (row) *y*, column *x*. This does not move the physical cursor of the terminal until **refresh**() is called. The position specified is relative to the upper left corner of the window, which is (**0**, **0**).

Note that *move*( ) is a macro.

overlay(srcwin, dstwin)
overwrite(srcwin, dstwin)
These routines overlay *srcwin* on top of *dstwin*; that is, all text in *srcwin* is copied into *dstwin*. *scrwin* and *dstwin* need not be

the same size; only text where the two windows overlap is copied. The difference is that *overlay*() is non-destructive (blanks are not copied), while *overwrite*() is destructive.

copywin(srcwin, dstwin, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol, overlay)
This routine provides a finer grain of control over the *overlay*() and *overwrite*() routines. Like in the *prefresh*() routine, a rectangle is specified in the destination window, (*dminrow*, *dmincol*) and (*dmaxrow*, *dmaxcol*), and the upper-left-corner coordinates of the source window, (*sminrow*, *smincol*). If the argument *overlay* is true, then copying is non-destructive, as in *overlay*().

printw(fmt [, arg . . .])
wprintw(win, fmt [, arg . . .])
mvprintw(y, x, fmt [, arg . . .])
mvwprintw(win, y, x, fmt [, arg . . .])
These routines are analogous to *printf*(3). The string which would be output by *printf*(3) is instead output using *waddstr*() on the given window.

vwprintw(win, fmt, varglist)
This routine corresponds to *vfprintf*(3S). It performs a *wprintw*() using a variable argument list. The third argument is a *va_list*, a pointer to a list of arguments, as defined in *<varargs.h>*. See the *vprintf*(3S) and *varargs*(5) manual pages for a detailed description on how to use variable argument lists.

scroll(win)
The window is scrolled up one line. This involves moving the lines in the window data structure. As an optimization, if the window is **stdscr** and the scrolling region is the entire window, the physical screen 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 do 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 occurs 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. No *refresh*() occurs between the *move*() and the *getch*() done within the routines *mvgetch*() and *mvwgetch*().

When using *getch*(), *wgetch*(), *mvgetch*(), or *mvwgetch*(), do not set both NOCBREAK mode (**nocbreak**()) and ECHO mode (**echo**()) at the same time. Depending on the state of the *tty*(7) driver when each character is typed, the program may produce undesirable results.

If *keypad*(win, **TRUE**) has been called, and a function key is pressed, the token for that function key 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 below.)

Note that *getch*(), *mvgetch*(), and *mvwgetch*() are macros.

getstr(str)
wgetstr(win, str)
mvgetstr(y, x, str)
mvwgetstr(win, y, x, str)

A series of calls to *getch*() is made, until a newline, carriage return, or enter key is received. The resulting value is placed in the area pointed at by the character pointer *str*. The user's erase and kill characters are interpreted. As in *mvgetch*(), no *refresh*() is done between the *move*() and *getstr*() within the routines *mvgetstr*() and *mvwgetstr*().

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

flushinp()
Throws away any typeahead that has been typed by the user and has not yet been read by the program.

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

inch()
winch(win)
mvinch(y, x)
mvwinch(win, y, x)
The character, of type **chtype**, at the current position in the named window is returned. If any attributes are set for that position, their values are OR'ed into the value returned. The predefined constants A_CHARTEXT and A_ATTRIBUTES, defined in <**curses.h**>, can be used with the C logical AND (&) operator to extract the character or attributes alone.

Note that *inch*(), *winch*(), *mvinch*(), and *mvwinch*() are macros.

scanw(fmt [, arg...])
wscanw(win, fmt [, arg...])
mvscanw(y, x, fmt [, arg...])
mvwscanw(win, y, x, fmt [, arg...])
These routines correspond to *scanf*(3S), as do their arguments and return values. *wgetstr*( ) is called on the window, and the resulting line is used as input for the scan.

vwscanw(win, fmt, ap)
This routine is similar to *vwprintw*( ) above in that performs a *wscanw*( ) using a variable argument list. The third argument is a *va_list*, a pointer to a list of arguments, as defined in <*varargs.h*>. See the *vprintf*(3S) and *varargs*(5) manual pages for a detailed description on how to use variable argument lists.

Output Options Setting

These routines set options within *curses* that deal with output. All options are initially FALSE, unless otherwise stated. It is not necessary to turn these options off before calling *endwin*( ).

clearok(win, bf) If enabled (*bf* is TRUE), the next call to *wrefresh*( ) with this window will clear the screen completely and redraw the entire screen from scratch. This is useful when the contents of the screen are uncertain, or in some cases for a more pleasing visual effect.

idlok(win, bf) If enabled (*bf* is TRUE), *curses* considers using the hardware "insert/delete-line" feature of terminals so equipped. If disabled (*bf* is FALSE), *curses* seldom uses this feature. (The "insert/delete-character" feature is always considered.) This option should be enabled only if your application needs "insert/delete-line", for example, for a screen editor. It is disabled by default because "insert/delete-line" tends to be visually annoying when used in applications where it isn't really needed. If "insert/delete-line" cannot be used, *curses* redraws the changed portions of all lines.

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*( ).)

nl( )
nonl( ) These routines control whether newline is translated into carriage return and linefeed on output, and whether return is translated into newline on input. Initially, the translations do occur. By disabling these translations using *nonl*( ), *curses* is able to make better use of the linefeed capability, resulting in faster cursor motion.

Input Options Setting

These routines set options within *curses* that deal with input. The options involve using *ioctl*(2) and therefore interact with *curses* routines. It is not necessary to turn these options off before calling *endwin*( ).

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* set CBREAK mode.

Note that *cbreak*( ) overrides *raw*( ). See *getch*( ) under "Input" for a discussion of how these routines interact with *echo*( ) and *noecho*( ).

echo( )
noecho( ) These routines control whether characters typed by the user are echoed by *getch*( ) as they are typed. Echoing by the tty driver is always disabled, but initially *getch*( ) is in ECHO mode, so characters typed are echoed. Authors of most interactive programs prefer to do their own echoing in a controlled area of the screen, or not to echo at all, so they disable echoing by calling *noecho*( ). See *getch*( ) under "Input" for a discussion of how these routines interact with *cbreak*( ) and *nocbreak*( ).

halfdelay(tenths) Half-delay mode is similar to CBREAK mode in that characters typed by the user are immediately available to the program. However, after blocking for *tenths* tenths of seconds, ERR will be returned if nothing has been typed. *tenths* must be a

	number between 1 and 255. Use *nocbreak*( ) to leave half-delay mode.
intrflush(win, bf)	If this option is enabled, when an interrupt key is pressed on the keyboard (interrupt, break, quit) all output in the tty driver queue is flushed, giving the effect of faster response to the interrupt, but causing *curses* to retain the wrong idea of what is on the screen. Disabling the option prevents the flush. The default for the option is inherited from the tty driver settings. The window argument is ignored.
keypad(win, bf)	This option enables the keypad of the user's terminal. If enabled, the user can press a function key (such as an arrow key) and *wgetch*( ) returns a single value representing the function key, as in KEY_LEFT. If disabled, *curses* does not treat function keys specially and the program needs to interpret the escape sequences itself. If the keypad in the terminal can be turned on (made to transmit) and off (made to work locally), turning on this option causes the terminal keypad to be turned on when *wgetch*( ) is called.
meta(win, bf)	If enabled, characters returned by *wgetch*( ) are transmitted with all 8 bits, instead of with the highest bit stripped. In order for *meta*( ) to work correctly, the **km** (has_meta_key) capability has to be specified in the terminal's **terminfo**(4) entry.
nodelay(win, bf)	This option causes *wgetch*( ) to be a non-blocking call. If no input is ready, *wgetch*( ) 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. RAW mode also causes 8-bit input and output. The behavior of the BREAK key depends on other bits in the *tty*(7) driver that are not set by *curses*.
typeahead(fildes)	*curses* does "line-breakout optimization" by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update is postponed until *refresh*( ) or *doupdate*( ) is called again. This allows faster response to commands typed in advance. Normally, the file descriptor for the input FILE pointer passed to *newterm*( ), or **stdin** in the case that *initscr*( ) was used, is used to do this typeahead checking. The *typeahead*( ) routine specifies that the file descriptor *fildes* is to be used to check for typeahead instead. If *fildes* is **–1**, then no typeahead checking is done. Note that *fildes* is a file descriptor, not a <*stdio.h*> FILE pointer.

Environment Queries

baudrate() Returns the output speed of the terminal. The number returned is in bits per second, for example, 9600, and is an integer.

char erasechar() The user's current erase character is returned.

has_ic() True if the terminal has insert- and delete-character capabilities.

has_il() True if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This might be used to check to see if it would be appropriate to turn on physical scrolling using *scrollok*().

char killchar() The user's current line-kill character is returned.

char *longname() This routine returns a pointer to a static area containing a verbose description of the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to *initscr*() or *newterm*(). The area is overwritten by each call to *newterm*() and is not restored by *set_term*(), so the value should be saved between calls to *newterm*() if *longname*() is going to be used with multiple terminals.

Soft Labels

If desired, *curses* manipulates the set of soft function-key labels that exist on many terminals. For those terminals that do not have soft labels, if you want to simulate them, *curses* takes over the bottom line of **stdscr**, reducing the size of **stdscr** and the variable LINES. *curses* standardizes on 8 labels of 8 characters each.

slk_init(labfmt) In order to use soft labels, this routine must be called before *initscr*() or *newterm*() is called. If winds up using a line from **stdscr** to emulate the soft labels, then *labfmt* determines how the labels are arranged on the screen. Setting *labfmt* to **0** indicates that the labels are to be arranged in a 3-2-3 arrangement; **1** asks for a 4-4 arrangement.

slk_set(labnum, label, labfmt)
labnum is the label number, from 1 to 8. *label* is the string to be put on the label, up to 8 characters in length. A NULL string or a NULL pointer puts up a blank label. *labfmt* is one of **0**, **1** or **2**, to indicate whether the label is to be left-justified, centered, or right-justified within the label.

slk_refresh()
slk_noutrefresh() These routines correspond to the routines *wrefresh*() and *wnoutrefresh*(). Most applications would use *slk_noutrefresh*() because a *wrefresh*() is most likely soon to follow.

char *slk_label(labnum)
The current label for label number *labnum*, with leading and trailing blanks stripped, is returned.

slk_clear() The soft labels are cleared from the screen.

slk_restore() The soft labels are restored to the screen after a *slk_clear*().

slk_touch() All of the soft labels are forced to be output the next time a *slk_noutrefresh*() is performed.

Low-Level *curses* Access

The following routines give low-level access to various *curses* functionality. These routines typically would be used inside of library routines.

def_prog_mode()
def_shell_mode()
	Save the current terminal modes as the "program" (in *curses*) or "shell" (not in *curses*) state for use by the *reset_prog_mode*() and *reset_shell_mode*() routines. This is done automatically by *initscr*().

reset_prog_mode()
reset_shell_mode()
	Restore the terminal to "program" (in *curses*) or "shell" (out of *curses*) state. These are done automatically by *endwin*() and *doupdate*() after an *endwin*(), so they normally would not be called.

resetty()
savetty()
	These routines save and restore the state of the terminal modes. *savetty*() saves the current state of the terminal in a buffer and *resetty*() restores the state to what it was at the last call to *savetty*().

getsyx(y, x)	The current coordinates of the virtual screen cursor are returned in *y* and *x*. Like *getyx*(), the variables *y* and *x* do not take an "&" before them. If *leaveok*() is currently **TRUE**, then **–1,–1** is returned. If lines may have been removed from the top of the screen using *ripoffline*() and the values are to be used beyond just passing them on to *setsyx*(), the value *y*+**stdscr–>_yoffset** should be used for those other uses. Note that *getsyx*() is a macro.
setsyx(y, x)	The virtual screen cursor is set to *y*, *x*. If *y* and *x* are both **–1**, then *leaveok*() 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 mess up the current position of the program's cursor. The library routine would call *getsyx*() at the beginning, do its manipulation of its own windows, do a *wnoutrefresh*() on its windows, call *setsyx*(), and then call *doupdate*().
ripoffline(line, init)	This routine provides access to the same facility that *slk_init*() uses to reduce the size of the screen. *ripoffline*() must be called before *initscr*() or *newterm*() is called. If *line* is positive, a line is removed from the top of **stdscr**; if negative, a line is removed from the bottom. When this is done inside *initscr*(), the routine *init*() is called with two arguments: a window pointer to the 1-line window that has been allocated and an integer with the number of columns in the window. Inside this initialization routine, the integer variables LINES and COLS (defined in <*curses.h*>) are not guaranteed to be accurate and *wrefresh*() or *doupdate*() must not be called. It is allowable to call *wnoutrefresh*() during the initialization routine. *ripoffline*() can be called up to five times before calling *initscr*() or *newterm*().

scr_dump(filename)	The current contents of the virtual screen are written to the file *filename*.
scr_restore(filename)	The virtual screen is set to the contents of *filename*, which must have been written using *scr_dump*(). The next call to *doupdate*() 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 is displaying 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.
curs_set(visibility)	The cursor is set to invisible, normal, or very visible for *visibility* equal to **0, 1** or **2**.
draino(ms)	Wait until the output has drained enough that it will only take *ms* more milliseconds to drain completely.

garbagedlines(win, begline, numlines)

This routine indicates to *curses* that a screen line is garbaged and should be thrown away before having anything written over the top of it. It could be used for programs such as editors which want a command to redraw just a single line. Such a command could be used in cases where there is a noisy communications line and redrawing the entire screen would be subject to even more communication noise. Just redrawing the single line gives some semblance of hope that it would show up unblemished. The current location of the window is used to determine which lines are to be redrawn.

napms(ms)	Sleep for *ms* milliseconds.

Terminfo-Level Manipulations

These low-level routines must be called by programs that need to deal directly with the *terminfo*(4) database to handle certain terminal capabilities, such as programming function keys. For all other functionality, *curses* routines are more suitable and their use is recommended.

Initially, *setupterm*() should be called. (Note that *setupterm*() is automatically called by *initscr*() and *newterm*().) This 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. Otherwise, the values for **lines** and **columns** specified in the *terminfo*(4) database are used.

The header files <**curses.h**> and <**term.h**> should be included, in this order, to get the definitions for these strings, numbers, and flags. Parameterized strings should be passed through *tparm*() to instantiate them. All *terminfo*(4) strings (including the output of *tparm*()) should be printed with *tputs*() or *putp*(). Before exiting, *reset_shell_mode*() should be called to restore the tty modes. Programs which use cursor addressing should output **enter_ca_mode** upon startup and should output **exit_ca_mode** before exiting (see *terminfo*(4)). (Programs desiring shell escapes should call *reset_shell_mode*() and output **exit_ca_mode** before the shell is called and should output **enter_ca_mode** and call *reset_prog_mode*() after returning from the shell. Note that this is different from the *curses* routines (see *endwin*()).

setupterm(term, fildes, errret)
: Reads in the *terminfo*(4) database, initializing the *terminfo*(4) structures, but does not set up the output virtualization structures used by *curses*. The terminal type is in the character string *term*; if *term* is NULL, the environment variable TERM is used. All output is sent to the file descriptor *fildes*. If *errret* is not NULL, then *setupterm*() returns **OK** or ERR and stores a status value in the integer pointed to by *errret*. A status of **1** in *errret* is normal, **0** means that the terminal could not be found, and **–1** means that the *terminfo*(4) database could not be found. If *errret* is NULL, *setupterm*() prints an error message upon finding an error and then exits. Thus, the simplest call is *setupterm ((char *)0, 1, (int *)0)*, which uses all the defaults.

  The *terminfo*(4) boolean, numeric and string variables are stored in a structure of type TERMINAL. After *setupterm*() returns successfully, the variable **cur_term** (of type TERMINAL *) is initialized with all of the information that the *terminfo*(4) boolean, numeric and string variables refer to. The pointer may be saved before calling *setupterm*() again. Further calls to *setupterm*() allocate new space rather than reuse the space pointed to by **cur_term**.

set_curterm(nterm)
: *nterm* is of type TERMINAL *. *set_curterm*() sets the variable **cur_term** to *nterm*, and makes all of the *terminfo*(4) boolean, numeric and string variables use the values from *nterm*.

del_curterm(oterm)
: *oterm* is of type TERMINAL *. *del_curterm*() frees the space pointed to by *oterm* and makes it available for further use. If *oterm* is the same as **cur_term**, then references to any of the *terminfo*(4) boolean, numeric and string variables thereafter may refer to invalid memory locations until another *setupterm*() has been called.

restartterm(term, fildes, errret)
: Like *setupterm*() after a memory restore.

char ***tparm**(str, p_1, p_2, ..., p_9)
: Instantiate the string *str* with parms p_i. A pointer is returned to the result of *str* with the parameters applied.

tputs(str, count, putc)
: Apply padding to the string *str* and output it. *str* must be a *terminfo*(4) string variable or the return value from *tparm*(), *tgetstr*(), *tigetstr*() or *tgoto*(). *count* is the number of lines affected, or **1** if not applicable. *putc*() is a *putchar*(3S)-like routine to which the characters are passed, one at a time.

putp(str)
: A routine that calls *tputs* (*str*, **1**, **putchar**()).

vidputs(attrs, putc)
: Output a string that puts the terminal in the video attribute mode *attrs*, which is any combination of the attributes listed below. The characters are passed to the *putchar*(3S)-like routine *putc*().

vidattr(attrs)	Like *vidputs*(), except that it outputs through *putchar*(3S).
mvcur(oldrow, oldcol, newrow, newcol)	Low-level cursor motion.

The following routines return the value of the capability corresponding to the *terminfo*(4) *capname* passed to them, such as **xenl**.

tigetflag(capname)	The value **–1** is returned if *capname* is not a boolean capability.
tigetnum(capname)	The value **–2** is returned if *capname* is not a numeric capability.
tigetstr(capname)	The value (char *) **–1** is returned if *capname* is not a string capability.

char *boolnames[ ], *boolcodes[ ], *boolfnames[ ]
char *numnames[ ], *numcodes[ ], *numfnames[ ]
char *strnames[ ], *strcodes[ ], *strfnames[ ]

These null-terminated arrays contain the *capnames*, the *termcap* codes, and the full C names, for each of the *terminfo*(4) variables.

Termcap Emulation

These routines are included as a conversion aid for programs that use the *termcap* library. Their parameters are the same and the routines are emulated using the *terminfo*(4) database.

tgetent(bp, name)	Look up *termcap* entry for *name*. The emulation ignores the buffer pointer *bp*.
tgetflag(codename)	Get the boolean entry for *codename*.
tgetnum(codes)	Get numeric entry for *codename*.
char ***tgetstr**(codename, area)	Return the string entry for *codename*. If *area* is not NULL, then also store it in the buffer pointed to by *area* and advance *area*. *tputs*() should be used to output the returned string.
char ***tgoto**(cap, col, row)	Instantiate the parameters into the given capability. The output from this routine is to be passed to *tputs*().
tputs(str, affcnt, putc)	See *tputs*() above, under "Terminfo-Level Manipulations".

Miscellaneous

traceoff() **traceon**()	Turn off and on debugging trace output when using the debug version of the *curses* library, */usr/lib/libdcurses.a*. This facility is available only to customers with a source license.
unctrl(c)	This macro expands to a character string which is a printable representation of the character *c*. Control characters are displayed in the ^X notation. Printing characters are displayed as is. *unctrl*() is a macro, defined in <**unctrl.h**>, which is automatically included by <*curses.h*>.
char ***keyname**(c)	A character string corresponding to the key *c* is returned.
filter()	This routine is one of the few that is to be called before *initscr*() or *newterm*() is called. It arranges things so that *curses* thinks that there is a 1-line screen. *curses* does not use any

terminal capabilities that assume knowledge of 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()withanywindow**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 are accessed.

Obsolete Calls

Various routines are provided to maintain compatibility in programs written for older versions of the curses library. These routines are all emulated as indicated below.

crmode()	Replaced by *cbreak()*.
fixterm()	Replaced by *reset_prog_mode()*.
gettmode()	A no-op.
nocrmode()	Replaced by *nocbreak()*.
resetterm()	Replaced by *reset_shell_mode()*.
saveterm()	Replaced by *def_prog_mode()*.
setterm()	Replaced by *setupterm()*.

ATTRIBUTES

The following video attributes, defined in <*curses.h*>, can be passed to the routines *attron()*, *attroff()*, and *attrset()*, or OR'ed with the characters passed to *addch()*.

A_STANDOUT	Terminal's best highlighting mode
A_UNDERLINE	Underlining
A_REVERSE	Reverse video
A_BLINK	Blinking
A_DIM	Half bright
A_BOLD	Extra bright or bold
A_ALTCHARSET	Alternate character set
A_CHARTEXT	Bit-mask to extract character (described under **winch()**)
A_ATTRIBUTES	Bit-mask to extract attributes (described under **winch()**)
A_NORMAL	Bit mask to reset all attributes off (for example: **attrset (A_NORMAL)**

FUNCTION-KEYS

The following function keys, defined in <*curses.h*>, might be returned by *getch()* if *keypad()* has been enabled. Note that not all of these may be supported on a particular terminal if the terminal does not transmit a unique code when the key is pressed or the definition for the key is not present in the *terminfo*(4) database.

Name	*Value*	*Key name*
KEY_BREAK	0401	break key (unreliable)
KEY_DOWN	0402	The four arrow keys ...
KEY_UP	0403	
KEY_LEFT	0404	
KEY_RIGHT	0405	...
KEY_HOME	0406	Home key (upward+left arrow)

KEY_BACKSPACE	0407	backspace (unreliable)
KEY_F0	0410	Function keys. Space for 64 keys is reserved.
KEY_F(n)	(KEY_F0+(n))	Formula for f_n.
KEY_DL	0510	Delete line
KEY_IL	0511	Insert line
KEY_DC	0512	Delete character
KEY_IC	0513	Insert char or enter insert mode
KEY_EIC	0514	Exit insert char mode
KEY_CLEAR	0515	Clear screen
KEY_EOS	0516	Clear to end of screen
KEY_EOL	0517	Clear to end of line
KEY_SF	0520	Scroll 1 line forward
KEY_SR	0521	Scroll 1 line backwards (reverse)
KEY_NPAGE	0522	Next page
KEY_PPAGE	0523	Previous page
KEY_STAB	0524	Set tab
KEY_CTAB	0525	Clear tab
KEY_CATAB	0526	Clear all tabs
KEY_ENTER	0527	Enter or send
KEY_SRESET	0530	soft (partial) reset
KEY_RESET	0531	reset or hard reset
KEY_PRINT	0532	print or copy
KEY_LL	0533	home down or bottom (lower left) keypad is arranged like this: A1 up A3 left B2 right C1 down C3
KEY_A1	0534	Upper left of keypad
KEY_A3	0535	Upper right of keypad
KEY_B2	0536	Center of keypad
KEY_C1	0537	Lower left of keypad
KEY_C3	0540	Lower right of keypad
KEY_BTAB	0541	Back tab key
KEY_BEG	0542	beg(inning) key
KEY_CANCEL	0543	cancel key
KEY_CLOSE	0544	close key
KEY_COMMAND	0545	cmd (command) key
KEY_COPY	0546	copy key
KEY_CREATE	0547	create key
KEY_END	0550	end key
KEY_EXIT	0551	exit key
KEY_FIND	0552	find key
KEY_HELP	0553	help key
KEY_MARK	0554	mark key
KEY_MESSAGE	0555	message key
KEY_MOVE	0556	move key
KEY_NEXT	0557	next object key
KEY_OPEN	0560	open key
KEY_OPTIONS	0561	options key
KEY_PREVIOUS	0562	previous object key
KEY_REDO	0563	redo key
KEY_REFERENCE	0564	ref(erence) key
KEY_REFRESH	0565	refresh key
KEY_REPLACE	0566	replace key

KEY_RESTART	0567	restart key
KEY_RESUME	0570	resume key
KEY_SAVE	0571	save key
KEY_SBEG	0572	shifted beginning key
KEY_SCANCEL	0573	shifted cancel key
KEY_SCOMMAND	0574	shifted command key
KEY_SCOPY	0575	shifted copy key
KEY_SCREATE	0576	shifted create key
KEY_SDC	0577	shifted delete char key
KEY_SDL	0600	shifted delete line key
KEY_SELECT	0601	select key
KEY_SEND	0602	shifted end key
KEY_SEOL	0603	shifted clear line key
KEY_SEXIT	0604	shifted exit key
KEY_SFIND	0605	shifted find key
KEY_SHELP	0606	shifted help key
KEY_SHOME	0607	shifted home key
KEY_SIC	0610	shifted input key
KEY_SLEFT	0611	shifted left arrow key
KEY_SMESSAGE	0612	shifted message key
KEY_SMOVE	0613	shifted move key
KEY_SNEXT	0614	shifted next key
KEY_SOPTIONS	0615	shifted options key
KEY_SPREVIOUS	0616	shifted prev key
KEY_SPRINT	0617	shifted print key
KEY_SREDO	0620	shifted redo key
KEY_SREPLACE	0621	shifted replace key
KEY_SRIGHT	0622	shifted right arrow
KEY_SRSUME	0623	shifted resume key
KEY_SSAVE	0624	shifted save key
KEY_SSUSPEND	0625	shifted suspend key
KEY_SUNDO	0626	shifted undo key
KEY_SUSPEND	0627	suspend key
KEY_UNDO	0630	undo key

LINE GRAPHICS

The following variables may be used to add line-drawing characters to the screen with *waddch*( ). When defined for the terminal, the variable turns on **A_ALTCHARSET** bit. 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 *setscrreg*( ), *wsetscrreg*( ), *getsyx*( ), *getyx*( ), *getbegy*( ), *getmaxyx*( ). For these macros, no useful value is returned.

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

BUGS

Currently typeahead checking is done using a nodelay read followed by an *ungetch*( ) of any character that may have been read. Typeahead checking is done only if *wgetch*( ) has been called at least once. This will be changed in future releases. Programs which use a mixture of their own input routines with *curses* input routines may wish to call *typeahead*(*–1*) to turn off typeahead checking.

The argument to *napms*( ) is currently rounded up to the nearest second.

draino (ms) only works for *ms* equal to **0**.

WARNINGS

To use the new *curses* features, use the Release 3.0 version of *curses* on UNIX System V Release 3.0. All programs that ran with System V Release 2 *curses* will run with System V Release 3.0. You may link applications with object files based on the Release 2 *curses/terminfo* with the Release 3.0 *libcurses.a* library. You may link applications with object files based on the Release 3.0 *curses/terminfo* with the Release 2 *libcurses.a* library, so long as the application does not use the new features in the Release 3.0 *curses/terminfo*.

The plotting library *plot*(3X) and the curses library *curses*(3X) both use the names *erase*( ) and *move*( ). The *curses* versions are macros. If you need both libraries, put the *plot*(3X) code in a source file different from the *curses*(3X) code, and/or **#undef move**( ) and **erase**( ) in the *plot*(3X) code.

Between the time a call to *initscr*( ) and *endwin*( ) has been issued, use only the routines in the *curses* library to generate output. Using system calls or the "standard I/O package" (see *stdio*(3S)) for output during that time can cause unpredictable results.

SEE ALSO

cc(1), ld(1), ioctl(2), plot(3X), putc(3S), scanf(3S), stdio(3S), system(3S), vprintf(3S), profile(4), term(4), terminfo(4), termio(7), tty(7) varargs(5).

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

opendir opens the directory named by *filename* and associates a *directory stream* with it. *opendir* returns a pointer to be used to identify the *directory stream* in subsequent operations. The pointer NULL is returned if *filename* cannot be accessed or is not a directory, or if it cannot *malloc*(3X) enough memory to hold a DIR structure or a buffer for the directory entries.

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

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

seekdir sets the position of the next *readdir* operation on the *directory stream.* The new position reverts to the one associated with the *directory stream* when the *telldir* operation was performed from which *loc* was obtained. Values returned by *telldir* are good only if the directory has not changed due to compaction or expansion. This is not a problem with System V, but it may be with some file system types.

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

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

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

opendir:

[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

rewinddir is implemented as a macro, so its function address cannot be taken.

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

SEE ALSO

env(1), getenv(3C), login(1), 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

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 occur if the space assigned by *malloc* is overrun or if some random number is handed to *free*.

realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents are not changed 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 still allow alignment of any data type. *value* is rounded up to a multiple of the default when *grain* is set.
M_KEEP	Preserve data in a freed block until the next *malloc, realloc,* or *calloc*. This option is provided only for compatibility with the old version of *malloc* and is not recommended.

These values are defined in the <*malloc.h*> header file.

mallopt may be called repeatedly, but may not be called after the first small block is allocated.

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

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 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 list shows the valid symbols and their associated meanings.

[ ] * . ^ These symbols retain their meaning from *ed*(1).

$ Matches the end of the string; **\n** matches a new-line.

– Within brackets the minus means *through*. For example, **[a–z]** is equivalent to **[abcd...xyz]**. The – can appear as itself only if used as the first or last character. For example, the character class expression **[ ]–]** matches the characters **]** and –.

+ A regular expression followed by + means *one or more times*. For example, **[0–9]+** is equivalent to **[0–9] [0–9]***.

{m} {m,} {m,u}
Integer values enclosed in { } indicate the number of times the preceding regular expression is to be applied. The value *m* is the minimum number and *u* is a number, less than 256, which is the maximum. If only *m* is present (e.g., {m}), it indicates the exact number of times the regular expression is to be applied. The value **{m,}** is analogous to {m,infinity}. The plus (+) and star (*) operations are equivalent to {1,} and {0,} respectively.

(. . .)$*n*
The value of the enclosed regular expression is to be returned. The value is stored in the *(n+1)*th argument following the subject argument. At most ten enclosed regular expressions are allowed. *regex* makes its assignments unconditionally.

(. . .) Parentheses are used for grouping. An operator, e.g., *, +, { }, 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

Example 1:
```
	char *cursor, *newcursor, *ptr;
		...
	newcursor = regex((ptr = regcmp("^\n", (char *)0)), cursor);
	free(ptr);
```
This example matches a leading new-line in the subject string pointed at by cursor.

Example 2:
```
	char ret0[9];
	char *newcursor, *name;
		...
	name = regcmp("([A–Za–z][A–za–z0–9]{0,7})$0", (char *)0);
	newcursor = regex(name, "012Testing345", ret0);
```
This example matches through the string "Testing3" and returns the address of the character after the last matched character (the "4"). The string "Testing3" is copied to the character array *ret0*.

Example 3:
```
	#include "file.i"
	char *string, *newcursor;
		...
	newcursor = regex(name, string);
```
This example applies a precompiled regular expression in **file.i** [see *regcmp*(1)] against *string*.

These routines are kept in **/lib/libPW.a**.

SEE ALSO

ed(1), malloc(3C), regcmp(1)

BUGS

The user program may run out of memory if *regcmp* is called iteratively without freeing the vectors no longer required.

NAME

intro – introduction to file formats

DESCRIPTION

This section outlines the formats of various files. The C structure declarations for the file formats are given where applicable. Usually, the header files containing these structure declarations can be found in the directories */usr/include* or */usr/include/sys*. For inclusion in C language programs, however, the syntax **#include <filename.h>** or **#include <sys/filename.h>** should be used.

NAME

a.out – common assembler and link editor output

SYNOPSIS

#include <a.out.h>

DESCRIPTION

The file name *a.out* is the default output file name from the link editor *ld*(1). The link editor makes *a.out* executable if there were no errors in linking. The output file of the assembler *as*(1), also follows the common object file format of the *a.out* file although the default file name is different.

A common object file consists of a file header, a UNIX system header, a table of section headers, section data, relocation information, a symbol table header, a string table, symbol table entries, and postload information. The order is given below.

File header.
UNIX system header.
Section 1 header.
...
Section n header.
Section 1 data.
...
Section n data.
Section 1 relocation.
...
Section n relocation.
Symbol table header.
String table.
Symbol table entries.
Postload information.

The file header and UNIX system header are always present. Other sections may be absent under different circumstances. Also note that relocation information is absent after linking unless the **–r** option of *ld*(1) was used.

When an **a.out** file is loaded into memory for execution, four logical segments are set up: the text segment, the data segment (initialized data followed by uninitialized, the latter actually being initialized to all 0's), thread local data (in the case that the process will use more than one processor), and a stack.

The *a.out* file produced by *ld*(1) is identified by the magic number 0413 in the first field of the UNIX system header. The headers (file header, UNIX system header, and section headers) are loaded at the beginning of the text segment and the text immediately follows the headers in the user address space. The first text address equals 0x400000 plus the size of the headers, and varies depending upon the number of section headers in the *a.out* file. The text segment is not writable by the program, and, to preserve the ability to postload, should contain only instructions. If other processes are executing the same *a.out* file, the processes share a single text segment.

The data segment starts by default at virtual address 0x10000000. The first three pages of this data region are mapped to contain registers for controlling the floating point unit, so the actual data begins in 0x10003000. By convention, the page at this address is dedicated to holding common floating-point constants, so the page at 0x10004000 begins the true data region. The data region can be grown by the *brk*(2) system call.

Threadlocal storage begins at virtual address 0x7fe00000. The threadlocal region cannot be initialized or grown.

The stack starts at 0x7fe00000 and grows toward smaller addresses. It can be grown simply by accessing off of the low end of it.

For relocatable files, the value of a word that is not relocated is exactly its value in the file. If a word is relocated by the base of a region (such as the text or data region), the base of the region is added to the word in the file. If a word is relocated by the address of a variable, the address of the variable is again added to the data in the file; in the case of an instruction, the target address is reflected in the instruction depending on the type of relocation and the kind of instruction being relocated.

File Header

The format of the **filehdr** header is

```
struct filehdr
{
        unsigned short  f_magic;   /* magic number */
        unsigned short  f_nscns;   /* number of sections */
        long            f_timdat;  /* time & date stamp */
        long            f_symptr;  /* file pointer to symbolic header */
        long            f_nsyms;   /* sizeof(symbolic hdr) */
        unsigned short  f_opthdr;  /* sizeof(optional hdr) */
        unsigned short  f_flags;   /* flags */
};
```

UNIX System Header

The format of the UNIX system header is

```
typedef         struct aouthdr
{
        short   magic;          /* magic number */
        short   vstamp;         /* version stamp */
        long    tsize;          /* text size in bytes, padded to FW bdry */
        long    dsize;          /* initialized data */
        long    bsize;          /* uninitialized data */
        long    entry;          /* entry pt. */
        long    text_start;     /* starting address of text */
        long    data_start;     /* starting address of data */
        long    bss_start;      /* starting address of bss */
        long    FIX_flags;      /* bits describing h'ware fixes applied */
        long    tlsize;         /* thread-local data area size */
        long    tl_start;       /* starting address of thread-local */
        long    stack_start;    /* starting address of stack */
        long    stamp;          /* 2 words of version/testing stamp */
        long    gp_value;       /* the gp value used for this object */
} AOUTHDR;
```

Section Header

The format of the section header is

```
struct scnhdr
{
        char            s_name[8]; /* section name */
        long            s_paddr;   /* physical address */
        long            s_vaddr;   /* virtual address */
        long            s_size;    /* section size */
        long            s_scnptr;  /* file ptr to raw data for section */
        long            s_relptr;  /* file ptr to relocation */
        unsigned        s_nreloc;  /* the number of relocation entries */
        unsigned short  o_nreloc;  /* field for backwards compatibility */
        unsigned short  filler2;   /* number of line numbers (not used) */
        long            s_flags;   /* flags */
};
```

Relocation

Object files contain one relocation entry for each relocatable reference in the text or data. If relocation information is present, it appears in the following format:

```
typedef struct
{
  longr_vaddr;                  /* (virtual) address of reference */
  unsigned  r_symndx:24;        /* index into symbol table */
            r_reserved:3;
            r_type:4;           /* relocation type */
            r_extern:1;         /* extern flag */
} RELOC;
```

The field *r_extern* distinguishes between relocation by the base of a section (*r_extern* is zero, *r_symndx* has the section number) and relocation by a symbol (*r_extern* is one, and *r_symndx* has the symbol table index for the symbol). If there is no relocation information, *s_relptr* is 0.

Symbol Table Header

The format of the symbol table header is

```
typedef struct
{
        short   magic;          /* to verify validity of the table */
        short   vstamp;         /* version stamp */
        long    filler1[15];    /* for backwards compatibility */
        long    strings_ct;     /* number of bytes of strings */
        long    strings_off;    /* offset of strings */
        long    comini_ct;      /* initialized common data */
        long    comini_off;     /* offset of initialized common data */
        long    address_ct;     /* number of text-relocated addresses */
        long    address_off;    /* offset of text-relocated addresses */
        long    symbols_ct;     /* number of external symbols */
        long    symbols_off;    /* offset of external symbols */
} SYMBOLS;
```

The symbol table header is followed by three additional blocks of data; the symbol table entries, the string table, and the list of text-relocated addresses that is used to support postloading.

DISCUSSION

Note that, in constrast to other UNIX implementations, there is no explicit *bss* section; all the information is taken from the UNIX system header. The threadlocal information is treated similarly.

In addition to the text and data sections, other sections may be present that do not directly affect the execution of the program. A section of #ident information may be present, giving information on how to rebuild the executable progra. A section may be present that supports FORTRAN common initialization (in object files only). Two different forms of debugging sections may be present with auxiliary information that may be used to improve the debugging.

Details about the values of specific fields can be found in the full *a.out.h* file.

SEE ALSO

as(1), brk(2), cc(1), ld(1)

NAME

acct – per-process accounting file format

SYNOPSIS

#include <sys/acct.h>

DESCRIPTION

Files produced as a result of calling *acct*(2) contain records in the form defined by <*sys/acct.h*>:

```
typedef  ushort comp_t; /* "floating point" */
                /* 13-bit fraction, 3-bit exponent  */

struct   acct
{
         char    ac_flag;       /* Accounting flag */
         char    ac_stat;       /* Exit status */
         ushort  ac_uid;        /* Accounting user ID */
         ushort  ac_gid;        /* Accounting group ID */
         dev_t   ac_tty;        /* control typewriter */
         time_t  ac_btime;      /* Beginning time */
         comp_t  ac_utime;      /* acctng user time in clock ticks */
         comp_t  ac_stime;      /* acctng system time in clock ticks */
         comp_t  ac_etime;      /* acctng elapsed time in clock ticks */
         comp_t  ac_mem;        /* memory usage in clicks */
         comp_t  ac_io;         /* chars trnsfrd by read/write */
         comp_t  ac_rw;         /* number of block reads/writes */
         char    ac_comm[8];    /* command name */
};

extern   struct   acct          acctbuf;
extern   struct   inode         *acctp;  /* inode of accounting file */

#define AFORK  01               /* has executed fork, but no exec */
#define ASU    02               /* used super-user privileges */
#define ACCTF  0300             /* record type: 00 = acct */
```

In *ac_flag*, the AFORK flag is turned on by each *fork*(2) and turned off by an *exec*(2). The *ac_comm* field is inherited from the parent process and is reset by any *exec*. Each time the system charges the process with a clock tick, it also adds to *ac_mem* the current process size, computed as follows:

(data size) + (text size) / (number of in-core processes using text)

The value of *ac_mem / (ac_stime + ac_utime)* can be viewed as an approximation to the mean process size, as modified by text-sharing.

The structure **tacct.h**, which resides with the source files of the accounting commands, represents the total accounting format used by the various accounting commands:

```
/*
 * total accounting (for acct period), also for day
 */

struct  tacct {
        uid_t          ta_uid;       /* userid */
        char           ta_name[8];   /* login name */
        float          ta_cpu[2];    /* cum. cpu time, p/np (mins) */
        float           ta_kcore[2]; /* cum kcore-minutes, p/np */
        float           ta_con[2];   /* cum. connect time, p/np, mins */
        float           ta_du;       /* cum. disk usage */
        long            ta_pc;       /* count of processes */
        unsigned short  ta_sc;       /* count of login sessions */
        unsigned short  ta_dc;       /* count of disk samples */
        unsigned short  ta_fee;      /* fee for special services */
};
```

SEE ALSO

acct(1M), acct(2), acctcom(1), exec(2), fork(2)

BUGS

The *ac_mem* value for a short-lived command gives little information about the actual size of the command, because *ac_mem* may be incremented while a different command (e.g., the shell) is being executed by the process.

NAME

ar – common archive file format

SYNOPSIS

#include <ar.h>

DESCRIPTION

The archive command *ar*(1) is used to combine several files into one. Archives are used mainly as libraries to be searched by the link editor *ld*(1).

Each archive begins with the archive magic string.

```
#define  ARMAG  "!<arch>\n"          /* magic string */
#define  SARMAG  8                   /* length of magic string */
```

Each archive which contains common object files [see *a.out*(4)] includes an archive symbol table. This symbol table is used by the link editor *ld*(1) to determine which archive members must be loaded during the link edit process. The archive symbol table (if it exists) is always the first file in the archive (but is never listed) and is automatically created or updated by *ar*.

Following the archive magic string are the archive file members. Each file member is preceded by a file member header which is of the following format:

```
#define  ARFMAG  "`\n"          /* header trailer string */

struct  ar_hdr                  /* file member header */
{
   char  ar_name[16];           /* '/' terminated file member name */
   char  ar_date[12];           /* file member date */
   char  ar_uid[6];             /* file member user identification */
   char  ar_gid[6];             /* file member group identification */
   char  ar_mode[8];            /* file member mode (octal) */
   char  ar_size[10];           /* file member size */
   char  ar_fmag[2];            /* header trailer string */
};
```

All information in the file member headers is stored in printable ASCII. The numeric information contained in the headers is stored as decimal numbers (except for *ar_mode* which is in octal). Thus, if the archive contains printable files, the archive itself is printable.

The *ar_name* field is blank-padded and slash (/) terminated. The *ar_date* field is the modification date of the file at the time of its insertion into the archive. Common format archives can be moved from system to system as long as the portable archive command *ar*(1) is used. Conversion tools such as *convert*(1) exist to aid in the transportation of non-common format archives to this format.

Each archive file member begins on an even byte boundary; a newline is inserted between files if necessary. Nevertheless the size given reflects the actual size of the file exclusive of padding.

Notice there is no provision for empty areas in an archive file.

If the archive symbol table exists, the first file in the archive has a zero length name (i.e., **ar_name[0] ==** '/'). The contents of this file are:

- The number of symbols. Length: 4 bytes.
- The array of offsets into the archive file. Length: 4 bytes * "the number of symbols".

- The name string table. Length: *ar_size* – (4 bytes * ("the number of symbols" + 1)).

The number of symbols and the array of offsets are managed with *sgetl* and *sputl*. The string table contains exactly as many null terminated strings as there are elements in the offsets array. Each offset from the array is associated with the corresponding name from the string table (in order). The names in the string table are all the defined global symbols found in the common object files in the archive. Each offset is the location of the archive header for the associated symbol.

SEE ALSO

a.out(4), ar(1), ld(1), sputl(3X), strip(1)

WARNINGS

strip(1) removes all archive symbol entries from the header. The archive symbol entries must be restored via the **ts** option of the *ar*(1) command before the archive can be used with the link editor *ld*(1).

NAME

checklist – list of file systems processed by fsck

DESCRIPTION

checklist contains a list of filesystems checked by **fsck**(1) during system bootup.

SEE ALSO

fsck(1)

NAME

core – format of core image file

DESCRIPTION

The UNIX system writes out a core image of a terminated process when any of various errors occur. See *signal*(2) for the list of reasons; the most common are memory violations, illegal instructions, bus errors, and user-generated quit signals. The core image is called *core* and is written in the process's working directory (provided it can be; normal access controls apply). A process with an effective user ID different from the real user ID does not produce a core image.

The core image contains one or more header sections, followed by the data and stack spaces of the user process. There is one header section for each active thread in the user process and only one header section if there are no threads.

Each header section consists of a copy of the system's user data for the process, including the registers as they existed at the time of the fault. The size of this section is USIZE pages; USIZE is defined in *<sys/param.h>*.

Following the user data in the header is the contents of the vector register file for the process. The VRF is only included if the thread/process was actually using the floating point unit. Following the VRF is any thread local storage. If there is no thread local storage in the process, this section is empty.

The format of the information in the first section is described by the *user* structure of the system, defined in *<sys/user.h>*. Not included in this file are the locations of the registers. These are outlined in *<machine/reg.h>*. The registers occupy the last EF_SIZE words of the user area.

Each section of the core file, user area, VRF, thread local, data space, and stack space is page-aligned.

SEE ALSO

a.out(4), crash(1M), dbg(1), setuid(2), signal(2), thread(2)

NAME

cpio – format of cpio archive

DESCRIPTION

The *header* structure, when the **–c** option of *cpio*(1) is not used, is:

```
struct {
      short h_magic,
            h_dev;
      ushort      h_ino,
            h_mode,
            h_uid,
            h_gid;
      short h_nlink,
            h_rdev,
            h_mtime[2],
            h_namesize,
            h_filesize[2];
      char  h_name[h_namesize rounded to word];
} Hdr;
```

When the **–c** option is used, the *header* information is described by:

```
sscanf(Chdr,"%6o%6o%6o%6o%6o%6o%6o%6o%11lo%6o%11lo%s",
      &Hdr.h_magic, &Hdr.h_dev, &Hdr.h_ino, &Hdr.h_mode,
      &Hdr.h_uid, &Hdr.h_gid, &Hdr.h_nlink, &Hdr.h_rdev,
      &Longtime, &Hdr.h_namesize,&Longfile,Hdr.h_name);
```

Longtime and *Longfile* are equivalent to *Hdr.h_mtime* and *Hdr.h_filesize*, respectively. The contents of each file are recorded in an element of the array of varying length structures, *archive*, together with other items describing the file. Every instance of *h_magic* contains the constant 070707 (octal). The items *h_dev* through *h_mtime* have meanings explained in *stat*(2). The length of the null-terminated path name *h_name*, including the null byte, is given by *h_namesize*.

The last record of the *archive* always contains the name TRAILER!!!. Special files, directories, and the trailer are recorded with *h_filesize* equal to zero.

SEE ALSO

cpio(1), find(1), stat(2)

NAME

dir – format of directories

SYNOPSIS

#include <sys/dir.h>

DESCRIPTION

A directory behaves exactly like an ordinary file, save that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry [see *fs*(4)]. The structure of a directory entry as given in the include file is:

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

By convention, the first two entries in each directory are for . and ... The first is an entry for the directory itself. The second is for the parent directory. The meaning of .. is modified for the root directory of the master file system; there is no parent, so .. has the same meaning as ..

WARNING

This structure reflects only one of several possible directory formats. A file system independent structure for directory entries is given in *dirent(4).*

SEE ALSO

dirent(4), fs(4)

NAME

dirent – file system independent directory entry

SYNOPSIS

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

DESCRIPTION

Different file system types may have different directory entries. The *dirent* structure defines a file system independent directory entry, which contains information common to directory entries in different file system types. A set of these structures is returned by the *getdents*(2) system call.

The *dirent* structure is defined below.

```
struct    dirent {
                    long                d_ino;
                    off_t               d_off;
                    unsigned short      d_reclen;
                    char                d_name[1];
          };
```

The *d_ino* is a number which is unique for each file in the file system. The field *d_off* is the offset of that directory entry in the actual file system directory. The field *d_name* is the beginning of the character array giving the name of the directory entry. This name is null terminated and may have at most MAXNAMLEN characters. This results in file system independent directory entries being variable length entities. The value of *d_reclen* is the record length of this entry. This length is defined to be the number of bytes between the current entry and the next one, so that it always results in the next entry being on a long boundary.

FILES

/usr/include/sys/dirent.h

SEE ALSO

getdents(2)

NAME

fs: file system – format of system volume

SYNOPSIS

#include <sys/types.h>
#include <sys/param.h>
#include <sys/sys5/filsys.h>

DESCRIPTION

This description covers System V file system formats. Formats for the Fast File System will be included in a later release of this document.

Every System V file system volume uses a common format for certain vital information. Every such volume is divided into 512-byte long sectors; sector 0 is unused and is available to contain a bootstrap program or other information. Sector 1 is the *super-block*. The format of a super-block is:

```
struct  filsys
{
ushort s_isize;        /* size in blocks of i-list */
daddr_t                s_fsize;            /* size in blocks of entire volume */
short  s_nfree;        /* number of addresses in s_free */
daddr_t                s_free[NICFREE];    /* free block list */
short  s_ninode;       /* number of i-nodes in s_inode */
ushort s_inode[NICINOD];                   /* free i-node list */
char   s_flock;        /* lock during free list manipulation */
char   s_ilock;        /* lock during i-list manipulation */
char   s_fmod;         /* super block modified flag */
char   s_ronly;        /* mounted read-only flag */
time_t s_time;         /* last super block update */
short  s_dinfo[4];     /* device information */
daddr_t                s_tfree;            /* total free blocks*/
ushort s_tinode;       /* total free i-nodes */
char   s_fname[6];     /* file system name */
char   s_fpack[6];     /* file system pack name */
long   s_fill[12];     /* ADJUST to make sizeof filsys
                                           be 512 bytes */
long   s_state;        /* file system state */
long   s_magic;        /* magic number to denote new
                                           file system */
long   s_type;         /* type of new file system */
};

#define         FsMAGIC             0xfd187e20/* s_magic number */

#define         Fs1b                1/* 512-byte block */
#define         Fs2b                2/* 1024-byte block */
#define         Fs8b                8/* 4096-byte block */

#define         FsOKAY              0x7c269d38/* s_state: clean */
#define         FsACTIVE            0x5e72d81a/* s_state: active */
#define         FsBAD               0xcb096f43/* s_state: bad root */
#define         FsBADBLK            0xbadbc14b/* s_state: bad block corrupted it */
```

s_type indicates the file system type. Currently, only the 4096-byte file system is supported. *s_magic* distinguishes the original 512-byte and 1024-byte oriented file systems from the newer file systems. If this field is not equal to the magic number, *FsMAGIC*, the volume is not considered a System V file system; otherwise the *s_type* field is used. In the following description, a block is then determined by the type. Stardent 1500/3000 disk drives are formatted with 1024-byte sectors. For the 4096-byte system, a block is 4096 bytes or 4 sectors. The operating system takes care of all conversions from logical block numbers to physical sector numbers.

s_state indicates the state of the file system. A cleanly unmounted, not damaged file system is indicated by the FsOKAY state. After a file system has been mounted for update, the state changes to FsACTIVE. A special case is used for the root file system. If the root file system appears damaged at boot time, it is mounted but marked FsBAD. After a file system has been unmounted, the state reverts to FsOKAY.

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

The free list for each volume is maintained as follows. The *s_free* array contains, in *s_free*[1], ..., *s_free*[*s_nfree*–1], up to 49 numbers of free blocks. *s_free*[0] is the block number of the head of a chain of blocks constituting the free list. The first long in each free-chain block is the number (up to 50) of free-block numbers listed in the next 50 longs of this chain member. The first of these 50 blocks is the link to the next member of the chain. To allocate a block: decrement *s_nfree*, and the new block is *s_free*[*s_nfree*]. If the new block number is 0, there are no blocks left, so give an error. If *s_nfree* became 0, read in the block named by the new block number, replace *s_nfree* by its first word, and copy the block numbers in the next 50 longs into the *s_free* array. To free a block, check if *s_nfree* is 50; if so, copy *s_nfree* and the *s_free* array into it, write it out, and set *s_nfree* to 0. In any event set *s_free*[*s_nfree*] to the freed block's number and increment *s_nfree*.

s_tfree is the total free blocks available in the file system.

s_ninode is the number of free i-numbers in the *s_inode* array. To allocate an i-node: if *s_ninode* is greater than 0, decrement it and return *s_inode*[*s_ninode*]. If it was 0, read the i-list and place the numbers of all free i-nodes (up to 100) into the *s_inode* array, then try again. To free an i-node, provided *s_ninode* is less than 100, place its number into *s_inode*[*s_ninode*] and increment *s_ninode*. If *s_ninode* is already 100, do not bother to enter the freed i-node into any table. This list of i-nodes is only to speed up the allocation process; the information as to whether the i-node is really free or not is maintained in the i-node itself.

s_tinode is the total free i-nodes available in the file system.

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

s_ronly is a read-only flag to indicate write-protection.

s_time is the last time the super-block of the file system was changed, and is the number of seconds that have elapsed since 00:00 Jan. 1, 1970 (GMT). During a reboot, the *s_time* of the super-block for the root file system is used to set the system's idea of the time.

s_fname is the name of the file system and *s_fpack* is the name of the pack.

I-numbers begin at 1, and the storage for i-nodes begins in block 2. Also, i-nodes are 64 bytes long. I-node 1 is reserved for future use. I-node 2 is reserved for the root directory of the file system, but no other i-number has a built-in meaning. Each i-node represents one file. For the format of an i-node and its flags, see *inode*(4).

SEE ALSO

fsck(1M), fstatf(2), fsdb(1M), inode(4), mkfs(1M), mount(2), statf(2)

NAME

fspec – format specification in text files

DESCRIPTION

It is sometimes convenient to maintain text files on the UNIX system with non-standard tabs, (i.e., tabs which are not set at every eighth column). Such files must generally be converted to a standard format, frequently by replacing all tabs with the appropriate number of spaces, before they can be processed by UNIX system commands. A format specification occurring in the first line of a text file specifies how tabs are to be expanded in the remainder of the file.

A format specification consists of a sequence of parameters separated by blanks and surrounded by the brackets **<:** and **:>**. Each parameter consists of a keyletter, possibly followed immediately by a value. The following parameters are recognized:

t*tabs* The **t** parameter specifies the tab settings for the file. The value of *tabs* must be one of the following:

1. a list of column numbers separated by commas, indicating tabs set at the specified columns;
2. a – followed immediately by an integer *n*, indicating tabs at intervals of *n* columns;
3. a – followed by the name of a "canned" tab specification.

Standard tabs are specified by **t–8**, or equivalently, **t1,9,17,25,**etc. The canned tabs which are recognized are defined by the *tabs*(1) command.

s*size* The **s** parameter specifies a maximum line size. The value of *size* must be an integer. Size checking is performed after tabs have been expanded, but before the margin is prepended.

m*margin* The **m** parameter specifies a number of spaces to be prepended to each line. The value of *margin* must be an integer.

d The **d** parameter takes no value. Its presence indicates that the line containing the format specification is to be deleted from the converted file.

e The **e** parameter takes no value. Its presence indicates that the current format is to prevail only until another format specification is encountered in the file.

Default values, which are assumed for parameters not supplied, are **t–8** and **m0**. If the **s** parameter is not specified, no size checking is performed. If the first line of a file does not contain a format specification, the above defaults are assumed for the entire file. The following is an example of a line containing a format specification:

```
* <:t5,10,15 s72:> *
```

If a format specification can be disguised as a comment, it is not necessary to code the **d** parameter.

SEE ALSO

ed(1), newform(1), tabs(1)

NAME

fstab – file-system-table

DESCRIPTION

The */etc/fstab* file contains information about file systems for use by *mount*(1M) and *mountall*(1M). Each entry in */etc/fstab* has the following format:

column 1	block special file name of file system or advertised remote resource
column 2	mount-point directory
column 3	(optional) "–r" if to be mounted read-only; "–d[r]" if remote
column 4	(optional) file system type string
column 5+	ignored

White-space separates columns. Lines beginning with "# " are comments. Empty lines are ignored.

A file-system-table might read:

```
/dev/dsk/c1d4s2 /usr S54K
/dev/dsk/c1d1s2 /usr/src -r   AFFS
adv_resource /mnt -d
server:/usr/src /usr/src      NFS, soft
```

FILES

/etc/fstab

SEE ALSO

mount(1M), mountall(1M)

NAME

gettydefs – speed and terminal settings used by getty

DESCRIPTION

The */etc/gettydefs* file contains information used by *getty*(1M) to set up the speed and terminal settings for a line. It supplies information on what the *login* prompt should look like. It also supplies the speed to try next if the user indicates the current speed is not correct by typing a <*break*> character.

Each entry in */etc/gettydefs* follows this format:

label# initial-flags # final-flags # login-prompt #next-label

Each entry is followed by a blank line. The various fields can contain quoted characters of the form **\b**, **\n**, **\c**, etc., as well as \nnn, where *nnn* is the octal value of the desired character. The various fields are:

label	This is the string against which *getty* tries to match its second argument. It is often the speed, such as **1200**, at which the terminal is supposed to run, but it need not be (see below).
initial-flags	These flags are the initial *ioctl*(2) settings to which the terminal is to be set if a terminal type is not specified to *getty*. The flags that *getty* understands are the same as the ones listed in */usr/include/sys/termio.h* [see *termio*(7)]. Normally only the speed flag is required in the *initial-flags*. *getty* automatically sets the terminal to raw input mode and takes care of most of the other flags. The *initial-flag* settings remain in effect until *getty* executes *login*(1).
final-flags	These flags take the same values as the *initial-flags* and are set just prior to *getty* executes *login*. The speed flag is again required. The composite flag **SANE** takes care of most of the other flags that need to be set so that the processor and terminal are communicating in a rational fashion. The other two commonly specified *final-flags* are **TAB3**, so that tabs are sent to the terminal as spaces, and **HUPCL**, so that the line is hung up on the final close.
login-prompt	This entire field is printed as the *login-prompt*. Unlike the above fields where white space is ignored (a space, tab or new-line), they are included in the *login-prompt* field.
next-label	If this entry does not specify the desired speed, indicated by the user typing a <*break*> character, then *getty* searches for the entry with *next-label* as its *label* field and set up the terminal for those settings. Usually, a series of speeds are linked together in this fashion, into a closed set; For instance, **2400** linked to **1200**, which in turn is linked to **300**, which finally is linked to **2400**.

If *getty* is called without a second argument, then the first entry of **/etc/gettydefs** is used, thus making the first entry of */etc/gettydefs* the default entry. It is also used if *getty* can not find the specified *label*. If */etc/gettydefs* itself is missing, there is one entry built into the command which will bring up a terminal at **300** baud.

It is strongly recommended that after making or modifying */etc/gettydefs*, it be run through *getty* with the check option to be sure there are no errors.

FILES

/etc/gettydefs

SEE ALSO

getty(1M), ioctl(2), login(1), termio(7)

NAME

group – group file

DESCRIPTION

group contains for each group the following information:

group name
encrypted password
numerical group ID
comma-separated list of all users allowed in the group

This is an ASCII file. The fields are separated by colons; each group is separated from the next by a new-line. If the password field is null, no password is demanded.

This file resides in directory */etc*. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical group ID's to names.

FILES

/etc/group

SEE ALSO

newgrp(1M), passwd(1), passwd(4)

NAME

hosts – host name data base

SYNOPSIS

/etc/hosts

DESCRIPTION

The *hosts* file contains information regarding the known hosts on the DARPA Internet. For each host a single line should be present with the following information:

Internet address
official host name
aliases

Items are separated by any number of blanks and/or tab characters. A '#' indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file. This file is normally created from the official host data base maintained at the Network Information Control Center (NIC), though local changes may be required to bring it up to date regarding unofficial aliases and/or unknown hosts.

Network addresses are specified in the conventional '.' notation using the *inet_addr*() routine from the Internet address manipulation library, *inet*(3N). Host names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/hosts

SEE ALSO

gethostent(3N), rhosts(4)

NAME

hosts.equiv – list of trusted hosts

DESCRIPTION

Hosts.equiv resides in directory */etc* and contains a list of trusted hosts. When an *rlogin*(1) or *rsh*(1) request from such a host is made, and the initiator of the request is in */etc/passwd*, then no further validity checking is done. That is, *rlogin* does not prompt for a password, and *rsh* completes successfully. So a remote user is "equivalenced" to a local user with the same user ID when the remote user is in *hosts.equiv*.

The format of *hosts.equiv* is a list of names, as in this example:

```
host1
host2
+@group1
-@group2
```

[The "+" and "-" features are only available with NFS systems.] A line consisting of a simple host name means that anyone logging in from that host is trusted. A line consisting of +@*group* means that all hosts in that network group are trusted. A line consisting of –@*group* means that hosts in that group are not trusted. Programs scan *hosts.equiv* linearly, and stop at the first hit (either positive for hostname and +@ entries, or negative for –@ entries). A line consisting of a single + means that everyone is trusted.

The *.rhosts* file has the same format as *hosts.equiv*. When user XXX executes *rlogin* or *rsh*, the *.rhosts* file from XXX's home directory is conceptually concatenated onto the end of *hosts.equiv* for permission checking. However, –@ entries are not sticky. If a user is excluded by a minus entry from *hosts.equiv* but included in *.rhosts*, then that user is considered trusted. In the special case when the user is root, then only the */.rhosts* file is checked.

It is also possible to have two entries (separated by a single space) on a line of these files. In this case, if the remote host is equivalenced by the first entry, then the user named by the second entry is allowed to log in as anyone, that is, specify any name to the –l flag (provided that name is in the */etc/passwd* file, of course). Thus

```
    sundown john
```

allows *john* to log in from sundown as anyone. The usual usage would be to put this entry in the *.rhosts* file in the home directory for *bill* . Then *john* may log in as *bill* when coming from sundown. The second entry may be a netgroup, thus

```
+@group1 +@group2
```

allows any user in *group2* coming from a host in *group1* to log in as anyone.

FILES

/etc/hosts.equiv	
/etc/yp/*domain*/netgroup	[NFS systems only]
/etc/yp/*domain*/netgroup.byuser	[NFS systems only]
/etc/yp/*domain*/netgroup.byhost	[NFS systems only]

SEE ALSO

rlogin(1), rsh(1), netgroup(5)

NAME

inittab – script for the init process

DESCRIPTION

The *inittab* file supplies the script to *init*'s role as a general process dispatcher. The process that constitutes the majority of *init*'s process dispatching activities is the line process */etc/getty* that initiates individual terminal lines. Other processes typically dispatched by *init* are daemons and the shell.

The *inittab* file is composed of entries that are position dependent and have the following format:

id:rstate:action:process

Each entry is delimited by a newline, however, a backslash (\) preceding a newline indicates a continuation of the entry. Up to 512 characters per entry are permitted. Comments may be inserted in the *process* field using the *sh*(1) convention for comments. Comments for lines that spawn *getty*s are displayed by the *who*(1) command. It is expected that they contain some information about the line such as the location. There are no limits (other than maximum entry size) imposed on the number of entries within the *inittab* file. The entry fields are:

id This is one or two characters used to uniquely identify an entry.

rstate This defines the *run-level* in which this entry is going to be processed. *Run-levels* effectively correspond to a configuration of processes in the system. That is, each process spawned by *init* is assigned a *run-level* or *run-levels* in which it is allowed to exist. The *run-levels* are represented by a number ranging from **0** through **6**. As an example, if the system is in *run-level* **1**, only those entries having a **1** in the *rstate* field are processed. When *init* is requested to change *run-levels*, all processes which do not have an entry in the *rstate* field for the target *run-level* are sent the warning signal (**SIGTERM**) and allowed a 20-second grace period before being forcibly terminated by a kill signal (**SIGKILL**). The *rstate* field can define multiple *run-levels* for a process by selecting more than one *run-level* in any combination from **0–6**. If no *run-level* is specified, then the process is assumed to be valid at all *run-levels* **0–6**. There are three other values, **a**, **b** and **c**, which can appear in the *rstate* field, even though they are not true *run-levels*. Entries which have these characters in the *rstate* field are processed only when the *telinit* [see *init*(1M)] process requests them to be run (regardless of the current *run-level* of the system). They differ from *run-levels* in that *init* can never enter *run-level* **a**, **b** or **c**. Also, a request for the execution of any of these processes does not change the current *run-level*. Furthermore, a process started by an **a**, **b** or **c** command is not killed when *init* changes levels. They are only killed if their line in */etc/inittab* is marked **off** in the *action* field, their line is deleted entirely from */etc/inittab*, or *init* goes into the *SINGLE USER* state.

action Key words in this field tell *init* how to treat the process specified in the *process* field. The actions recognized by *init* are:

respawn If the process does not exist then start the process, do not wait for its termination (continue scanning the *inittab* file), and when it dies restart the process. If the process currently exists then do nothing and continue scanning the *inittab* file.

wait Upon *init*'s entering the *run-level* that matches the entry's *rstate*, start the process and wait for its termination. All subsequent reads of the *inittab* file while *init* is in the same *run-level* cause *init* to ignore this entry.

once Upon *init*'s entering a *run-level* that matches the entry's *rstate*, start the process, do not wait for its termination. When it dies, do not restart the process. If upon entering a new *run-level*, where the process is still running from a previous *run-level* change, the program is not restarted.

boot The entry is processed only at *init*'s boot-time read of the *inittab* file. *init* is to start the process, not wait for its termination; and when it dies, not restart the process. In order for this instruction to be meaningful, the *rstate* should be the default or it must match *init*'s *run-level* at boot time. This action is useful for an initialization function following a hardware reboot of the system.

bootwait The entry is to be processed the first time *init* goes from single-user to multi-user state after the system is booted. (If **initdefault** is set to **2**, the process runs right after the boot.) *init* starts the process, waits for its termination and, when it dies, does not restart the process.

powerfail Execute the process associated with this entry only when *init* receives a power fail signal [SIGPWR see *signal*(2)].

powerwait Execute the process associated with this entry only when *init* receives a power fail signal (SIGPWR) and wait until it terminates before continuing any processing of *inittab*.

off If the process associated with this entry is currently running, send the warning signal (SIGTERM) and wait 20 seconds before forcibly terminating the process via the kill signal (SIGKILL). If the process is nonexistent, ignore the entry.

ondemand This instruction is really a synonym for the **respawn** action. It is functionally identical to **respawn** but is given a different keyword in order to divorce its association with *run-levels*. This is used only with the **a**, **b** or **c** values described in the *rstate* field.

initdefault An entry with this *action* is only scanned when *init* is initially invoked. *init* uses this entry, if it exists, to determine which *run-level* to enter initially. It does this by taking the highest *run-level* specified in the **rstate** field and using that as its initial state. If the *rstate* field is empty, this is interpreted as **0123456** and so *init* enters *run-level* **6**. Additionally, if *init* does not find an **initdefault** entry in */etc/inittab*, then it requests an initial *run-level* from the user at reboot time.

sysinit Entries of this type are executed before *init* tries to access the console (i.e., before the **Console Login:** prompt). It is expected that this entry is used only to initialize devices on which *init* might try to ask the *run-level* question. These entries are executed and waited for before continuing.

process This is a *sh* command to be executed. The entire **process** field is prefixed with *exec* and passed to a forked *sh* as **sh –c 'exec** *command*'. For this reason, any legal *sh* syntax can appear in the *process* field. Comments can be inserted with the *; #comment* syntax.

FILES

/etc/inittab

SEE ALSO

exec(2), getty(1M), init(1M), open(2), sh(1), signal(2), who(1)

NAME

inode – format of an i-node

SYNOPSIS

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

DESCRIPTION

An i-node for a plain file or directory in a file system has the following structure defined by *<sys/ino.h>*.

```
/*      Inode structure as it appears on a disk block.  */

struct  dinode
{
        ushort  di_mode;        /* mode and type of file */
        short   di_nlink;       /* number of links to file */
        ushort  di_uid;         /* owner's user id */
        ushort  di_gid;         /* owner's group id */
        off_t   di_size;        /* number of bytes in file */
        char    di_addr[39];    /* disk block addresses */
        char    di_gen;         /* file generation number */
        time_t  di_atime;       /* time last accessed */
        time_t  di_mtime;       /* time last modified */
        time_t  di_ctime;       /* time created */
};
/*
 * The 39 address bytes: 13 addresses of 3 bytes each.
 *
 * The 40'th byte is used as generation count to allow detection of
 * the disk inode being reused.
 */
```

For the meaning of the defined types *off_t* and *time_t* see *types*(5).

SEE ALSO

fs(4), stat(2), types(5)

NAME

issue – issue identification file

DESCRIPTION

The file */etc/issue* contains the *issue* or project identification to be printed as a login prompt. This is an ASCII file which is read by program *getty* and then written to any terminal spawned or respawned from the *lines* file.

FILES

/etc/issue

SEE ALSO

login(1)

NAME

limits – operating system magnitude limits

DESCRIPTION

The header file <limits.h> is a list of magnitude limitations imposed by the Stardent 1500/3000 operating system.

ARG_MAX	20480	/* max length of arguments to exec */
CHAR_BIT	8	/* # of bits in a "char" */
CHAR_MAX	127	/* max integer value of a "char" */
CHAR_MIN	-128	/* min integer value of a "char" */
CHILD_MAX	50	/* max # of processes per user id */
CLK_TCK	100	/* # of clock ticks per second */
DBL_DIG	16	/* digits of precision of a "double" */
DBL_MAX	0d7fefffffffffffff	/* Approx. 1.7976931348623157e+308 */ /* Maximum decimal value of a "double"*/
DBL_MIN	0d0010000000000000	/* Approx. 2.2250738585072015E-308 */ /* Minimum decimal value of a "double"*/
FCHR_MAX	2147483647	/* max size of a file in bytes */
FLT_DIG	7	/* digits of precision of a "float" */
FLT_MAX	0d47efffffe0000000	/* Approx. 3.40282346638528860e+38 */ /* Maximum decimal value of a "float"*/
FLT_MIN	0d3810000000000000	/* Approx. 1.1754943508222875e-38 */ /* Minimum decimal value of a "float"*/
HUGE_VAL	(DBL_MAX)	/*error value returned by Math lib */
INT_MAX	2147483647	/* max decimal value of an "int" */
INT_MIN	-2147483648	/* min decimal value of an "int" */
LINK_MAX	1000	/* max # of links to a single file */
LONG_MAX	2147483647	/* max decimal value of a "long" */
LONG_MIN	-2147483648	/* min decimal value of a "long" */
NAME_MAX	14	/* max # of characters in a file name */
OPEN_MAX	64	/* max # of files a process can have open */
PASS_MAX	8	/* max # of characters in a password */
PATH_MAX	1024	/* max # of characters in a path name */
PID_MAX	30000	/* max value for a process ID */
PIPE_BUF	4096	/* max # bytes atomic in write to a pipe */
PIPE_MAX	32768	/* max # bytes written to a pipe in a write */
SHRT_MAX	32767	/* max decimal value of a "short" */
SHRT_MIN	-32768	/* min decimal value of a "short" */
STD_BLK	1024	/* # bytes in a physical I/O block */
SYS_NMLN	9	/* # of chars in uname-returned strings */
UID_MAX	60000	/* max value for a user or group ID */
USI_MAX	4294967295	/* max decimal value of an "unsigned" */
WORD_BIT	32	/* # of bits in a "word" or "int" */

NAME

mnttab – mounted file system table

SYNOPSIS

#include <mnttab.h>

DESCRIPTION

mnttab resides in directory */etc* and contains a table of devices, mounted by the *mount*(1M) command, in the following structure as defined by <*mnttab.h*>:

```
struct  mnttab {
        char     mt_dev[32];
        char     mt_filsys[32];
        short    mt_ro_flg;
        time_t   mt_time;
        char mt_fstyp[16];
        char mt_mntopts[64];
};
```

Each entry is 150 bytes in length; the first 32 bytes are the null-padded name of the place where the *special file* is mounted; the next 32 bytes represent the null-padded root name of the mounted special file; the next 6 bytes contain the mounted *special file*'s read/write permissions and the date on which it was mounted the following bytes are the null-padded name of the file system type; and the remaining 64 bytes are the null-padded string of mount options.

The maximum number of entries in *mnttab* is based on the system parameter **NMOUNT** located in */usr/crs/uts/cf/master.c* which defines the number of allowable mounted special files.

SEE ALSO

mount(1M), setmnt(1M)

NAME

networks – network name data base

DESCRIPTION

The *networks* file contains information regarding the known networks which comprise the DARPA Internet. For each network a single line should be present with the following information:

official network name
network number
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file. This file is normally created from the official network data base maintained at the Network Information Control Center (NIC), though local changes may be required to bring it up to date regarding unofficial aliases and/or unknown networks.

Network number may be specified in the conventional "." notation using the *inet_network*() routine from the Internet address manipulation library, *inet*(3N). Network names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/networks

SEE ALSO

getnetent(3N)

BUGS

A name server should be used instead of a static file.

NAME

passwd – password file

DESCRIPTION

passwd contains for each user the following information:

login name
encrypted password
numerical user ID
numerical group ID
GCOS job number, box number, optional GCOS user ID
initial working directory
program to use as shell

This is an ASCII file. Each field within each user's entry is separated from the next by a colon. The GCOS field is used only when communicating with that system, and in other installations can contain any desired information. Each user is separated from the next by a new-line. If the password field is null, no password is demanded; if the shell field is null, the shell itself is used.

This file resides in directory */etc*. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical user IDs to names.

The encrypted password consists of 13 characters chosen from a 64-character alphabet (**.**, **/**, **0–9**, **A–Z**, **a–z**), except when the password is null, in which case the encrypted password is also null. Password aging is effected for a particular user if his encrypted password in the password file is followed by a comma and a non-null string of characters from the above alphabet. (Such a string must be introduced in the first instance by the super-user.)

The first character of the age, *M* say, denotes the maximum number of weeks for which a password is valid. A user who attempts to login after his password has expired is forced to supply a new one. The next character, *m* say, denotes the minimum period in weeks which must expire before the password may be changed. The remaining characters define the week (counted from the beginning of 1970) when the password was last changed. (A null string is equivalent to zero.) *M* and *m* have numerical values in the range 0–63 that correspond to the 64-character alphabet shown above (i.e., **/** = 1 week; **z** = 63 weeks). If *m* = *M* = 0 (derived from the string **.** or **..**) the user is forced to change his password the next time he logs in (and the "age" disappears from his entry in the password file). If *m* > *M* (signified, e.g., by the string **./**) only the super-user is able to change the password.

FILES

/etc/passwd

SEE ALSO

getpwent(3C), group(4), login(1), passwd(1)

NAME

plot – graphics interface

DESCRIPTION

Files of this format are produced by routines described in *plot*(3X) and are interpreted for various devices by commands described in *tplot*(1G). A graphics file is a stream of plotting instructions. Each instruction consists of an ASCII letter usually followed by bytes of binary information. The instructions are executed in order. A point is designated by four bytes representing the **x** and **y** values; each value is a signed integer. The last designated point in an **l**, **m**, **n**, or **p** instruction becomes the "current point" for the next instruction.

Each of the following descriptions begins with the name of the corresponding routine in *plot*(3X).

m move: The next four bytes give a new current point.

n cont: Draw a line from the current point to the point given by the next four bytes [see *tplot*(1G)].

p point: Plot the point given by the next four bytes.

l line: Draw a line from the point given by the next four bytes to the point given by the following four bytes.

t label: Place the following ASCII string so that its first character falls on the current point. The string is terminated by a new-line.

e erase: Start another frame of output.

f linemod: Take the following string, up to a new-line, as the style for drawing further lines. The styles are "dotted", "solid", "longdashed", "shortdashed", and "dotdashed". Effective only for the **–T4014** and **–Tver** options of *tplot*(1G) (TEKTRONIX 4014 terminal and Versatec plotter).

s space: The next four bytes give the lower left corner of the plotting area; the following four give the upper right corner. The plot will be magnified or reduced to fit the device as closely as possible.

Space settings that exactly fill the plotting area with unity scaling appear below for devices supported by the filters of *tplot*(1G). The upper limit is just outside the plotting area. In every case the plotting area is taken to be square; points outside may be displayable on devices whose face is not square.

DASI 300	space(0, 0, 4096, 4096);
DASI 300s	space(0, 0, 4096, 4096);
DASI 450	space(0, 0, 4096, 4096);
TEKTRONIX 4014	space(0, 0, 3120, 3120);
Versatec plotter	space(0, 0, 2048, 2048);

SEE ALSO

plot(3X), gps(4), term(5).
graph(1G), tplot(1G) in the *User's Reference Manual.*

WARNING

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.

NAME

pnch – file format for card images

DESCRIPTION

The PNCH format is a convenient representation for files consisting of card images in an arbitrary code.

A PNCH file is a simple concatenation of card records. A card record consists of a single control byte followed by a variable number of data bytes. The control byte specifies the number (which must lie in the range 0-80) of data bytes that follow. The data bytes are 8-bit codes that constitute the card image. If there are fewer than 80 data bytes, it is understood that the remainder of the card image consists of trailing blanks.

NAME

profile – setting up an environment at login time

SYNOPSIS

/etc/profile
$HOME/.profile

DESCRIPTION

Commands in these files are executed as part of their login sequence for all users who have the shell, *sh*(1), as their login command.

/etc/profile allows the system administrator to perform services for all users who use the shell, as opposed to the C-shell, *csh*(1). Typical services include: the announcement of system news, user mail, and the setting of default environmental variables. It is not unusual for */etc/profile* to execute special actions for the **root** login or the *su*(1) command. Computers running outside the Eastern time zone should have the line

```
    . /etc/TIMEZONE
```

included early in */etc/profile* (see timezone(4)).

The file *$HOME/.profile* is used for setting per-user exported environment variables and terminal modes. The following example is typical (except for the comments):

```
# Make some environment variables global
export MAIL PATH TERM
# Set file creation mask
umask 027
# Tell me when new mail comes in
MAIL=/usr/mail/$LOGNAME
# Add my /bin directory to the shell search sequence
PATH=$PATH:$HOME/bin
# Set terminal type
while :
do          echo "terminal: \c"
            read TERM
            if [ -f ${TERMINFO:-/usr/lib/terminfo}/?/$TERM ]
            then break
            elif [ -f /usr/lib/terminfo/?/$TERM ]
            then break
            else echo "invalid term $TERM" 1>&2
            fi
done
# Initialize the terminal and set tabs
# The environmental variable TERM must have been exported
# before the "tput init" command is executed.
tput init
# Set the erase character to backspace
stty erase '^H' echoe
```

FILES

/etc/TIMEZONE	timezone environment
$HOME/.profile	user-specific environment
/etc/profile	system-wide environment

SEE ALSO

env(1), environ(5), login(1), mail(1), sh(1), stty(1), su(1), su(1M), term(5), terminfo(4), timezone(4), tput(1)

NOTES

Care must be taken in providing system-wide services in */etc/profile*. Personal *.profile* files are better for serving all but the most global needs.

NAME

protocols – protocol name data base

DESCRIPTION

The *protocols* file contains information regarding the known protocols used in the DARPA Internet. For each protocol a single line should be present with the following information:

official protocol name
protocol number
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

Protocol names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/protocols

SEE ALSO

getprotoent(3N)

BUGS

A name server should be used instead of a static file.

NAME

queuedefs – at/batch/cron queue description file

SYNOPSIS

/usr/lib/cron/queuedefs

DESCRIPTION

The *queuedefs* file describes the characteristics of the queues managed by *cron*(1M). Each non-comment line in this file describes one queue. The format of the lines are as follows:

q.[*nobj***j**][*nice***n**][*nwait***w**]

The fields in this line are

q	The name of the queue. **a** is the default queue for jobs started by *at*(1); **b** is the default queue for jobs started by *batch*(1); **c** is the default queue for jobs run from a **crontab** file.
njob	The maximum number of jobs that can be run simultaneously in that queue; if more than *njobs* are ready to run, only the first *njobs* will be run, and the others will be run as jobs that are currently running terminate. The default value is 100.
nice	The *nice*(1) value to give to all jobs in that queue that are not run with a user ID of super-user. The default value is 2.
nwait	The number of seconds to wait before rescheduling a job that was deferred because more than *njobs* were running in that job's queue, or because more than 25 jobs were running in all the queues. The default value is 60.

Lines beginning with # are comments, and are ignored.

EXAMPLE

a.4j1n b.2j2n90w

This file specifies that the **a** queue, for *at* jobs, can have up to 4 jobs running simultaneously; those jobs will be run with a *nice* value of 1. As no *nwait* value was given, if a job cannot be run because too many other jobs are running, *cron* will wait 60 seconds before trying again to run it. The **b** queue, for *batch* jobs, can have up to 2 jobs running simultaneously; those jobs will be run with a *nice* value of 2. If a job cannot be run because too many other jobs are running, *cron* will wait 90 seconds before trying again to run it. All other queues can have up to 100 jobs running simultaneously; they will be run with a *nice* value of 2, and if a job cannot be run because too many other jobs are running, *cron* will wait 60 seconds before trying again to run it.

FILES

/usr/lib/cron/queuedefs

SEE ALSO

cron(1M)

NAME

rhosts – host name data base

DESCRIPTION

The *rhosts* file contains information regarding the known hosts on the network. For each host a single line should be present with the following information.
Internet address
official host name
aliases

Internet addresses are specified in the conventional "." notation using the *inet_addr*() routine from the Internet address manipulation library, *inet*(3N). Host names may contain any printable character other than a space, tab, newline, or comment character.

Items are separated by any number of blanks and/or tab characters. Characters between a "#" and the end of the line are comments and are not interpreted by routines which search the file. For hosts residing on the ARPANET, this file is normally created from the official host data base maintained at the Network Information Control Center (NIC), although local changes may be required to bring it up to date regarding unofficial aliases and/or unknown hosts. As the database maintained at NIC is incomplete, use of the name server is recommended for sites on the DARPA Internet.

When the name server *named*(8) is in use, this file provides a backup when the name server is not running. For the name server, it is suggested that only a few addresses be included in this file. These include addresses for the local interfaces that *ifconfig*(8C) needs at boot time and a few machines on the local network.

EXAMPLES

A properly functioning network is extremely sensitive to the contents of /etc/hosts. Below are examples of the minimum requirements for /etc/hosts on a network with only two hosts, **fred** and **ethyl**.

The same host file may be used on both machines:

```
127.0.0.1          loop localhost
address1  fred
address2  ethyl
```

The two addresses, *address1* and *address2* must refer to the same network, as described in *inet*(3N). The network must also appear in /etc/networks, as described in *networks*(5).

FILES

/etc/hosts

SEE ALSO

gethostent(3N), inet(3N), networks(5), ifconfig(8c), named(8)

NAME

sccsfile – format of SCCS file

DESCRIPTION

An SCCS (Source Code Control System) file is an ASCII file. It consists of six logical parts: the *checksum*, the *delta table* (contains information about each delta), *user names* (contains login names or numerical group IDs of users who may add deltas), *flags* (contains definitions of internal keywords), *comments* (contains arbitrary descriptive information about the file), and the *body* (contains the actual text lines intermixed with control lines).

Throughout an SCCS file there are lines which begin with the **ASCII SOH** (start of heading) character (octal 001). This character is hereafter referred to as *the control character* and is represented graphically as @. Any line described below which is not depicted as beginning with the control character is prevented from beginning with the control character.

Entries of the form DDDDD represent a five-digit string (a number between 00000 and 99999).

Each logical part of an SCCS file is described in detail below.

Checksum

The checksum is the first line of an SCCS file. The form of the line is:

```
@hDDDDD
```

The value of the checksum is the sum of all characters, except those of the first line. The **@h** provides a *magic number* of (octal) 064001.

Delta table

The delta table consists of a variable number of entries of the form:

```
@s DDDDD/DDDDD/DDDDD
@d <type> <SCCS ID> yr/mo/da hr:mi:se <pgmr> DDDDD DDDDD
@i DDDDD ...
@x DDDDD ...
@g DDDDD ...
@m <MR number>
 .
 .
 .
@c <comments> ...
 .
 .
 .
@e
```

The first line (**@s**) contains the number of lines inserted/deleted/unchanged, respectively. The second line (**@d**) contains the type of the delta (currently, normal: **D**, and removed: **R**), the SCCS ID of the delta, the date and time of creation of the delta, the login name corresponding to the real user ID at the time the delta was created, and the serial numbers of the delta and its predecessor, respectively.

The **@i**, **@x**, and **@g** lines contain the serial numbers of deltas included, excluded, and ignored, respectively. These lines are optional.

The **@m** lines (optional) each contain one MR number associated with the delta; the **@c** lines contain comments associated with the delta.

The @**e** line ends the delta table entry.

User names

The list of login names or numerical group IDs of users who may add deltas to the file, separated by new-lines. The lines containing these login names or numerical group IDs are surrounded by the bracketing lines @**u** and @**U**. An empty list allows anyone to make a delta. Any line starting with a **!** prohibits the succeeding group or user from making deltas.

Flags

Keywords used internally. [See *admin*(1) for more information on their use.] Each flag line takes the form:

```
@f <flag>        <optional text>
```

The following flags are defined:

```
@f t    <type of program>
@f v    <program name>
@f i    <keyword string>
@f b
@f m    <module name>
@f f    <floor>
@f c    <ceiling>
@f d    <default-sid>
@f n
@f j
@f l    <lock-releases>
@f q    <user defined>
@f z    <reserved for use in interfaces>
```

The **t** flag defines the replacement for the %Y% identification keyword. The **v** flag controls prompting for MR numbers in addition to comments; if the optional text is present it defines an MR number validity checking program. The **i** flag controls the warning/error aspect of the "No id keywords" message. When the **i** flag is not present, this message is only a warning; when the **i** flag is present, this message causes a "fatal" error (the file is not gotten, or the delta is not made). When the **b** flag is present the **–b** keyletter may be used on the *get* command to cause a branch in the delta tree. The **m** flag defines the first choice for the replacement text of the %M% identification keyword. The **f** flag defines the "floor" release; the release below which no deltas may be added. The **c** flag defines the "ceiling" release; the release above which no deltas may be added. The **d** flag defines the default SID to be used when none is specified on a *get* command. The **n** flag causes *delta* to insert a "null" delta (a delta that applies *no* changes) in those releases that are skipped when a delta is made in a *new* release (e.g., when delta 5.1 is made after delta 2.7, releases 3 and 4 are skipped). The absence of the **n** flag causes skipped releases to be completely empty. The **j** flag causes *get* to allow concurrent edits of the same base SID. The **l** flag defines a *list* of releases that are *locked* against editing [*get*(1) with the **–e** keyletter]. The **q** flag defines the replacement for the %Q% identification keyword. The **z** flag is used in certain specialized interface programs.

Comments

Arbitrary text is surrounded by the bracketing lines @**t** and @**T**. The comments section typically contains a description of the file's purpose.

Body

The body consists of text lines and control lines. Text lines do not begin with the control character, control lines do. There are three kinds of control lines:

insert, *delete*, and *end*, represented by:

```
@I DDDDD
@D DDDDD
@E DDDDD
```

respectively. The digit string is the serial number corresponding to the delta for the control line.

SEE ALSO

admin(1), delta(1), get(1), prs(1)

NAME

scr_dump – format of curses screen image file.

SYNOPSIS

scr_dump(file)

DESCRIPTION

The *curses*(3X) function *scr_dump*() copies the contents of the screen into a file. The format of the screen image is described below.

The name of the tty is 20 characters long and the modification time (the *mtime* of the tty that this is an image of) is of the type *time_t*. All other numbers and characters are stored as *chtype* (see <*curses.h*>). No newlines are stored between fields.

```
<magic number:  octal 0433>
<name of tty>
<mod time of tty>
<columns> <lines>
<line length> <chars in line>        for each line on the screen
<line length> <chars in line>
  .
  .
  .
<labels?>                            1, if soft screen labels are present
<cursor row> <cursor column>
```

Only as many characters as fit on a line are listed. For example, if the <*line length*> is **0**, there are no characters following <*line length*>. If <*labels?*> is TRUE, following it are

```
<number of labels>
<label width>
<chars in label 1>
<chars in label 2>
  .
  .
  .
```

SEE ALSO

curses(3X).

NAME

services – service name data base

DESCRIPTION

The *services* file contains information regarding the known services available in the DARPA Internet. For each service a single line should be present with the following information:

official service name
port number
protocol name
aliases

Items are separated by any number of blanks and/or tab characters. The port number and protocol name are considered a single *item*; a "/" is used to separate the port and protocol (e.g. "512/tcp"). A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

Service names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/services

SEE ALSO

getservent(3N)

BUGS

A name server should be used instead of a static file.

NAME

tar – tape archive file format

DESCRIPTION

Tar, (the tape archive command) dumps several files into one, in a medium suitable for transportation.

A "tar tape" or file is a series of blocks. Each block is of size TBLOCK. A file on the tape is represented by a header block which describes the file, followed by zero or more blocks which give the contents of the file. At the end of the tape are two blocks filled with binary zeros, as an end-of-file indicator.

The blocks are grouped for physical I/O operations. Each group of *n* blocks (where *n* is set by the **b** keyletter on the *tar*(1) command line — default is 20 blocks) is written with a single system call; on nine-track tapes, the result of this write is a single tape record. The last group is always written at the full size, so blocks after the two zero blocks contain random data. On reading, the specified or default group size is used for the first read, but if that read returns less than a full tape block, the reduced block size is used for further reads.

The header block looks like:

```
#define TBLOCK      512
#define NAMSIZ      100

union hblock {
        char dummy[TBLOCK];
        struct header {
                char name[NAMSIZ];
                char mode[8];
                char uid[8];
                char gid[8];
                char size[12];
                char mtime[12];
                char chksum[8];
                char linkflag;
                char linkname[NAMSIZ];
        } dbuf;
};
```

Name is a null-terminated string. The other fields are zero-filled octal numbers in ASCII. Each field (of width w) contains w-2 digits, a space, and a null, except *size* and *mtime,* which do not contain the trailing null and *chksum* which has a null followed by a space. *Name* is the name of the file, as specified on the *tar* command line. Files dumped because they were in a directory which was named in the command line have the directory name as prefix and */filename* as suffix. *Mode* is the file mode, with the top bit masked off. *Uid* and *gid* are the user and group numbers which own the file. *Size* is the size of the file in bytes. Links and symbolic links are dumped with this field specified as zero. *Mtime* is the modification time of the file at the time it was dumped. *Chksum* is an octal ASCII value which represents the sum of all the bytes in the header block. When calculating the checksum, the *chksum* field is treated as if it were all blanks. *Linkflag* is NULL if the file is "normal" or a special file, ASCII '1' if it is an hard link, and ASCII '2' if it is a symbolic link. The name linked-to, if any, is in *linkname,* with a trailing null. Unused fields of the header are binary zeros (and are included in the checksum).

The first time a given i-node number is dumped, it is dumped as a regular file. The second and subsequent times, it is dumped as a link instead. Upon retrieval, if a link entry is retrieved, but not the file it was linked to, an error message is printed and the tape must be manually re-scanned to retrieve the linked-to file.

The encoding of the header is designed to be portable across machines.

SEE ALSO

tar(1)

BUGS

Names or linknames longer than NAMSIZ produce error reports and cannot be dumped.

NAME

term – format of compiled term file.

SYNOPSIS

/usr/lib/terminfo/?/*

DESCRIPTION

Compiled *terminfo*(4) descriptions are placed under the directory */usr/lib/terminfo*. In order to avoid a linear search of a huge UNIX system directory, a two-level scheme is used: */usr/lib/terminfo/c/name* where *name* is the name of the terminal, and *c* is the first character of *name*. Thus, **att4425** can be found in the file */usr/lib/terminfo/a/att4425*. Synonyms for the same terminal are implemented by multiple links to the same compiled file.

The format has been chosen so that it is the same on all hardware. An 8-bit byte is assumed, but no assumptions about byte ordering or sign extension are made. Thus, these binary *terminfo*(4) files can be transported to other hardware with 8-bit bytes.

Short integers are stored in two 8-bit bytes. The first byte contains the least significant 8 bits of the value, and the second byte contains the most significant 8 bits. (Thus, the value represented is 256*second+first.) The value **–1** is represented by **0377,0377**, and the value **–2** is represented by **0376,0377**; other negative values are illegal. Computers where this does not correspond to the hardware read the integers as two bytes and compute the result, making the compiled entries portable between machine types. The **–1** generally means that a capability is missing from this terminal. The **–2** means that the capability has been cancelled in the *terminfo*(4) source and also is to be considered missing.

The compiled file is created from the source file descriptions of the terminals (see the **–I** option of *infocmp*(1M)) by using the *terminfo*(4) compiler, *tic*(1M), and read by the routine *setupterm*( ). (See *curses*(3X).) The file is divided into six parts: the header, terminal names, boolean flags, numbers, strings, and string table.

The header section begins the file. This section contains six short integers in the format described below. These integers are (1) the magic number (octal **0432**); (2) the size, in bytes, of the names section; (3) the number of bytes in the boolean section; (4) the number of short integers in the numbers section; (5) the number of offsets (short integers) in the strings section; (6) the size, in bytes, of the string table.

The terminal names section comes next. It contains the first line of the *terminfo*(4) description, listing the various names for the terminal, separated by the bar (|) character (see *term*(5)). The section is terminated with an ASCII NUL character.

The boolean flags have one byte for each flag. This byte is either **0** or **1** as the flag is present or absent. The value of **2** means that the flag has been cancelled. The capabilities are given in the same order as the file **<term.h>**.

Between the boolean section and the number section, a null byte is inserted, if necessary, to ensure that the number section begins on an even byte. All short integers are aligned on a short word boundary.

The numbers section is similar to the boolean flags section. Each capability takes up two bytes, and is stored as a short integer. If the value represented is **–1** or **–2**, the capability is taken to be missing.

The strings section is also similar. Each capability is stored as a short integer, in the format above. A value of **–1** or **–2** means the capability is missing. Otherwise, the value is taken as an offset from the beginning of the string table. Special characters in ^X or \c notation are stored in their interpreted form, not the printing representation. Padding information ($<nn>) and parameter information (%x) are stored intact in uninterpreted form.

The final section is the string table. It contains all the values of string capabilities referenced in the string section. Each string is null terminated.

Note that it is possible for *setupterm*() to expect a different set of capabilities than are actually present in the file. Either the database may have been updated since *setupterm*() has been recompiled (resulting in extra unrecognized entries in the file) or the program may have been recompiled more recently than the database was updated (resulting in missing entries). The routine *setupterm*() must be prepared for both possibilities – this is why the numbers and sizes are included. Also, new capabilities must always be added at the end of the lists of boolean, number, and string capabilities.

As an example, an octal dump of the description for the AT&T Model 37 KSR is included:

```
37 | tty37 | AT&T model 37 teletype,
    hc, os, xon,
    bel=^G, cr=\r, cub1=\b, cud1=\n, cuu1=\E7, hd=\E9,
    hu=\E8, ind=\n,

0000000 032 001      \0 032  \0 013  \0 021 001   3  \0   3   7   |   t
0000020   t   y   3   7   |   A   T   &   T       m   o   d   e   l
0000040   3   7       t   e   l   e   t   y   p   e  \0  \0  \0  \0  \0
0000060  \0  \0  \0 001  \0  \0  \0  \0  \0  \0  \0 001  \0  \0  \0  \0
0000100 001  \0  \0  \0  \0  \0 377 377 377 377 377 377 377 377 377 377
0000120 377 377 377 377 377 377 377 377 377 377 377 377 377 377   &  \0
0000140      \0 377 377 377 377 377 377 377 377 377 377 377 377 377 377
0000160 377 377   "  \0 377 377 377 377   (  \0 377 377 377 377 377 377
0000200 377 377   0  \0 377 377 377 377 377 377 377 377   -  \0 377 377
0000220 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377
*
0000520 377 377 377 377 377 377 377 377 377 377 377 377 377 377   $  \0
0000540 377 377 377 377 377 377 377 377 377 377 377 377 377 377   *  \0
0000560 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377
*
0001160 377 377 377 377 377 377 377 377 377 377 377 377 377 377   3   7
0001200   |   t   t   y   3   7   |   A   T   &   T       m   o   d   e
0001220   l       3   7       t   e   l   e   t   y   p   e  \0  \r  \0
0001240  \n  \0  \n  \0 007  \0  \b  \0 033   8  \0 033   9  \0 033   7
0001260  \0  \0
0001261
```

Some limitations: total compiled entries cannot exceed 4096 bytes; all entries in the name field cannot exceed 128 bytes.

FILES

/usr/lib/terminfo/?/* compiled terminal description database
/usr/include/term.h *terminfo*(4) header file

SEE ALSO

curses(3X), infocmp(1M), term(5), terminfo(4)

NAME

terminfo – terminal capability data base

SYNOPSIS

/usr/lib/terminfo/?/*

DESCRIPTION

terminfo is a compiled database (see *tic*(1M)) describing the capabilities of terminals. Terminals are described in *terminfo* source descriptions by giving a set of capabilities which they have, by describing how operations are performed, by describing padding requirements, and by specifying initialization sequences. This database is used by applications programs, such as *vi*(1) and *curses*(3X), so they can work with a variety of terminals without changes to the programs. To obtain the source description for a terminal, use the **–I** option of *infocmp*(1M).

Entries in *terminfo* source files consist of a number of comma-separated fields. White space after each comma is ignored. The first line of each terminal description in the *terminfo* database gives the name by which *terminfo* knows the terminal, separated by bar (|) characters. The first name given is the most common abbreviation for the terminal (this is the one to use to set the environment variable TERM in *$HOME/.profile*; see *profile*(4)), the last name given should be a long name fully identifying the terminal, and all others are understood as synonyms for the terminal name. All names but the last should contain no blanks and must be unique in the first 14 characters; the last name may contain blanks for readability.

Terminal names (except for the last, verbose entry) should be chosen using the following conventions. The particular piece of hardware making up the terminal should have a root name chosen, for example, for the AT&T 4425 terminal, **att4425**. Hardware modes, or user preferences, should be indicated by appending a hyphen and an indicator of the mode. See *term*(5) for examples and more information on choosing names and synonyms.

CAPABILITIES

In the table below, the **Variable** is the name by which the C programmer (at the *terminfo* level) accesses the capability. The **Capname** is the short name for this variable used in the text of the database. It is used by a person updating the database and by the *tput*(1) command when asking what the value of the capability is for a particular terminal. The **Termcap Code** is a two-letter code that corresponds to the old *termcap* capability name.

Capability names have no set length limit, but an informal limit of 5 characters has been adopted to keep them short. Whenever possible, names are chosen to be the same as or similar to the ANSI X3.64-1979 standard. Semantics are also intended to match those of the specification.

All string capabilities listed below may have padding specified, with the exception of those used for input. Input capabilities, listed under the **Strings** section in the table below, use names beginning with **key_**. The following indicators may appear at the end of the **Description** for a variable.

(G) indicates that the string is passed through **tparm**( ) with parameters (parms) as given ($\#_i$).

(*) indicates that padding may be based on the number of lines affected.

($\#_i$) indicates the i^{th} parameter.

Variable	Cap-name	Termcap Code	Description
Booleans:			
auto_left_margin	bw	bw	**cub**1 wraps from column 0 to last column
auto_right_margin	am	am	Terminal has automatic margins
no_esc_ctlc	xsb	xb	Beehive (f1=escape, f2=ctrl C)
ceol_standout_glitch	xhp	xs	Standout not erased by overwriting (hp)
eat_newline_glitch	xenl	xn	Newline ignored after 80 cols (*Concept*)
erase_overstrike	eo	eo	Can erase overstrikes with a blank
generic_type	gn	gn	Generic line type (e.g. dialup, switch).
hard_copy	hc	hc	Hardcopy terminal
hard_cursor	chts	HC	Cursor is hard to see.
has_meta_key	km	km	Has a meta key (shift, sets parity bit)
has_status_line	hs	hs	Has extra "status line"
insert_null_glitch	in	in	Insert mode distinguishes nulls
memory_above	da	da	Display may be retained above the screen
memory_below	db	db	Display may be retained below the screen
move_insert_mode	mir	mi	Safe to move while in insert mode
move_standout_mode	msgr	ms	Safe to move in standout modes
needs_xon_xoff	nxon	nx	Padding won't work, xon/xoff required
non_rev_rmcup	nrrmc	NR	**smcup** does not reverse **rmcup**
no_pad_char	npc	NP	Pad character doesn't exist
over_strike	os	os	Terminal overstrikes on hard-copy terminal
prtr_silent	mc5i	5i	Printer won't echo on screen.
status_line_esc_ok	eslok	es	Escape can be used on the status line
dest_tabs_magic_smso	xt	xt	Destructive tabs, magic **smso** char (t1061)
tilde_glitch	hz	hz	Hazeltine; can't print tildes(~)
transparent_underline	ul	ul	Underline character overstrikes
xon_xoff	xon	xo	Terminal uses xon/xoff handshaking
Numbers:			
columns	cols	co	Number of columns in a line
init_tabs	it	it	Tabs initially every # spaces.
label_height	lh	lh	Number of rows in each label
label_width	lw	lw	Number of cols in each label
lines	lines	li	Number of lines on screen or page
lines_of_memory	lm	lm	Lines of memory if > **lines**; **0** means varies
magic_cookie_glitch	xmc	sg	Number blank chars left by **smso** or **rmso**
num_labels	nlab	Nl	Number of labels on screen (start at 1)
padding_baud_rate	pb	pb	Lowest baud rate where padding needed
virtual_terminal	vt	vt	Virtual terminal number (UNIX system)
width_status_line	wsl	ws	Number of columns in status line
Strings:			
acs_chars	acsc	ac	Graphic charset pairs aAbBcC - def=vt100+
back_tab	cbt	bt	Back tab
bell	bel	bl	Audible signal (bell)
carriage_return	cr	cr	Carriage return (*)
change_scroll_region	csr	cs	Change to lines #1 thru #2 (vt100) (G)
char_padding	rmp	rP	Like **ip** but when in replace mode
clear_all_tabs	tbc	ct	Clear all tab stops

clear_margins	mgc	MC	Clear left and right soft margins
clear_screen	clear	cl	Clear screen and home cursor (*)
clr_bol	el1	cb	Clear to beginning of line, inclusive
clr_eol	el	ce	Clear to end of line
clr_eos	ed	cd	Clear to end of display (*)
column_address	hpa	ch	Horizontal position absolute (G)
command_character	cmdch	CC	Term. settable cmd char in prototype
cursor_address	cup	cm	Cursor motion to row #1 col #2 (G)
cursor_down	cud1	do	Down one line
cursor_home	home	ho	Home cursor (if no **cup**)
cursor_invisible	civis	vi	Make cursor invisible
cursor_left	cub1	le	Move cursor left one space.
cursor_mem_address	mrcup	CM	Memory relative cursor addressing (G)
cursor_normal	cnorm	ve	Make cursor appear normal (undo **vs/vi**)
cursor_right	cuf1	nd	Non-destructive space (cursor right)
cursor_to_ll	ll	ll	Last line, first column (if no **cup**)
cursor_up	cuu1	up	Upline (cursor up)
cursor_visible	cvvis	vs	Make cursor very visible
delete_character	dch1	dc	Delete character (*)
delete_line	dl1	dl	Delete line (*)
dis_status_line	dsl	ds	Disable status line
down_half_line	hd	hd	Half-line down (forward 1/2 linefeed)
ena_acs	enacs	eA	Enable alternate char set
enter_alt_charset_mode	smacs	as	Start alternate character set
enter_am_mode	smam	SA	Turn on automatic margins
enter_blink_mode	blink	mb	Turn on blinking
enter_bold_mode	bold	md	Turn on bold (extra bright) mode
enter_ca_mode	smcup	ti	String to begin programs that use **cup**
enter_delete_mode	smdc	dm	Delete mode (enter)
enter_dim_mode	dim	mh	Turn on half-bright mode
enter_insert_mode	smir	im	Insert mode (enter);
enter_protected_mode	prot	mp	Turn on protected mode
enter_reverse_mode	rev	mr	Turn on reverse video mode
enter_secure_mode	invis	mk	Turn on blank mode (chars invisible)
enter_standout_mode	smso	so	Begin standout mode
enter_underline_mode	smul	us	Start underscore mode
enter_xon_mode	smxon	SX	Turn on xon/xoff handshaking
erase_chars	ech	ec	Erase #1 characters (G)
exit_alt_charset_mode	rmacs	ae	End alternate character set
exit_am_mode	rmam	RA	Turn off automatic margins
exit_attribute_mode	sgr0	me	Turn off all attributes
exit_ca_mode	rmcup	te	String to end programs that use **cup**
exit_delete_mode	rmdc	ed	End delete mode
exit_insert_mode	rmir	ei	End insert mode;
exit_standout_mode	rmso	se	End standout mode
exit_underline_mode	rmul	ue	End underscore mode
exit_xon_mode	rmxon	RX	Turn off xon/xoff handshaking
flash_screen	flash	vb	Visible bell (may not move cursor)
form_feed	ff	ff	Hardcopy terminal page eject (*)
from_status_line	fsl	fs	Return from status line
init_1string	is1	i1	Terminal initialization string
init_2string	is2	is	Terminal initialization string

init_3string	is3	i3	Terminal initialization string
init_file	if	if	Name of initialization file containing **is**
init_prog	iprog	iP	Path name of program for init.
insert_character	ich1	ic	Insert character
insert_line	il1	al	Add new blank line (*)
insert_padding	ip	ip	Insert pad after character inserted (*)
key_a1	ka1	K1	KEY_A1, 0534, Upper left of keypad
key_a3	ka3	K3	KEY_A3, 0535, Upper right of keypad
key_b2	kb2	K2	KEY_B2, 0536, Center of keypad
key_backspace	kbs	kb	KEY_BACKSPACE, 0407, Sent by backspace key
key_beg	kbeg	@1	KEY_BEG, 0542, Sent by beg(inning) key
key_btab	kcbt	kB	KEY_BTAB, 0541, Sent by back-tab key
key_c1	kc1	K4	KEY_C1, 0537, Lower left of keypad
key_c3	kc3	K5	KEY_C3, 0540, Lower right of keypad
key_cancel	kcan	@2	KEY_CANCEL, 0543, Sent by cancel key
key_catab	ktbc	ka	KEY_CATAB, 0526, Sent by clear-all-tabs key
key_clear	kclr	kC	KEY_CLEAR, 0515, Sent by clear-screen or erase key
key_close	kclo	@3	KEY_CLOSE, 0544, Sent by close key
key_command	kcmd	@4	KEY_COMMAND, 0545, Sent by cmd (command) key
key_copy	kcpy	@5	KEY_COPY, 0546, Sent by copy key
key_create	kcrt	@6	KEY_CREATE, 0547, Sent by create key
key_ctab	kctab	kt	KEY_CTAB, 0525, Sent by clear-tab key
key_dc	kdch1	kD	KEY_DC, 0512, Sent by delete-character key
key_dl	kdl1	kL	KEY_DL, 0510, Sent by delete-line key
key_down	kcud1	kd	KEY_DOWN, 0402, Sent by terminal down-arrow key
key_eic	krmir	kM	KEY_EIC, 0514, Sent by **rmir** or **smir** in insert mode
key_end	kend	@7	KEY_END, 0550, Sent by end key
key_enter	kent	@8	KEY_ENTER, 0527, Sent by enter/send key
key_eol	kel	kE	KEY_EOL, 0517, Sent by clear-to-end-of-line key
key_eos	ked	kS	KEY_EOS, 0516, Sent by clear-to-end-of-screen key
key_exit	kext	@9	KEY_EXIT, 0551, Sent by exit key
key_f0	kf0	k0	KEY_F(0), 0410, Sent by function key f0
key_f1	kf1	k1	KEY_F(1), 0411, Sent by function key f1
key_f2	kf2	k2	KEY_F(2), 0412, Sent by function key f2
key_f3	kf3	k3	KEY_F(3), 0413, Sent by function key f3
key_f4	kf4	k4	KEY_F(4), 0414, Sent by function key f4
key_f5	kf5	k5	KEY_F(5), 0415, Sent by function key f5
key_f6	kf6	k6	KEY_F(6), 0416, Sent by function key f6
key_f7	kf7	k7	KEY_F(7), 0417, Sent by function key f7
key_f8	kf8	k8	KEY_F(8), 0420, Sent by function key f8
key_f9	kf9	k9	KEY_F(9), 0421, Sent by function key f9
key_f10	kf10	k;	KEY_F(10), 0422, Sent by function key f10
key_f11	kf11	F1	KEY_F(11), 0423, Sent by function key f11
key_f12	kf12	F2	KEY_F(12), 0424, Sent by function key f12
key_f13	kf13	F3	KEY_F(13), 0425, Sent by function key f13
key_f14	kf14	F4	KEY_F(14), 0426, Sent by function key f14
key_f15	kf15	F5	KEY_F(15), 0427, Sent by function key f15
key_f16	kf16	F6	KEY_F(16), 0430, Sent by function key f16

key_f17	kf17	F7	KEY_F(17), 0431, Sent by function key f17
key_f18	kf18	F8	KEY_F(18), 0432, Sent by function key f18
key_f19	kf19	F9	KEY_F(19), 0433, Sent by function key f19
key_f20	kf20	FA	KEY_F(20), 0434, Sent by function key f20
key_f21	kf21	FB	KEY_F(21), 0435, Sent by function key f21
key_f22	kf22	FC	KEY_F(22), 0436, Sent by function key f22
key_f23	kf23	FD	KEY_F(23), 0437, Sent by function key f23
key_f24	kf24	FE	KEY_F(24), 0440, Sent by function key f24
key_f25	kf25	FF	KEY_F(25), 0441, Sent by function key f25
key_f26	kf26	FG	KEY_F(26), 0442, Sent by function key f26
key_f27	kf27	FH	KEY_F(27), 0443, Sent by function key f27
key_f28	kf28	FI	KEY_F(28), 0444, Sent by function key f28
key_f29	kf29	FJ	KEY_F(29), 0445, Sent by function key f29
key_f30	kf30	FK	KEY_F(30), 0446, Sent by function key f30
key_f31	kf31	FL	KEY_F(31), 0447, Sent by function key f31
key_f32	kf32	FM	KEY_F(32), 0450, Sent by function key f32
key_f33	kf33	FN	KEY_F(13), 0451, Sent by function key f13
key_f34	kf34	FO	KEY_F(34), 0452, Sent by function key f34
key_f35	kf35	FP	KEY_F(35), 0453, Sent by function key f35
key_f36	kf36	FQ	KEY_F(36), 0454, Sent by function key f36
key_f37	kf37	FR	KEY_F(37), 0455, Sent by function key f37
key_f38	kf38	FS	KEY_F(38), 0456, Sent by function key f38
key_f39	kf39	FT	KEY_F(39), 0457, Sent by function key f39
key_f40	kf40	FU	KEY_F(40), 0460, Sent by function key f40
key_f41	kf41	FV	KEY_F(41), 0461, Sent by function key f41
key_f42	kf42	FW	KEY_F(42), 0462, Sent by function key f42
key_f43	kf43	FX	KEY_F(43), 0463, Sent by function key f43
key_f44	kf44	FY	KEY_F(44), 0464, Sent by function key f44
key_f45	kf45	FZ	KEY_F(45), 0465, Sent by function key f45
key_f46	kf46	Fa	KEY_F(46), 0466, Sent by function key f46
key_f47	kf47	Fb	KEY_F(47), 0467, Sent by function key f47
key_f48	kf48	Fc	KEY_F(48), 0470, Sent by function key f48
key_f49	kf49	Fd	KEY_F(49), 0471, Sent by function key f49
key_f50	kf50	Fe	KEY_F(50), 0472, Sent by function key f50
key_f51	kf51	Ff	KEY_F(51), 0473, Sent by function key f51
key_f52	kf52	Fg	KEY_F(52), 0474, Sent by function key f52
key_f53	kf53	Fh	KEY_F(53), 0475, Sent by function key f53
key_f54	kf54	Fi	KEY_F(54), 0476, Sent by function key f54
key_f55	kf55	Fj	KEY_F(55), 0477, Sent by function key f55
key_f56	kf56	Fk	KEY_F(56), 0500, Sent by function key f56
key_f57	kf57	Fl	KEY_F(57), 0501, Sent by function key f57
key_f58	kf58	Fm	KEY_F(58), 0502, Sent by function key f58
key_f59	kf59	Fn	KEY_F(59), 0503, Sent by function key f59
key_f60	kf60	Fo	KEY_F(60), 0504, Sent by function key f60
key_f61	kf61	Fp	KEY_F(61), 0505, Sent by function key f61
key_f62	kf62	Fq	KEY_F(62), 0506, Sent by function key f62
key_f63	kf63	Fr	KEY_F(63), 0507, Sent by function key f63
key_find	kfnd	@0	KEY_FIND, 0552, Sent by find key
key_help	khlp	%1	KEY_HELP, 0553, Sent by help key
key_home	khome	kh	KEY_HOME, 0406, Sent by home key
key_ic	kich1	kI	KEY_IC, 0513, Sent by ins-char/enter ins-mode key

key_il	kil1	kA	KEY_IL, 0511, Sent by insert-line key
key_left	kcub1	kl	KEY_LEFT, 0404, Sent by terminal left-arrow key
key_ll	kll	kH	KEY_LL, 0533, Sent by home-down key
key_mark	kmrk	%2	KEY_MARK, 0554, Sent by mark key
key_message	kmsg	%3	KEY_MESSAGE, 0555, Sent by message key
key_move	kmov	%4	KEY_MOVE, 0556, Sent by move key
key_next	knxt	%5	KEY_NEXT, 0557, Sent by next-object key
key_npage	knp	kN	KEY_NPAGE, 0522, Sent by next-page key
key_open	kopn	%6	KEY_OPEN, 0560, Sent by open key
key_options	kopt	%7	KEY_OPTIONS, 0561, Sent by options key
key_ppage	kpp	kP	KEY_PPAGE, 0523, Sent by previous-page key
key_previous	kprv	%8	KEY_PREVIOUS, 0562, Sent by previous-object key
key_print	kprt	%9	KEY_PRINT, 0532, Sent by print or copy key
key_redo	krdo	%0	KEY_REDO, 0563, Sent by redo key
key_reference	kref	&1	KEY_REFERENCE, 0564, Sent by ref(erence) key
key_refresh	krfr	&2	KEY_REFRESH, 0565, Sent by refresh key
key_replace	krpl	&3	KEY_REPLACE, 0566, Sent by replace key
key_restart	krst	&4	KEY_RESTART, 0567, Sent by restart key
key_resume	kres	&5	KEY_RESUME, 0570, Sent by resume key
key_right	kcuf1	kr	KEY_RIGHT, 0405, Sent by terminal right-arrow key
key_save	ksav	&6	KEY_SAVE, 0571, Sent by save key
key_sbeg	kBEG	&9	KEY_SBEG, 0572, Sent by shifted beginning key
key_scancel	kCAN	&0	KEY_SCANCEL, 0573, Sent by shifted cancel key
key_scommand	kCMD	*1	KEY_SCOMMAND, 0574, Sent by shifted command key
key_scopy	kCPY	*2	KEY_SCOPY, 0575, Sent by shifted copy key
key_screate	kCRT	*3	KEY_SCREATE, 0576, Sent by shifted create key
key_sdc	kDC	*4	KEY_SDC, 0577, Sent by shifted delete-char key
key_sdl	kDL	*5	KEY_SDL, 0600, Sent by shifted delete-line key
key_select	kslt	*6	KEY_SELECT, 0601, Sent by select key
key_send	kEND	*7	KEY_SEND, 0602, Sent by shifted end key
key_seol	kEOL	*8	KEY_SEOL, 0603, Sent by shifted clear-line key
key_sexit	kEXT	*9	KEY_SEXIT, 0604, Sent by shifted exit key
key_sf	kind	kF	KEY_SF, 0520, Sent by scroll-forward/down key
key_sfind	kFND	*0	KEY_SFIND, 0605, Sent by shifted find key
key_shelp	kHLP	#1	KEY_SHELP, 0606, Sent by shifted help key
key_shome	kHOM	#2	KEY_SHOME, 0607, Sent by shifted home key
key_sic	kIC	#3	KEY_SIC, 0610, Sent by shifted input key
key_sleft	kLFT	#4	KEY_SLEFT, 0611, Sent by shifted left-arrow key
key_smessage	kMSG	%a	KEY_SMESSAGE, 0612, Sent by shifted message key
key_smove	kMOV	%b	KEY_SMOVE, 0613, Sent by shifted move key
key_snext	kNXT	%c	KEY_SNEXT, 0614, Sent by shifted next key
key_soptions	kOPT	%d	KEY_SOPTIONS, 0615, Sent by shifted options key
key_sprevious	kPRV	%e	KEY_SPREVIOUS, 0616, Sent by shifted prev key
key_sprint	kPRT	%f	KEY_SPRINT, 0617, Sent by shifted print key
key_sr	kri	kR	KEY_SR, 0521, Sent by scroll-backward/up key
key_sredo	kRDO	%g	KEY_SREDO, 0620, Sent by shifted redo key

key_sreplace	kRPL	%h	KEY_SREPLACE, 0621, Sent by shifted replace key
key_sright	kRIT	%i	KEY_SRIGHT, 0622, Sent by shifted right-arrow key
key_srsume	kRES	%j	KEY_SRSUME, 0623, Sent by shifted resume key
key_ssave	kSAV	!1	KEY_SSAVE, 0624, Sent by shifted save key
key_ssuspend	kSPD	!2	KEY_SSUSPEND, 0625, Sent by shifted suspend key
key_stab	khts	kT	KEY_STAB, 0524, Sent by set-tab key
key_sundo	kUND	!3	KEY_SUNDO, 0626, Sent by shifted undo key
key_suspend	kspd	&7	KEY_SUSPEND, 0627, Sent by suspend key
key_undo	kund	&8	KEY_UNDO, 0630, Sent by undo key
key_up	kcuu1	ku	KEY_UP, 0403, Sent by terminal up-arrow key
keypad_local	rmkx	ke	Out of "keypad-transmit" mode
keypad_xmit	smkx	ks	Put terminal in "keypad-transmit" mode
lab_f0	lf0	l0	Labels on function key f0 if not f0
lab_f1	lf1	l1	Labels on function key f1 if not f1
lab_f2	lf2	l2	Labels on function key f2 if not f2
lab_f3	lf3	l3	Labels on function key f3 if not f3
lab_f4	lf4	l4	Labels on function key f4 if not f4
lab_f5	lf5	l5	Labels on function key f5 if not f5
lab_f6	lf6	l6	Labels on function key f6 if not f6
lab_f7	lf7	l7	Labels on function key f7 if not f7
lab_f8	lf8	l8	Labels on function key f8 if not f8
lab_f9	lf9	l9	Labels on function key f9 if not f9
lab_f10	lf10	la	Labels on function key f10 if not f10
label_off	rmln	LF	Turn off soft labels
label_on	smln	LO	Turn on soft labels
meta_off	rmm	mo	Turn off "meta mode"
meta_on	smm	mm	Turn on "meta mode" (8th bit)
newline	nel	nw	Newline (behaves like **cr** followed by **lf**)
pad_char	pad	pc	Pad character (rather than null)
parm_dch	dch	DC	Delete #1 chars (G*)
parm_delete_line	dl	DL	Delete #1 lines (G*)
parm_down_cursor	cud	DO	Move cursor down #1 lines. (G*)
parm_ich	ich	IC	Insert #1 blank chars (G*)
parm_index	indn	SF	Scroll forward #1 lines. (G)
parm_insert_line	il	AL	Add #1 new blank lines (G*)
parm_left_cursor	cub	LE	Move cursor left #1 spaces (G)
parm_right_cursor	cuf	RI	Move cursor right #1 spaces. (G*)
parm_rindex	rin	SR	Scroll backward #1 lines. (G)
parm_up_cursor	cuu	UP	Move cursor up #1 lines. (G*)
pkey_key	pfkey	pk	Prog funct key #1 to type string #2
pkey_local	pfloc	pl	Prog funct key #1 to execute string #2
pkey_xmit	pfx	px	Prog funct key #1 to xmit string #2
plab_norm	pln	pn	Prog label #1 to show string #2
print_screen	mc0	ps	Print contents of the screen
prtr_non	mc5p	pO	Turn on the printer for #1 bytes
prtr_off	mc4	pf	Turn off the printer
prtr_on	mc5	po	Turn on the printer
repeat_char	rep	rp	Repeat char #1 #2 times (G*)
req_for_input	rfi	RF	Send next input char (for ptys)
reset_1string	rs1	r1	Reset terminal completely to sane modes

reset_2string	rs2	r2	Reset terminal completely to sane modes
reset_3string	rs3	r3	Reset terminal completely to sane modes
reset_file	rf	rf	Name of file containing reset string
restore_cursor	rc	rc	Restore cursor to position of last sc
row_address	vpa	cv	Vertical position absolute (G)
save_cursor	sc	sc	Save cursor position.
scroll_forward	ind	sf	Scroll text up
scroll_reverse	ri	sr	Scroll text down
set_attributes	sgr	sa	Define the video attributes #1-#9 (G)
set_left_margin	smgl	ML	Set soft left margin
set_right_margin	smgr	MR	Set soft right margin
set_tab	hts	st	Set a tab in all rows, current column.
set_window	wind	wi	Current window is lines #1-#2 cols #3-#4 (G)
tab	ht	ta	Tab to next 8 space hardware tab stop.
to_status_line	tsl	ts	Go to status line, col #1 (G)
underline_char	uc	uc	Underscore one char and move past it
up_half_line	hu	hu	Half-line up (reverse 1/2 linefeed)
xoff_character	xoffc	XF	X-off character
xon_character	xonc	XN	X-on character

SAMPLE ENTRY

The following entry, which describes the *Concept*–100 terminal, is among the more complex entries in the *terminfo* file.

```
concept100 | c100 | concept | c104 | c100-4p | concept 100,
    am, db, eo, in, mir, ul, xenl,
    cols#80, lines#24, pb#9600, vt#8,
    bel=^G, blank=\EH, blink=\EC, clear=^L$<2*>,
    cnorm=\Ew, cr=^M$<9>, cub1=^H, cud1=^J,
    cuf1=\E=, cup=\Ea%p1%' '%+%c%p2%' '%+%c,
    cuu1=\E;, cvvis=\EW, dch1=\E^A$<16*>, dim=\EE,
    dl1=\E^B$<3*>, ed=\E^C$<16*>, el=\E^U$<16>,
    flash=\Ek$<20>\EK, ht=\t$<8>, il1=\E^R$<3*>,
    ind=^J, .ind=^J$<9>, ip=$<16*>,
    is2=\EU\Ef\E7\E5\E8\El\ENH\EK\E\0\Eo&\0\Eo\47\E,
    kbs=^h, kcub1=\E>, kcud1=\E<, kcuf1=\E=, kcuu1=\E;,
    kf1=\E5, kf2=\E6, kf3=\E7, khome=\E?,
    prot=\EI, rep=\Er%p1%c%p2%' '%+%c$<.2*>,
    rev=\ED, rmcup=\Ev\s\s\s\s$<6>\Ep\r\n,
    rmir=\E\0, rmkx=\Ex, rmso=\Ed\Ee, rmul=\Eg,
    rmul=\Eg, sgr0=\EN\0, smcup=\EU\Ev\s\s8p\Ep\r,
    smir=\E^P, smkx=\EX, smso=\EE\ED, smul=\EG,
```

Entries may continue onto multiple lines by placing white space at the beginning of each line except the first. Lines beginning with "#" are taken as comment lines. Capabilities in *terminfo* are of three types: boolean capabilities which indicate that the terminal supports some particular feature, numeric capabilities giving the size of the terminal or particular features, and string capabilities, which give a sequence that can be used to perform particular terminal operations.

Types of Capabilities

All capabilities have names. For instance, the fact that the *Concept* has *automatic margins* (i.e., an automatic return and linefeed when the end of a line is reached) is indicated by the capability **am**. Hence the description of the *Concept* includes **am**. Numeric capabilities are followed by the character '#' and then the value. Thus **cols**, which indicates the number of columns the terminal has, gives the value **80** for the

Concept. The value may be specified in decimal, octal or hexadecimal using normal C conventions.

Finally, string-valued capabilities, such as **el** (clear to end of line sequence) are given by the two- to five-character capname, an '=', and then a string ending at the next following comma. A delay in milliseconds may appear anywhere in such a capability, enclosed in **$<..>** brackets, as in **el=\EK$<3>**, and padding characters are supplied by **tputs**( ) (see *curses*(3X)) to provide this delay. The delay can be either a number, e.g., **20**, or a number followed by an '*' (i.e., **3***), a '/' (i.e., **5/**), or both (i.e., **10*/**). A '*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. (In the case of insert character, the factor is still the number of lines affected. This is always one unless the terminal supports **in** and the software uses it.) When a '*' is specified, it is sometimes useful to give a delay of the form **3.5** to specify a delay per unit to tenths of milliseconds. (Only one decimal place is allowed.) A '/' indicates that the padding is mandatory. Otherwise, if **xon** is defined for the terminal, the padding information is advisory and is only used for cost estimates or when the terminal is in raw mode. Mandatory padding is transmitted regardless of the setting of **xon**.

A number of escape sequences are provided in the string valued capabilities for easy encoding of characters there. Both **\E** and **\e** map to an ESCAPE character, ^*x* maps to a control–*x* for any appropriate *x*, and the sequences **\n**, **\l**, **\r**, **\t**, **\b**, **\f**, and **\s** give a newline, linefeed, return, tab, backspace, formfeed, and space, respectively. Other escapes include: **\^** for caret (^); \ for backslash (\); **\,** for comma (,); **\:** for colon (:); and **\0** for null. (**\0** actually produces **\200**, which does not terminate a string but behaves as a null character on most terminals.) Finally, characters may be given as three octal digits after a backslash (e.g., \123).

Sometimes individual capabilities must be commented out. To do this, put a period before the capability name. For example, see the second **ind** in the example above. Note that capabilities are defined in left-to-right order and, therefore, a prior definition overrides a later definition.

Preparing Descriptions

The most effective way to prepare a terminal description is by imitating the description of a similar terminal in *terminfo* and to build up a description gradually, using partial descriptions with **vi**(1) to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the **terminfo** file to describe it or the inability of **vi**(1) to work with that terminal. To test a new terminal description, set the environment variable TERMINFO to a pathname of a directory containing the compiled description you are working on and programs look there rather than in */usr/lib/terminfo*. To get the padding correctly for insert-line (if the terminal manufacturer did not document it) a severe test is to comment out **xon**, edit a large file at 9600 baud with **vi**(1), delete 16 or so lines from the middle of the screen, then hit the **u** key several times quickly. If the display is corrupted, more padding is usually needed. A similar test can be used for insert-character.

Basic Capabilities

The number of columns on each line for the terminal is given by the **cols** numeric capability. If the terminal has a screen, then the number of lines on the screen is given by the **lines** capability. If the terminal wraps around to the beginning of the next line when it reaches the right margin, then it should have the **am** capability. If the terminal can clear its screen, leaving the cursor in the home position, then this is given by the **clear** string capability. If the terminal overstrikes (rather than clearing a position when a character is struck over) then it should have the **os** capability. If the terminal is a printing terminal, with no soft copy unit, give it both **hc** and **os**. (**os** applies to storage scope terminals, such as Tektronix 4010 series, as well as hard-copy and APL terminals.) If there is a code to move the cursor to the left edge of the current row, give this as **cr**. (Normally this is carriage return, control-M.) If there is a code to produce an audible signal (bell, beep, etc) give this as **bel**. If the terminal uses the xon-xoff flow-control protocol, like most terminals, specify **xon**.

If there is a code to move the cursor one position to the left (such as backspace) that capability should be given as **cub1**. Similarly, codes to move to the right, up, and down should be given as **cuf1**, **cuu1**, and **cud1**. These local cursor motions should not alter the text they pass over; for example, you would not normally use "**cuf1**=\s" because the space would erase the character moved over.

A very important point here is that the local cursor motions encoded in *terminfo* are undefined at the left and top edges of a screen terminal. Programs should never attempt to backspace around the left edge, unless **bw** is given, and should never attempt to go up locally off the top. In order to scroll text up, a program goes to the bottom left corner of the screen and sends the **ind** (index) string.

To scroll text down, a program goes to the top left corner of the screen and sends the **ri** (reverse index) string. The strings **ind** and **ri** are undefined when not on their respective corners of the screen.

Parameterized versions of the scrolling sequences are **indn** and **rin** which have the same semantics as **ind** and **ri** except that they take one parameter, and scroll that many lines. They are also undefined except at the appropriate edge of the screen.

The **am** capability tells whether the cursor sticks at the right edge of the screen when text is output, but this does not necessarily apply to a **cuf1** from the last column. The only local motion defined from the left edge is if **bw** is given; then a **cub1** from the left edge moves to the right edge of the previous row. If **bw** is not given, the effect is undefined. This is useful for drawing a box around the edge of the screen, for example. If the terminal supports switch selectable automatic margins, the *terminfo* file usually assumes that this is on; i.e., **am**. If the terminal supports a command which moves to the first column of the next line, that command can be given as **nel** (newline). It does not matter if the command clears the remainder of the current line; so, if the terminal has no **cr** and **lf** it may still be possible to craft a working **nel** out of one or both of them.

These capabilities are enough to describe hardcopy and screen terminals. Thus the model 33 teletype is described as

```
33 | tty33 | tty | model 33 teletype,
        bel=^G, cols#72, cr=^M, cud1=^J, hc, ind=^J, os,
```

while the Lear Siegler ADM-3 is described as

```
adm3 | lsi adm3,
        am, bel=^G, clear=^Z, cols#80, cr=^M, cub1=^H, cud1=^J,
        ind=^J, lines#24,
```

Parameterized Strings

Cursor addressing and other strings requiring parameters in the terminal are described by a parameterized string capability, with **printf**(3S)-like escapes (%x) in it. For example, to address the cursor, the **cup** capability is given, using two parameters: the row and column to address to. (Rows and columns are numbered from zero and refer to the physical screen visible to the user, not to any unseen memory.) If the terminal has memory relative cursor addressing, that can be indicated by **mrcup**.

The parameter mechanism uses a stack and special % codes to manipulate it in the manner of a Reverse Polish Notation (postfix) calculator. Typically a sequence pushes one of the parameters onto the stack and then prints it in some format. Often more complex operations are necessary. Binary operations are accomplished in postfix form with the operands in the usual order. That is, to get x–5 one would use **%gx%{5}%–**.

The % encodings have the following meanings:

```
%%              outputs '%'
%[[:]flags][width[.precision]][doxXs]
                as in printf, flags are [–+#] and space
%c              print pop() gives %c

%p[1-9]         push i^th parm
%P[a-z]         set variable [a-z] to pop()
%g[a-z]         get variable [a-z] and push it
%'c'            push char constant c
%{nn}           push decimal constant nn
%l              push strlen(pop())

%+ %– %* %/ %m
                arithmetic (%m is mod): push(pop() op pop())
%& %| %^        bit operations: push(pop() op pop())
%= %> %<        logical operations: push(pop() op pop())
%A %O           logical operations: and, or
%! %~           unary operations: push(op pop())
%i              (for ANSI terminals)
                  add 1 to first parm, if one parm present,
                  or first two parms, if more than one parm present

%? expr %t thenpart %e elsepart %;
                if-then-else, %e elsepart is optional;
                else-if's are possible ala Algol 68:
                %? c1 %t b1 %e c2 %t b2 %e c3 %t b3 %e c4 %t b4 %e b5%;
                ci are conditions, bi are bodies.
```

If the "–" flag is used with "%[doxXs]", then a colon (:) must be placed between the "%" and the "–" to differentiate the flag from the binary "%–" operator, .e.g "%:–16.16s".

Consider the Hewlett-Packard 2645, which, to get to row 3 and column 12, needs to be sent **\E&a12c03Y** padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are zero-padded as two digits. Thus its **cup** capability is "**cup**=\E&a%p2%2.2dc%p1%2.2dY$<6>".

The Micro-Term ACT-IV needs the current row and column sent preceded by a **^T**, with the row and column simply encoded in binary, "**cup**=^T%p1%c%p2%c". Terminals which use "%c" need to be able to backspace the cursor (**cub1**), and to move the cursor up one line on the screen (**cuu1**). This is necessary because it is not always safe to transmit **\n**, **^D**, and **\r**, as the system may change or discard them. (The library routines dealing with *terminfo* set tty modes so that tabs are never expanded, so **\t** is safe to send. This turns out to be essential for the Ann Arbor 4080.)

A final example is the LSI ADM-3a, which uses row and column offset by a blank character, thus "**cup**=\E=%p1%'\s'%+%c%p2%'\s'%+%c". After sending "\E=", this pushes the first parameter, pushes the ASCII value for a space (32), adds them (pushing the sum on the stack in place of the two previous values), and outputs that value as a character. Then the same is done for the second parameter. More complex arithmetic is possible using the stack.

Cursor Motions

If the terminal has a fast way to home the cursor (to very upper left corner of screen) then this can be given as **home**; similarly a fast way of getting to the lower left-hand corner can be given as **ll**; this may involve going up with **cuu1** from the home position, but a program should never do this itself (unless **ll** does) because it can make no assumption about the effect of moving up from the home position. Note that the home position is the same as addressing to (0,0): to the top left corner of the screen, not of memory. (Thus, the \EH sequence on Hewlett-Packard terminals cannot be used for **home** without losing some of the other features on the terminal.)

If the terminal supports row or column absolute-cursor addressing, these can be given as single parameter capabilities **hpa** (horizontal position absolute) and **vpa** (vertical position absolute). Sometimes these are shorter than the more general two-parameter sequence (as with the Hewlett-Packard 2645) and can be used in preference to **cup**. If there are parameterized local motions (e.g., move *n* spaces to the right) these can be given as **cud**, **cub**, **cuf**, and **cuu** with a single parameter indicating how many spaces to move. These are primarily useful if the terminal does not support **cup**, such as the Tektronix 4025.

Area Clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as **el**. If the terminal can clear from the beginning of the line to the current position inclusive, leaving the cursor where it is, this should be given as **el1**. If the terminal can clear from the current position to the end of the display, then this should be given as **ed**. **ed** is only defined from the first column of a line. (Thus, it can be simulated by a request to delete a large number of lines, if a true **ed** is not available.)

Insert/delete line

If the terminal can open a new blank line before the line where the cursor is located, this should be given as **il1**; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line on which the cursor is located, then this should be given as **dl1**; this is done only from the first position on the line to be deleted. Versions of **il1** and **dl1** which take a single parameter and insert or delete that many lines can be given as **il** and **dl**.

If the terminal has a settable destructive scrolling region (like the VT100) the command to set this can be described with the **csr** capability, which takes two parameters: the top and bottom lines of the scrolling region. Unfortunately, the cursor position is undefined after using this command. It is possible to get the effect of insert or delete line using this command – the **sc** and **rc** (save and restore cursor) commands are also useful. Inserting lines at the top or bottom of the screen can also be accomplished using **ri** or **ind** on many terminals without a true insert/delete line, and is often faster even on terminals with those features.

To determine whether a terminal supports destructive scrolling regions or non-destructive scrolling regions, create a scrolling region in the middle of the screen, place data on the bottom line of the scrolling region, move the cursor to the top line of the scrolling region, and do a reverse index (**ri**) followed by a delete line (**dl1**) or index (**ind**). If the data that was originally on the bottom line of the scrolling region was restored into the scrolling region by the **dl1** or **ind**, then the terminal supports non-destructive scrolling regions. Otherwise, it has destructive scrolling regions. Do not specify **csr** if the terminal supports non-destructive scrolling regions, unless **ind**, **ri**, **indn**, **rin**, **dl**, and **dl1** all simulate destructive scrolling.

If the terminal provides the ability to define a window as part of memory, which all commands affect, it should be given as the parameterized string **wind**. The four parameters are the starting and ending lines in memory and the starting and ending columns in memory, in that order.

If the terminal can retain display memory above, then the **da** capability should be given; if display memory can be retained below, then **db** should be given. These indicate that deleting a line or scrolling a full screen may bring non-blank lines up from below or that scrolling back with **ri** may bring down non-blank lines.

Insert/Delete Character

There are two basic kinds of intelligent terminals with respect to insert/delete character operations which can be described using *terminfo.* The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the *Concept* 100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated, or expanded to two untyped blanks. You can determine the kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type "**abc def**" using local cursor motions (not spaces) between the **abc** and the **def**. Then position the cursor before the **abc** and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the **abc** shifts over to the **def** which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability **in**, which stands for "insert null". While these are two logically separate attributes (one line versus multiline insert mode, and special treatment of untyped spaces) there are currently no terminals whose insert mode cannot be described with the single attribute.

terminfo can describe both terminals which support an insert mode and terminals which send a simple sequence to open a blank position on the current line. Give as **smir** the sequence to get into insert mode. Give as **rmir** the sequence to leave insert mode. Now give as **ich1** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode do not give **ich1**; terminals which send a sequence to open a screen position should give it here. (If your terminal supports both, insert mode is usually preferable to **ich1**. Do not give both unless the terminal actually requires both to be used in combination.) If post-insert padding is needed, give this as a number of milliseconds padding in **ip** (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in **ip**. If your terminal needs both to be placed into an 'insert mode' and a special code to precede each inserted character, then both **smir/rmir** and **ich1** can be given, and both will be used. The **ich** capability, with one parameter, *n*, repeats the effects of **ich1** *n* times.

If padding is necessary between characters typed while not in insert mode, give this as a number of milliseconds padding in **rmp**.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (e.g., if there is a tab after the insertion position). If your terminal allows motion while in insert mode you can give the capability **mir** to speed up inserting in this case. Omitting **mir** affects only speed. Some terminals (notably Datamedia's) must not have **mir** because of the way their insert mode works.

Finally, you can specify **dch1** to delete a single character, **dch** with one parameter, *n*, to delete *n* characters, and delete mode by giving **smdc** and **rmdc** to enter and exit delete mode (any mode the terminal needs to be placed in for **dch1** to work).

A command to erase *n* characters (equivalent to outputting *n* blanks without moving the cursor) can be given as **ech** with one parameter.

Highlighting, Underlining, and Visible Bells

If your terminal provides one or more kinds of display attributes, these can be represented in a number of different ways. You should choose one display form as *standout mode* (see *curses*(3X)), representing a good, high contrast, easy-on-the-eyes, format for highlighting error messages and other attention getters. (If you have a choice, reverse-video plus half-bright is good, or reverse-video alone; however, different users have different preferences on different terminals.) The sequences to enter and exit standout mode are given as **smso** and **rmso**, respectively. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TVI 912 and Teleray 1061 do, then **xmc** should be given to tell how many spaces are left.

Codes to begin underlining and end underlining can be given as **smul** and **rmul** respectively. If the terminal has a code to underline the current character and move the cursor one space to the right, such as the Micro-Term MIME, this can be given as **uc**.

Other capabilities to enter various highlighting modes include **blink** (blinking), **bold** (bold or extra-bright), **dim** (dim or half-bright), **invis** (blanking or invisible text), **prot** (protected), **rev** (reverse-video), **sgr0** (turn off all attribute modes), **smacs** (enter alternate-character-set mode), and **rmacs** (exit alternate-character-set mode). Turning on any of these modes singly may or may not turn off other modes. If a command is necessary before alternate character set mode is entered, give the sequence in **enacs** (enable alternate-character-set mode).

If there is a sequence to set arbitrary combinations of modes, this should be given as **sgr** (set attributes), taking nine parameters. Each parameter is either **0** or non-zero, as the corresponding attribute is on or off. The nine parameters are, in order: standout, underline, reverse, blink, dim, bold, blank, protect, alternate character set. Not all modes need be supported by **sgr**, only those for which corresponding separate attribute commands exist. (See the example at the end of this section.)

Terminals with the "magic cookie" glitch (**xmc**) deposit special "cookies" when they receive mode-setting sequences, which affect the display algorithm rather than having extra bits for each character. Some terminals, such as the Hewlett-Packard 2621, automatically leave standout mode when they move to a new line or the cursor is addressed. Programs using standout mode should exit standout mode before moving the cursor or sending a newline, unless the **msgr** capability is present, asserting that it is safe to move in standout mode.

If the terminal can flash the screen to indicate an error quietly (a bell replacement), then this can be given as **flash**; it must not move the cursor. A good flash can be done by changing the screen into reverse video, pad for 200 ms, then return the screen to normal video.

If the cursor needs to be made more visible than normal when it is not on the bottom line (to make, for example, a non-blinking underline into an easier to find block or blinking underline) give this sequence as **cvvis**. The boolean **chts** should also be given. If there is a way to make the cursor completely invisible, give that as **civis**. The capability **cnorm** should be given which undoes the effects of either of these modes.

If the terminal needs to be in a special mode when running a program that uses these capabilities, the codes to enter and exit this mode can be given as **smcup** and **rmcup**. This arises, for example, from terminals like the *Concept* with more than one page of memory. If the terminal supports only memory relative cursor addressing and not screen relative cursor addressing, a one screen-sized window must be fixed into the terminal for cursor addressing to work properly. This is also used for the Tektronix

4025, where **smcup** sets the command character to be the one used by **terminfo**. If the **smcup** sequence does not restore the screen after an **rmcup** sequence is output (to the state prior to outputting **rmcup**), specify **nrrmc**.

If your terminal generates underlined characters by using the underline character (with no special codes needed) even though it does not otherwise overstrike characters, then you should give the capability **ul**. For terminals where a character overstriking another leaves both characters on the screen, give the capability **os**. If overstrikes are erasable with a blank, then this should be indicated by giving **eo**.

Example of highlighting: assume that the terminal under question needs the following escape sequences to turn on various modes.

tparm parameter	attribute	escape sequence
	none	\E[0m
p1	standout	\E[0;4;7m
p2	underline	\E[0;3m
p3	reverse	\E[0;4m
p4	blink	\E[0;5m
p5	dim	\E[0;7m
p6	bold	\E[0;3;4m
p7	invis	\E[0;8m
p8	protect	not available
p9	altcharset	^O (off) ^N(on)

Note that each escape sequence requires a **0** to turn off other modes before turning on its own mode. Also note that, as suggested above, *standout* is set up to be the combination of *reverse* and *dim*. Also, since this terminal provides no *bold* mode, *bold* is set up as the combination of *reverse* and *underline*. In addition, to allow combinations, such as *underline+blink*, the sequence to use would be **\E[0;3;5m**. The terminal doesn't provide *protect* mode, either, but that cannot be simulated in any way, so **p8** is ignored. The *altcharset* mode is different in that it is either **^O** or **^N** depending on whether it is off or on. If all modes were to be turned on, the sequence would be **\E[0;3;4;5;7;8m^N**.

Now look at what happens when different sequences are output. For example, **;3** is output when either **p2** or **p6** is true, that is, if either *underline* or *bold* modes are turned on. Writing out the above sequences, along with their dependencies, gives the following:

sequence	when to output	terminfo translation
\E[0	always	\E[0
;3	if p2 or p6	%?%p2%p6%|%t;3%;
;4	if p1 or p3 or p6	%?%p1%p3%|%p6%|%t;4%;
;5	if p4	%?%p4%t;5%;
;7	if p1 or p5	%?%p1%p5%|%t;7%;
;8	if p7	%?%p7%t;8%;
m	always	m
^N or ^O	if p9 ^N, else ^O	%?%p9%t^N%e^O%;

Putting this all together into the **sgr** sequence gives:

```
sgr=\E[0%?%p2%p6%|%t;3%;%?%p1%p3%|%p6%|%t;4%;%?%p5%t;5%;%?%p1%p5%
          |%t;7%;%?%p7%t;8%;m%?%p9%t^N%e^O%;,
```

Keypad

If the terminal keypad transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local (this applies, for example, to the unshifted Hewlett-Packard 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as **smkx** and **rmkx**. Otherwise the keypad is assumed always to transmit.

The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as **kcub1, kcuf1, kcuu1, kcud1,** and **khome** respectively. If there are function keys such as f0, f1, ..., f63, the codes they send can be given as **kf0, kf1, ..., kf63.** If the first 11 keys use labels other than the default f0 through f10, the labels can be given as **lf0, lf1, ..., lf10.** The codes transmitted by certain other special keys can be given: **kll** (home down), **kbs** (backspace), **ktbc** (clear all tabs), **kctab** (clear the tab stop in this column), **kclr** (clear screen or erase key), **kdch1** (delete character), **kdl1** (delete line), **krmir** (exit insert mode), **kel** (clear to end of line), **ked** (clear to end of screen), **kich1** (insert character or enter insert mode), **kil1** (insert line), **knp** (next page), **kpp** (previous page), **kind** (scroll forward/down), **kri** (scroll backward/up), **khts** (set a tab stop in this column). In addition, if the keypad contains a 3 by 3 array of keys including the four arrow keys, the other five keys can be given as **ka1, ka3, kb2, kc1,** and **kc3.** These keys are useful when the effects of a 3 by 3 directional pad are needed. Further keys are defined above in the capabilities list.

Strings to program function keys can be given as **pfkey, pfloc,** and **pfx.** A string to program their soft-screen labels can be given as **pln.** Each of these strings takes two parameters: the function key number to program (from 0 to 10) and the string to program it with. Function key numbers out of this range may program undefined keys in a terminal-dependent manner. The difference between the capabilities is that **pfkey** causes pressing the given key to be the same as the user typing the given string; **pfloc** causes the string to be executed by the terminal in local mode; and **pfx** causes the string to be transmitted to the computer. The capabilities **nlab, lw** and **lh** define how many soft labels there are and their width and height. If there are commands to turn the labels on and off, give them in **smln** and **rmln. smln** is normally output after one or more **pln** sequences to make sure that the change becomes visible.

Tabs and Initialization

If the terminal provides hardware tabs, the command to advance to the next tab stop can be given as **ht** (usually control-I). A "backtab" command which moves leftward to the next tab stop can be given as **cbt.** By convention, if the teletype modes indicate that tabs are being expanded by the computer rather than being sent to the terminal, programs should not use **ht** or **cbt** even if they are present, since the user may not have set the tab stops properly. If the terminal provides hardware tabs which are initially set every *n* spaces when the terminal is powered up, the numeric parameter **it** is given, showing the number of spaces the tabs are set to. This is normally used by **tput init** (see *tput*(1)) to determine whether to set the mode for hardware tab expansion and whether to set the tab stops. If the terminal supports tab stops that can be saved in nonvolatile memory, the *terminfo* description can assume that they are properly set. If there are commands to set and clear tab stops, they can be given as **tbc** (clear all tab stops) and **hts** (set a tab stop in the current column of every row).

Other capabilities include: **is1, is2,** and **is3,** initialization strings for the terminal; **iprog,** the path name of a program to be run to initialize the terminal; and **if,** the name of a file containing long initialization strings. These strings are expected to set the terminal into modes consistent with the rest of the *terminfo* description. They must be sent to the terminal each time the user logs in and be output in the following order: run the program **iprog;** output **is1;** output **is2;** set the margins using **mgc,**

smgl and **smgr**; set the tabs using **tbc** and **hts**; print the file **if**; and finally output **is3**. This is usually done using the **init** option of *tput*(1); see *profile*(4).

Most initialization is done with **is2**. Special terminal modes can be set up without duplicating strings by putting the common sequences in **is2** and special cases in **is1** and **is3**. Sequences that do a harder reset from a totally unknown state can be given as **rs1**, **rs2**, **rf**, and **rs3**, analogous to **is1**, **is2**, **is3**, and **if**. (The method using files, **if** and **rf**, is used for a few terminals, from */usr/lib/tabset/**; however, the recommended method is to use the initialization and reset strings.) These strings are output by **tput reset**, which is used when the terminal gets into a wedged state. Commands are normally placed in **rs1**, **rs2**, **rs3**, and **rf** only if they produce annoying effects on the screen and are not necessary when logging in. For example, the command to set a terminal into 80-column mode would normally be part of **is2**, but on some terminals it causes an annoying glitch on the screen and is not normally needed since the terminal is usually already in 80-column mode.

If a more complex sequence is needed to set the tabs than can be described by using **tbc** and **hts**, the sequence can be placed in **is2** or **if**.

If there are commands to set and clear margins, they can be given as **mgc** (clear all margins), **smgl** (set left margin), and **smgr** (set right margin).

Delays

Certain capabilities control padding in the *tty*(7) driver. These are primarily needed by hard-copy terminals, and are used by **tput init** to set tty modes appropriately. Delays embedded in the capabilities **cr**, **ind**, **cub1**, **ff**, and **tab** can be used to set the appropriate delay bits to be set in the tty driver. If **pb** (padding baud rate) is given, these values can be ignored at baud rates below the value of **pb**.

Status Lines

If the terminal provides an extra "status line" that is not normally used by software, this fact can be indicated. If the status line is viewed as an extra line below the bottom line, into which one can cursor address normally (such as the Heathkit h19's 25th line, or the 24th line of a VT100 which is set to a 23-line scrolling region), the capability **hs** should be given. Special strings that go to a given column of the status line and return from the status line can be given as **tsl** and **fsl**. (**fsl** must leave the cursor position in the same place it was before **tsl**. If necessary, the **sc** and **rc** strings can be included in **tsl** and **fsl** to get this effect.) The capability **tsl** takes one parameter, which is the column number of the status line the cursor is to be moved to.

If escape sequences and other special commands, such as tab, work while in the status line, the flag **eslok** can be given. A string which turns off the status line (or otherwise erases its contents) should be given as **dsl**. If the terminal supports commands to save and restore the position of the cursor, give them as **sc** and **rc**. The status line is normally assumed to be the same width as the rest of the screen, e.g., **cols**. If the status line is a different width (possibly because the terminal does not allow an entire line to be loaded) the width, in columns, can be indicated with the numeric parameter **wsl**.

Line Graphics

If the terminal supports a line drawing alternate character set, the mapping of glyph to character would be given in **acsc**. The definition of this string is based on the alternate character set used in the DEC VT100 terminal, extended slightly with some characters from the AT&T 4410v1 terminal.

glyph name	vt100+ character

arrow pointing right	+
arrow pointing left	,
arrow pointing down	.
solid square block	0
lantern symbol	I
arrow pointing up	–
diamond	`
checker board (stipple)	a
degree symbol	f
plus/minus	g
board of squares	h
lower right corner	j
upper right corner	k
upper left corner	l
lower left corner	m
plus	n
scan line 1	o
horizontal line	q
scan line 9	s
left tee (├)	t
right tee (┤)	u
bottom tee (┴)	v
top tee (┬)	w
vertical line	x
bullet	~

The best way to describe a new terminal's line graphics set is to add a third column to the above table with the characters for the new terminal that produce the appropriate glyph when the terminal is in the alternate character set mode. For example,

glyph name	vt100+ char	new tty char
upper left corner	l	R
lower left corner	m	F
upper right corner	k	T
lower right corner	j	G
horizontal line	q	,
vertical line	x	.

Now write down the characters left to right, as in "**acsc**=lRmFkTjGq\,x.".

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as **pad**. Only the first character of the **pad** string is used. If the terminal does not provide a pad character, specify **npc**.

If the terminal can move up or down half a line, this can be indicated with **hu** (half-line up) and **hd** (half-line down). This is primarily useful for superscripts and subscripts on hardcopy terminals. If a hardcopy terminal can eject to the next page (form feed), give this as **ff** (usually control-L).

If there is a command to repeat a given character a given number of times (to save time transmitting a large number of identical characters) this can be indicated with the parameterized string **rep**. The first parameter is the character to be repeated and the second is the number of times to repeat it. Thus, **tparm(repeat_char, 'x', 10)** is the same as **xxxxxxxxxx**.

If the terminal provides a settable command character, such as the Tektronix 4025, this can be indicated with **cmdch**. A prototype command character is chosen which is used in all capabilities. This character is given in the **cmdch** capability to identify it. The following convention is supported on some UNIX systems: If the environment variable **CC** exists, all occurrences of the prototype character are replaced with the character in **CC**.

Terminal descriptions that do not represent a specific kind of known terminal, such as **switch**, **dialup**, **patch**, and **network**, should include the **gn** (generic) capability so that programs can complain that they do not know how to talk to the terminal. (This capability does not apply to **virtual** terminal descriptions for which the escape sequences are known.) If the terminal is one of those supported by the UNIX system virtual terminal protocol, the terminal number can be given as **vt**. A line-turn-around sequence to be transmitted before doing reads should be specified in **rfi**.

If the terminal uses xon/xoff handshaking for flow control, give **xon**. Padding information should still be included so that routines can make better decisions about costs, but actual pad characters are not transmitted. Sequences to turn on and off xon/xoff handshaking may be given in **smxon** and **rmxon**. If the characters used for handshaking are not ^S and ^Q, they may be specified with **xonc** and **xoffc**.

If the terminal has a "meta key" which acts as a shift key, setting the 8th bit of any character transmitted, this fact can be indicated with **km**. Otherwise, software assumes that the 8th bit is parity and it is usually cleared. If strings exist to turn this "meta mode" on and off, they can be given as **smm** and **rmm**.

If the terminal provides more lines of memory than fit on the screen at once, the number of lines of memory can be indicated with **lm**. A value of **lm**#0 indicates that the number of lines is not fixed, but that there is still more memory than fits on the screen.

Media copy strings that control an auxiliary printer connected to the terminal can be given as **mc0**: print the contents of the screen, **mc4**: turn off the printer, and **mc5**: turn on the printer. When the printer is on, all text sent to the terminal is sent to the printer. A variation, **mc5p**, takes one parameter, and leaves the printer on for as many characters as the value of the parameter, then turns the printer off. The parameter should not exceed 255. If the text is not displayed on the terminal screen when the printer is on, specify **mc5i** (silent printer). All text, including **mc4**, is transparently passed to the printer while an **mc5p** is in effect.

Special Cases

The working model used by *terminfo* fits most terminals reasonably well. However, some terminals do not completely match that model, requiring special support by *terminfo*. These are not meant to be construed as deficiencies in the terminals; they are just differences between the working model and the actual hardware. They may be unusual devices or, for some reason, do not provide all the features of the *terminfo* model implemented.

Terminals which can not display tilde (~) characters, such as certain Hazeltine terminals, should indicate **hz**.

Terminals which ignore a linefeed immediately after an **am** wrap, such as the *Concept* 100, should indicate **xenl**. Those terminals on which the cursor remains on the right-most column until another character has been received, rather than wrapping immediately upon receiving the right-most character, such as the VT100, should also indicate **xenl**.

If **el** is required to get rid of standout (instead of writing normal text on top of it), **xhp** should be given.

Those Teleray terminals on which tabs turn all characters moved over to blanks, should indicate **xt** (destructive tabs). This capability is also taken to mean that it is not possible to position the cursor on top of a "magic cookie" therefore, to erase standout mode, it is instead necessary to use delete and insert line.

Those Beehive Superbee terminals which do not transmit the escape or control–C characters, should specify **xsb**, indicating that the f1 key is to be used for escape and the f2 key for control–C.

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **use** can be given with the name of the similar terminal. The capabilities given before **use** override those in the terminal type invoked by **use**. A capability can be canceled by placing *xx@* to the left of the capability definition, where *xx* is the capability. For example, the entry

```
att4424-2|Teletype 4424 in display function group ii,
    rev@, sgr@, smul@, use=att4424,
```

defines an AT&T 4424 terminal that does not support the **rev**, **sgr**, and **smul** capabilities, and hence cannot do highlighting. This is useful for different modes for a terminal, or for different user preferences. More than one **use** capability may be given.

FILES

/usr/lib/terminfo/?/*	compiled terminal description database
/usr/lib/.COREterm/?/*	subset of compiled terminal description database
/usr/lib/tabset/*	tab settings for some terminals, in a format appropriate to be output to the terminal (escape sequences that set margins and tabs)

SEE ALSO

captoinfo(1M), curses(3X), infocmp(1M), printf(3S), term(5), tic(1M), tput(1), tty(7)

WARNING

As described in the "Tabs and Initialization" section above, a terminal's initialization strings, **is1, is2,** and **is3**, if defined, must be output before a *curses*(3X) program is run. An available mechanism for outputting such strings is **tput init** (see *tput*(1) and *profile*(4)).

Tampering with entries in */usr/lib/.COREterm/?/** or */usr/lib/terminfo/?/** (for example, changing or removing an entry) can affect programs such as *vi*(1) that expect the entry to be present and correct. In particular, removing the description for the "dumb" terminal causes unexpected problems.

NOTE

The *termcap* database (from earlier releases of UNIX System V) may not be supplied in future releases.

NAME

timezone – set default system time zone

SYNOPSIS

/etc/TIMEZONE

DESCRIPTION

This file sets and exports the time zone environmental variable **TZ**.

This file is "dotted" into other files that must know the time zone.

EXAMPLES

/etc/TIMEZONE for the east coast:

```
#       Time Zone
TZ=EST5EDT
export TZ
```

SEE ALSO

ctime(3C), profile(4), rc2(1M)

NAME

unistd – file header for symbolic constants

SYNOPSIS

#include <unistd.h>

DESCRIPTION

The header file *<unistd.h>* lists the symbolic constants and structures not already defined or declared in some other header file.

```
/* Symbolic constants for the "access" routine: */

#define R_OK        4       /*Test for Read permission */
#define W_OK        2       /*Test for Write permission */
#define X_OK        1       /*Test for eXecute permission */
#define F_OK        0       /*Test for existence of File */

#define F_ULOCK     0       /*Unlock a previously locked region */
#define F_LOCK      1       /*Lock a region for exclusive use */
#define F_TLOCK     2       /*Test and lock a region for exclusive use */
#define F_TEST      3       /*Test a region for other processes locks */

/*Symbolic constants for the "lseek" routine: */

#define SEEK_SET    0       /* Set file pointer to "offset" */
#define SEEK_CUR    1       /* Set file pointer to current plus "offset" */
#define SEEK_END    2       /* Set file pointer to EOF plus "offset" */

/*Pathnames:*/

#define GF_PATH  /etc/group/*Pathname of the group file */
#define PF_PATH  /etc/passwd/*Pathname of the passwd file */
```

NAME

utmp, wtmp – utmp and wtmp entry formats

SYNOPSIS

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

DESCRIPTION

These files, which hold user and accounting information for such commands as *who*(1), *write*(1), and *login*(1), use the following structure as defined by **<utmp.h>**:

```
#define UTMP_FILE  "/etc/utmp"
#define WTMP_FILE  "/etc/wtmp"
#define ut_name    ut_user

struct utmp {
        char   ut_user[8];          /* User login name */
        char   ut_id[4];            /* /etc/inittab id (usually line #) */
        char   ut_line[12];         /* device name (console, lnxx) */
        char   ut_host[16];         /* host name, if remote login */
        short  ut_pid;              /* process id */
        short  ut_type;             /* type of entry */
        struct exit_status {
         short  e_termination;      /* Process termination status */
         short  e_exit;             /* Process exit status */
        } ut_exit;                  /* The exit status of a process
                                       marked as DEAD_PROCESS. */
        time_t ut_time;             /* time entry was made */
};

/* Definitions for ut_type */
#define EMPTY          0
#define RUN_LVL        1
#define BOOT_TIME      2
#define OLD_TIME       3
#define NEW_TIME       4
#define INIT_PROCESS   5            /* Process spawned by "init" */
#define LOGIN_PROCESS  6            /* A "getty" process waiting for login */
#define USER_PROCESS   7            /* A user process */
#define DEAD_PROCESS   8
#define ACCOUNTING     9
#define UTMAXTYPE      ACCOUNTING   /* Largest legal value of ut_type */
#define RUNLVL_MSG     "run-level %c"
#define BOOT_MSG       "system boot"
#define OTIME_MSG      "old time"
#define NTIME_MSG      "new time"
```

FILES

/etc/utmp
/etc/wtmp

SEE ALSO

getut(3C), login(1), who(1), write(1)

NAME

uuencode – format of an encoded uuencode file

DESCRIPTION

Files output by *uuencode(1C)* consist of a header line, followed by a number of body lines, and a trailer line. *Uudecode(1C)* will ignore any lines preceding the header or following the trailer. Lines preceding a header must not, of course, look like a header.

The header line is distinguished by having the first 6 characters "begin ". The word *begin* is followed by a mode (in octal), and a string which names the remote file. A space separates the three items in the header line.

The body consists of a number of lines, each at most 62 characters long (including the trailing newline). These consist of a character count, followed by encoded characters, followed by a newline. The character count is a single printing character, and represents an integer, the number of bytes the rest of the line represents. Such integers are always in the range from 0 to 63 and can be determined by subtracting the character space (octal 40) from the character.

Groups of 3 bytes are stored in 4 characters, 6 bits per character. All are offset by a space to make the characters printing. The last line may be shorter than the normal 45 bytes. If the size is not a multiple of 3, this fact can be determined by the value of the count on the last line. Extra garbage will be included to make the character count a multiple of 4. The body is terminated by a line with a count of zero. This line consists of one ASCII space.

The trailer line consists of "end" on a line by itself.

SEE ALSO

uuencode(1C), uudecode(1C), uusend(1C), uucp(1C), mail(1)

NAME

intro – introduction to miscellany

DESCRIPTION

This section describes miscellaneous facilities such as macro packages, character set tables, etc.

NAME

ascii – map of ASCII character set

DESCRIPTION

ascii is a map of the ASCII character set, giving both octal and hexadecimal equivalents of each character:

```
|000 nul |001 soh |002 stx |003 etx |004 eot |005 enq |006 ack |007 bel |
|010 bs  |011 ht  |012 nl  |013 vt  |014 np  |015 cr  |016 so  |017 si  |
|020 dle |021 dc1 |022 dc2 |023 dc3 |024 dc4 |025 nak |026 syn |027 etb |
|030 can |031 em  |032 sub |033 esc |034 fs  |035 gs  |036 rs  |037 us  |
|040 sp  |041 !   |042 "   |043 #   |044 $   |045 %   |046 &   |047 '   |
|050 (   |051 )   |052 *   |053 +   |054 ,   |055 -   |056 .   |057 /   |
|060 0   |061 1   |062 2   |063 3   |064 4   |065 5   |066 6   |067 7   |
|070 8   |071 9   |072 :   |073 ;   |074 <   |075 =   |076 >   |077 ?   |
|100 @   |101 A   |102 B   |103 C   |104 D   |105 E   |106 F   |107 G   |
|110 H   |111 I   |112 J   |113 K   |114 L   |115 M   |116 N   |117 O   |
|120 P   |121 Q   |122 R   |123 S   |124 T   |125 U   |126 V   |127 W   |
|130 X   |131 Y   |132 Z   |133 [   |134 \   |135 ]   |136 ^   |137 _   |
|140 `   |141 a   |142 b   |143 c   |144 d   |145 e   |146 f   |147 g   |
|150 h   |151 i   |152 j   |153 k   |154 l   |155 m   |156 n   |157 o   |
|160 p   |161 q   |162 r   |163 s   |164 t   |165 u   |166 v   |167 w   |
|170 x   |171 y   |172 z   |173 {   |174 |   |175 }   |176 ~   |177 del |

| 00 nul | 01 soh | 02 stx | 03 etx | 04 eot | 05 enq | 06 ack | 07 bel |
| 08 bs  | 09 ht  | 0a nl  | 0b vt  | 0c np  | 0d cr  | 0e so  | 0f si  |
| 10 dle | 11 dc1 | 12 dc2 | 13 dc3 | 14 dc4 | 15 nak | 16 syn | 17 etb |
| 18 can | 19 em  | 1a sub | 1b esc | 1c fs  | 1d gs  | 1e rs  | 1f us  |
| 20 sp  | 21 !   | 22 "   | 23 #   | 24 $   | 25 %   | 26 &   | 27 '   |
| 28 (   | 29 )   | 2a *   | 2b +   | 2c ,   | 2d -   | 2e .   | 2f /   |
| 30 0   | 31 1   | 32 2   | 33 3   | 34 4   | 35 5   | 36 6   | 37 7   |
| 38 8   | 39 9   | 3a :   | 3b ;   | 3c <   | 3d =   | 3e >   | 3f ?   |
| 40 @   | 41 A   | 42 B   | 43 C   | 44 D   | 45 E   | 46 F   | 47 G   |
| 48 H   | 49 I   | 4a J   | 4b K   | 4c L   | 4d M   | 4e N   | 4f O   |
| 50 P   | 51 Q   | 52 R   | 53 S   | 54 T   | 55 U   | 56 V   | 57 W   |
| 58 X   | 59 Y   | 5a Z   | 5b [   | 5c \   | 5d ]   | 5e ^   | 5f _   |
| 60     | 61 a   | 62 b   | 63 c   | 64 d   | 65 e   | 66 f   | 67 g   |
| 68 h   | 69 i   | 6a j   | 6b k   | 6c l   | 6d m   | 6e n   | 6f o   |
| 70 p   | 71 q   | 72 r   | 73 s   | 74 t   | 75 u   | 76 v   | 77 w   |
| 78 x   | 79 y   | 7a z   | 7b {   | 7c |   | 7d }   | 7e ~   | 7f del |
```

NAME

environ – user environment

DESCRIPTION

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

PATH
: The sequence of directory prefixes that *sh*(1), *time*(1), *nice*(1), *nohup*(1), etc., apply in searching for a file known by an incomplete path name. The prefixes are separated by colons (**:**). *Login*(1) sets **PATH=:/bin:/usr/bin.**

HOME
: Name of the user's login directory, set by *login*(1) from the password file *passwd*(4).

TERM
: The kind of terminal for which output is to be prepared. This information is used by commands, such as *mm*(1) or *tplot*(1G), which may exploit special capabilities of that terminal.

TZ
: Time zone information. The format is **xxx***n***zzz** where **xxx** is standard local time zone abbreviation, *n* is the difference in hours from GMT, and **zzz** is the abbreviation for the daylight-saving local time zone, if any; for example, **EST5EDT.**

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

SEE ALSO

env(1), exec(2), login(1), mm(1), nice(1), nohup(1), sh(1), time(1), tplot(1G)

NAME

fcntl – file control options

SYNOPSIS

#include <fcntl.h>

DESCRIPTION

The *fcntl*(2) function provides for control over open files. This include file describes *requests* and *arguments* to *fcntl* and *open*(2).

```
/* Flag values accessible to open(2) and fcntl(2) */
/*  (The first three can only be set by open) */
#define O_RDONLY  0
#define O_WRONLY  1
#define O_RDWR    2
#define O_NDELAY  04     /* Non-blocking I/O */
#define O_APPEND  010    /* append (writes guaranteed at the end) */
#define O_SYNC    020    /* synchronous write option */

/* Flag values accessible only to open(2) */
#define O_CREAT   00400  /* open with file create (uses third open arg)*/
#define O_TRUNC   01000  /* open with truncation */
#define O_EXCL    02000  /* exclusive open */

/* fcntl(2) requests */
#define F_DUPFD   0      /* Duplicate fildes */
#define F_GETFD   1      /* Get fildes flags */
#define F_SETFD   2      /* Set fildes flags */
#define F_GETFL   3      /* Get file flags */
#define F_SETFL   4      /* Set file flags */
#define F_GETLK   5      /* Get file lock */
#define F_SETLK   6      /* Set file lock */
#define F_SETLKW  7      /* Set file lock and wait */
#define F_CHKFL   8      /* Check legality of file flag changes */

/* file segment locking control structure */
struct flock {
        short   l_type;
        short   l_whence;
        long    l_start;
        long    l_len;       /* if 0 then until EOF */
        short   l_sysid;     /* returned with F_GETLK*/
        short   l_pid;       /* returned with F_GETLK*/
}

/* file segment locking types */
#define F_RDLCK   01     /* Read lock */
#define F_WRLCK   02     /* Write lock */
#define F_UNLCK   03     /* Remove locks */
```

SEE ALSO

fcntl(2), open(2)

SYNOPSIS

/usr/lib/getNAME
/usr/lib/makeindex
/usr/lib/makewhatis

NAME

man – format of man pages

DESCRIPTION

Preformatted versions of the manual pages are kept on-line in /usr/man/cat? and /usr/man/bsd/cat? directories. Each page is preformatted by *nroff* using the *–man* option. After preformatting, the pages are compressed using the program *compress* so that the man pages occupy as little disk space as possible.

In each of the */usr/man/cat* and */usr/man/bsd/cat* directories there is an *Index* file. This index file contains a list of all the names various commands and functions are known by, as well as the name of the file containing the preformatted man page. For instance, the entries *sin, cos, tan,* and other trigonometric functions are described by the man page *trig.3m*.

There is one other file involved with manual page related commands, the whatis data base file. This data base cross references the entry names against a brief one line description. It is used by the *whatis*(1) and *apropos*(1) commands to search for manual pages by keyword and by name.

The utility *getNAME* reads a set of unformatted manual pages and outputs lines which combine the '.TH' line with the name(s) of the entry, followed by the one line description. The *makewhatis* shell script massages this information, creating the whatis data base. The *makeindex* command takes a list of manual page files and builds the *Index* file.

FILES

/usr/man/cat?	formatted manual pages
/usr/man/cat?/Index	index of entries and filenames
/usr/man/bsd/cat?	formatted manual pages for bsd manual pages
/usr/man/whatis	Data base

SEE ALSO

apropos(1), man(1), whatis(1), catman(8)

NAME

math – math functions and constants

SYNOPSIS

#include <math.h>

DESCRIPTION

This file contains declarations of all the functions in the Math Library (described in Section 3M), as well as various functions in the C Library (Section 3C) that return floating-point values.

It defines the structure and constants used by the error-handling mechanisms, including the following constant:

HUGE	The maximum value of a single-precision floating-point number.

The following mathematical constants are defined for user convenience:

M_E	The base of natural logarithms (*e*).
M_LOG2E	The base-2 logarithm of *e*.
M_LOG10E	The base-10 logarithm of *e*.
M_LN2	The natural logarithm of 2.
M_LN10	The natural logarithm of 10.
M_PI	π, the ratio of the circumference of a circle to its diameter.
M_PI_2	π/2.
M_PI_4	π/4.
M_1_PI	1/π.
M_2_PI	2/π.
M_2_SQRTPI	2/√π.
M_SQRT2	The positive square root of 2.
M_SQRT1_2	The positive square root of 1/2.

For the definitions of various machine-dependent "constants," see the description of the <*values.h*> header file.

SEE ALSO

intro(3), values(5), vmath(5)

NAME

mtio – UNIX magtape manipulation interface

SYNOPSIS

```
#include <sys/types.h>
#include <sys/43ioctl.h>
#include <sys/mtio.h>
```

DESCRIPTION

The following information is from <*sys/mtio.h*>:

```
/*
 * Structures and definitions for mag tape io control commands
 */

/* structure for MTIOCTOP - mag tape op command */
struct  mtop  {
        short   mt_op;          /* operations defined below */
        daddr_tmt_count;        /* how many of them */
};

/* operations */
#define MTWEOF      0       /* write an end-of-file record */
#define MTFSF 1     /* forward space file */
#define MTBSF 2     /* backward space file */
#define MTFSR 3     /* forward space record */
#define MTBSR 4     /* backward space record */
#define MTREW       5       /* rewind */
#define MTOFFL      6       /* rewind and put the drive offline */
#define MTNOP       7       /* no operation, sets status only */
#define MTRETEN 8   /* retension the tape */
#define MTERASE 9   /* erase the entire tape */

/* structure for MTIOCGET - mag tape get status command */

struct  mtget  {
        short   mt_type;        /* type of magtape device */
/* the following two registers are grossly device dependent */
        short   mt_dsreg;       /* "drive status" register */
        short   mt_erreg;       /* "error" register */
/* end device-dependent registers */
        short   mt_resid;       /* residual count */
/* the following two are not yet implemented */
        daddr_tmt_fileno;       /* file number of current position */
        daddr_tmt_blkno;        /* block number of current position */
/* end not yet implemented */
};

/*
 * Constants for mt_type byte.  These are the same
 * for other controllers compatible with the types listed.
 */
#define MT_ISTS         0x01        /* TS-11 */
#define MT_ISHT         0x02        /* TM03 Massbus: TE16, TU45, TU77 */
#define MT_ISTM         0x03        /* TM11/TE10 Unibus */
#define MT_ISMT         0x04        /* TM78/TU78 Massbus */
```

```
#define MT_ISUT         0x05        /* SI TU-45 emulation on Unibus */
#define MT_ISCPC        0x06        /* SUN */
#define MT_ISAR         0x07        /* SUN */
#define MT_ISTMSCP      0x08        /* DEC TMSCP protocol (TU81, TK50) */
#define MT_ISSCSI       0x0A        /* titan: Generic SCSI */


/* mag tape io control commands */
#define MTIOCTOP        _IOW(m, 1, struct mtop)         /* do a mag tape op */
#define MTIOCGET        _IOR(m, 2, struct mtget) /* get tape status */

#ifndef KERNEL
#define DEFTAPE         "/dev/rmt8"
#endif
```

FILES

/dev/{r}mt/c?d?[lmh]{n}
/dev/{r}mt/0m{n}
/dev/{r}mt[0-9]*

SEE ALSO

mt(1), tar(1)

BUGS

The status should be returned in a device independent format.

NAME

mv – a troff macro package for typesetting viewgraphs and slides

SYNOPSIS

mvt [**–a**] [*options*] [*files*] **troff** [**–a**] **–mv** [*options*] [*files*]

DESCRIPTION

This package makes it easy to typeset viewgraphs and projection slides in a variety of sizes. A few macros (briefly described below) accomplish most of the formatting tasks needed in making transparencies. All of the facilities of **troff**(1), **eqn**(1), **tbl**(1), **pic**(1), and **grap**(1) are available for more difficult tasks.

To preview output, specify the -a option.

The available macros are:

.VS [*n*] [*i*] [*d*]
: Foil-start macro; foil size is to be 7″x7″; *n* is the foil number, *i* is the foil identification, *d* is the date; the foil-start macro resets all parameters (indent, point size, etc.) to initial default values, except for the values of *i* and *d* arguments inherited from a previous foil-start macro; it also invokes the **.A** macro (see below).

  The naming convention for this and the following eight macros is that the first character of the name (**V** or **S**) distinguishes between viewgraphs and slides, respectively, while the second character indicates whether the foil is square (**S**), small wide (**w**), small high (**h**), big wide (**W**), or big high (**H**). Slides are "skinnier" than the corresponding viewgraphs: the ratio of the longer dimension to the shorter one is larger for slides than for viewgraphs. As a result, slide foils can be used for viewgraphs, but not vice versa; on the other hand, viewgraphs can accommodate a bit more text.

.Vw [*n*] [*i* Same as **.VS**, except that foil size is 7″ wide x 5″ high.

.Vh [*n*] [*i*] [*d*]
: Same as **.VS**, except that foil size is 5″x7″.

.VW [*n*] [*i* Same as **.VS**, except that foil size is 7″x5.4″.

.VH [*n*] [*i* Same as **.VS**, except that foil size is 7″x9″.

.Sw [*n*] [*i* Same as **.VS**, except that foil size is 7″x5″.

.Sh [*n*] [*i* Same as **.VS**, except that foil size is 5″x7″.

.SW [*n*] [*i* Same as **.VS**, except that foil size is 7″x5.4″.

.SH [*n*] [*i* Same as **.VS**, except that foil size is 7″x9″.

.A [*x*] Place text that follows at the first indentation level (left margin); the presence of *x* suppresses the half-line spacing from the preceding text.

.B [*m*] [*s*] Place text that follows at the second indentation level; text is preceded by a mark; *m* is the mark (default is a large bullet); *s* is the increment or decrement to the point size of the mark with respect to the *prevailing* point size (default is 0); if *s* is 100, it causes the point size of the mark to be the same as that of the *default* mark.

.C [*m*] [*s*] Same as **.B**, but for the third indentation level; default mark is a dash.

.D [*m*] [*s*]	Same as **.B** , but for the fourth indentation level; default mark is a small bullet.
.T *string*	*string* is printed as an over-size, centered title.
.I [*in*] [*a* [*x*]]	Change the current text indent (does not affect titles); *in* is the indent (in inches unless dimensioned, default is 0); if *in* is signed, it is an increment or decrement; the presence of *a* invokes the **.A** macro (see below) and passes *x* (if any) to it.
.S j"[*p*] [*l*	Set the point size and line length; *p* is the point size (default is "previous"); if *p* is 100, the point size reverts to the *initial* default for the current foil-start macro; if *p* is signed, it is an increment or decrement (default is 18 for **.VS**, **.VH**, and **.SH**, and 14 for the other foil-start macros); *l* is the line length (in inches unless dimensioned; default is 4.2" for **.Vh**, 3.8" for **.Sh**, 5" for **.SH**, and 6" for the other foil-start macros).
.DF *n f* [*n f* . . .]	Define font positions; may not appear within a foil's input text (i.e., it may only appear after all the input text for a foil, but before the next foil-start macro); *n* is the position of font *f*; up to four pairs may be specified; the first font named becomes the *prevailing* font; the initial setting is (**H** is a synonym for **G**): **.DF 1 H 2 I 3 B 4 S**
.DV [*a*] [*b*] [*c*] [*d*]	Alter the vertical spacing between indentation levels; *a* is the spacing for **.A**, *b* is for **.B**, *c* is for **.C**, and *d* is for **.D**; all non-null arguments must be dimensioned; null arguments leave the corresponding spacing unaffected; initial setting is: **.DV .5v .5v .5v 0v**
.U *str1* [*str2*]	Underline *str1* and concatenate *str2* (if any) to it.

The last four macros in the above list do not cause a break; the **.I** macro causes a break only if it is invoked with more than one argument; all the other macros cause a break.

The macro package also recognizes the following upper-case synonyms for the corresponding lower-case **troff** requests:

.AD .BR .CE .FI .HY .NA .NF .NH .NX .SO .SP .TA .TI

The **Tm** string produces the trademark symbol.

FILES

/usr/lib/tmac/tmac.v
/usr/lib/macros/vmca

SEE ALSO

eqn(1), grap(1), mvt(1), pic(1), tbl(1), troff(1).

BUGS

The **.VW** and **.SW** foils are meant to be 9" wide by 7" high, but because the typesetter paper is generally only 8" wide, they are printed 7" wide by 5.4" high and have to be enlarged by a factor of 9/7 before use as viewgraphs; this makes them less than totally useful.

NAME

regexp – regular expression compile and match routines

SYNOPSIS

#define INIT <declarations>
#define GETC( ) <getc code>
#define PEEKC( ) <peekc code>
#define UNGETC(c) <ungetc code>
#define RETURN(pointer) <return code>
#define ERROR(val) <error code>

#include <regexp.h>

char *__*compile__ (instring, **expbuf,** endbuf, eof)"
char *__*instring,__ *__*expbuf,__ *__*endbuf;__
int **eof;**

int step (string, expbuf)
char *string, *expbuf;

extern char *loc1, *loc2, *locs;

extern int circf, sed, nbra;

DESCRIPTION

These general-purpose regular expression matching routines in the form of *ed*(1), are defined in *<regexp.h>* . Programs such as *ed*(1), *sed*(1), *grep*(1), *bs*(1), *expr*(1), etc., which perform regular expression matching use this source file. In this way, only this file need be changed to maintain regular expression compatibility.

The interface to this file is unpleasantly complex. Programs that include this file must have the following five macros declared before the "#include <regexp.h>" statement. These macros are used by the *compile* routine.

GETC( )	Return the value of the next character in the regular expression pattern. Successive calls to GETC( ) should return successive characters of the regular expression.
PEEKC( )	Return the next character in the regular expression. Successive calls to PEEKC( ) should return the same character [which should also be the next character returned by GETC( )].
UNGETC(*c*)	Cause the argument *c* to be returned by the next call to GETC( ) [and PEEKC( )]. No more that one character of pushback is ever needed and this character is guaranteed to be the last character read by GETC( ). The value of the macro UNGETC(*c*) is always ignored.
RETURN(*pointer*)	This macro is used on normal exit of the *compile* routine. The value of the argument *pointer* is a pointer to the character after the last character of the compiled regular expression. This is useful to programs which have memory allocation to manage.
ERROR(*val*)	This is the abnormal return from the *compile* routine. The argument *val* is an error number (see table below for meanings). This call should never return.

ERROR	MEANING
11	Range endpoint too large.
16	Bad number.
25	"\digit" out of range.
36	Illegal or missing delimiter.
41	No remembered search string.
42	\(\) imbalance.
43	Too many \(.
44	More than 2 numbers given in \{ \}.
45	} expected after \.
46	First number exceeds second in \{ \}.
49	[ ] imbalance.
50	Regular expression overflow.

The syntax of the *compile* routine follows:

```
compile(instring, expbuf, endbuf, eof)
```

The first parameter *instring* is never used explicitly by the *compile* routine but is useful for programs that pass down different pointers to input characters. It is sometimes used in the INIT declaration (see below). Programs that call functions to input characters or hold characters in an external array can pass down a value of ((char *) 0) for this parameter.

The parameter *expbuf* is a character pointer. It points to the place where the compiled regular expression will be placed.

The parameter *endbuf* is one more than the highest address where the compiled regular expression may be placed. If the compiled expression cannot fit in (*endbuf–expbuf*) bytes, a call to ERROR(50) is made.

The parameter *eof* is the character which marks the end of the regular expression. For example, in *ed*(1), this character is usually a /.

Each program that includes this file must have a **#define** statement for INIT. This definition is placed right after the declaration for the function *compile* and the opening curly brace ({). It is used for dependent declarations and initializations. Most often it is used to set a register variable to point the beginning of the regular expression so that this register variable can be used in the declarations for GETC( ), PEEKC( ) and UNGETC( ). Otherwise it can be used to declare external variables that might be used by GETC( ), PEEKC( ) and UNGETC( ). See the example below of the declarations taken from *grep*(1).

There are other functions in this file which perform actual regular expression matching, one of which is the function *step*. The call to *step* follows:

```
step(string, expbuf)
```

The first parameter to *step* is a pointer to a string of characters to be checked for a match. This string should be null terminated.

The parameter *expbuf* is the compiled regular expression which was obtained by a call of the function *compile*.

The function *step* returns non-zero if the given string matches the regular expression, and zero if the expressions do not match. If there is a match, two external character pointers are set as a side effect to the call to *step*. The variable set in *step* is *loc1*. This is a pointer to the first character that matched the regular expression. The variable *loc2*, which is set by the function *advance*, points to the character after the last character that matches the regular expression. Thus if the regular expression matches the entire line, *loc1* points to the first character of *string* and *loc2* points to the null at the end of *string*.

Step uses the external variable *circf* which is set by *compile* if the regular expression begins with ^. If this is set then *step* tries to match the regular expression to the beginning of the string only. If more than one regular expression is to be compiled before the first is executed the value of *circf* should be saved for each compiled expression and *circf* should be set to that saved value before each call to *step*.

The function *advance* is called from *step* with the same arguments as *step*. The purpose of *step* is to step through the *string* argument and call *advance* until *advance* returns non-zero indicating a match or until the end of *string* is reached. If one wants to constrain *string* to the beginning of the line in all cases, *step* need not be called; simply call *advance*.

When *advance* encounters a * or \{ \} sequence in the regular expression, it advances its pointer to the string to be matched as far as possible and recursively calls itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, *advance* backs up along the string until it finds a match or reaches the point in the string that initially matched the * or \{ \}. It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external character pointer *locs* is equal to the point in the string at sometime during the backing up process, *advance* breaks out of the loop that backs up and returns zero. This is used by *ed*(1) and *sed*(1) for substitutions done globally (not just the first occurrence, but the whole line) so, for example, expressions like **s/y*//g** do not loop forever.

The additional external variables *sed* and *nbra* are used for special purposes.

EXAMPLES

The following example shows how the regular expression macros and calls look from *grep*(1):

```
#define INIT            register char *sp = instring;
#define GETC()          (*sp++)
#define PEEKC()         (*sp)
#define UNGETC(c)       (– –sp)
#define RETURN(c)       return;
#define ERROR(c)        regerr()

#include <regexp.h>
...
                (void) compile(*argv, expbuf, &expbuf[ESIZE], '\0');
...
                if (step(linebuf, expbuf))
                        succeed();
```

SEE ALSO

ed(1), expr(1), grep(1), sed(1)

NAME

resolver – resolver configuration file

SYNOPSIS

/etc/resolv.conf

DESCRIPTION

The resolver configuration file contains information that is read by the resolver routines the first time they are invoked by a process. The file is designed to be human readable and contains a list of name-value pairs that provide various types of resolver information.

On a normally configured system this file should not be necessary. The only name server to be queried will be on the local machine and the domain name is retrieved from the system.

The different configuration options are:

nameserver
followed by the Internet address (in dot notation) of a name server that the resolver should query. At least one name server should be listed. Up to MAXNS (currently 3) name servers may be listed, in that case the resolver library queries tries them in the order listed. If no **nameserver** entries are present, the default is to use the name server on the local machine. (The algorithm used is to try a name server, and if the query times out, try the next, until out of name servers, then repeat trying all the name servers until a maximum number of retries are made).

domain
followed by a domain name, that is the default domain to append to names that do not have a dot in them. If no **domain** entries are present, the domain returned by *gethostname* (2) is used (everything after the first '.'). Finally, if the host name does not contain a domain part, the root domain is assumed.

The name value pair must appear on a single line, and the keyword (e.g. **nameserver**) must start the line. The value follows the keyword, separated by white space.

FILES

/etc/resolv.conf

SEE ALSO

gethostbyname(3N), resolver(3), named(8)
Name Server Operations Guide for BIND

NAME

stat – data returned by stat system call

SYNOPSIS

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

DESCRIPTION

The system calls *stat* and *fstat* return data whose structure is defined by this include file. The encoding of the field *st_mode* is defined in this file also.

Structure of the result of stat

```
struct  stat
{
        dev_t    st_dev;
        ulong    st_ino;
        ushort   st_mode;
        short    st_nlink;
        ushort   st_uid;
        ushort   st_gid;
        dev_t    st_rdev;
        off_t    st_size;
        time_t   st_atime;
        time_t   st_mtime;
        time_t   st_ctime;
};

#define S_IFMT     0170000    /* type of file */
#define S_IFDIR    0040000    /* directory */
#define S_IFCHR    0020000    /* character special */
#define S_IFBLK    0060000    /* block special */
#define S_IFREG    0100000    /* regular */
#define S_IFIFO    0010000    /* fifo */
#define S_IFLINK   0120000    /* symbolic link */
#define S_ISUID    04000      /* set user id on execution */
#define S_ISGID    02000      /* set group id on execution */
#define S_ISVTX    01000      /* save swapped text even after use */
#define S_IREAD    00400      /* read permission, owner */
#define S_IWRITE   00200      /* write permission, owner */
#define S_IEXEC    00100      /* execute/search permission, owner */
#define S_ENFMT    S_ISGID    /* record locking enforcement flag */
#define S_IRWXU    00700      /* read,write, execute: owner */
#define S_IRUSR    00400      /* read permission: owner */
#define S_IWUSR    00200      /* write permission: owner */
#define S_IXUSR    00100      /* execute permission: owner */
#define S_IRWXG    00070      /* read, write, execute: group */
#define S_IRGRP    00040      /* read permission: group */
#define S_IWGRP    00020      /* write permission: group */
#define S_IXGRP    00010      /* execute permission: group */
#define S_IRWXO    00007      /* read, write, execute: other */
#define S_IROTH    00004      /* read permission: other */
#define S_IWOTH    00002      /* write permission: other */
#define S_IXOTH    00001      /* execute permission: other */
```

SEE ALSO

stat(2), types(5).

NAME

term – conventional names for terminals

DESCRIPTION

These names are used by certain commands (e.g., *man*(1), *tabs*(1), *tput*(1), *vi*(1) and *curses*(3X)) and are maintained as part of the shell environment in the environment variable TERM (see *sh*(1), *profile*(4), and *environ*(5)).

Entries in *terminfo*(4) source files consist of a number of comma-separated fields. (To obtain the source description for a terminal, use the **–I** option of *infocmp*(1M).) White space after each comma is ignored. The first line of each terminal description in the *terminfo*(4) database gives the names by which *terminfo*(4) knows the terminal, separated by bar (|) characters. The first name given is the most common abbreviation for the terminal (this is the one to use to set the environment variable TERMINFO in *$HOME/.profile;* see *profile*(4)), the last name given should be a long name fully identifying the terminal, and all others are understood as synonyms for the terminal name. All names but the last should contain no blanks and must be unique in the first 14 characters; the last name may contain blanks for readability.

Terminal names (except for the last, verbose entry) should be chosen using the following conventions. The particular piece of hardware making up the terminal should use a unique root name, for example, for the AT&T 4425 terminal, **att4425**. This name should not contain hyphens, except that synonyms may be chosen that do not conflict with other names. Up to 8 characters, chosen from [a–z0–9], make up a basic terminal name. Names should generally be based on original vendors, rather than local distributors. A terminal acquired from one vendor should not have more than one distinct basic name. Terminal sub-models, hardware modes, or user preferences should be indicated by appending a hyphen and an indicator of the mode. Thus, an AT&T 4425 terminal in 132 column mode would be **att4425–w**. The following suffixes should be used where possible:

Suffix	**Meaning**	**Example**
–w	Wide mode (more than 80 columns)	att4425–w
–am	With auto. margins (usually default)	vt100–am
–nam	Without automatic margins	vt100–nam
–n	Number of lines on the screen	aaa–60
–na	No arrow keys (leave them in local)	c100–na
–np	Number of pages of memory	c100–4p
–rv	Reverse video	att4415–rv

To avoid conflicts with the naming conventions used in describing the different modes of a terminal (e.g., **–w**), it is recommended that a terminal's root name not contain hyphens. Further, it is good practice to make all terminal names used in the *terminfo*(4) database unique. Terminal entries that are present only for inclusion in other entries via the **use=** facilities should have a '+' in their name, as in **4415+nl**.

Some of the known terminal names may include the following (for a complete list, type: **ls -C /usr/lib/terminfo/?**):

2621,hp2621	Hewlett-Packard 2621 series
2631	Hewlett-Packard 2631 line printer
2631–c	Hewlett-Packard 2631 line printer - compressed mode
2631–e	Hewlett-Packard 2631 line printer - expanded mode
2640,hp2640	Hewlett-Packard 2640 series
2645,hp2645	Hewlett-Packard 2645 series
3270	IBM Model 3270
33,tty33	AT&T Teletype Model 33 KSR
35,tty35	AT&T Teletype Model 35 KSR

37,tty37	AT&T Teletype Model 37 KSR
4000a	Trendata 4000a
4014,tek4014	TEKTRONIX 4014
40,tty40	AT&T Teletype Dataspeed 40/2
43,tty43	AT&T Teletype Model 43 KSR
4410,5410	AT&T 4410/5410 terminal in 80-column mode - version 2
4410–nfk,5410–nfk	AT&T 4410/5410 without function keys - version 1
4410–nsl,5410–nsl	AT&T 4410/5410 without pln defined
4410–w,5410–w	AT&T 4410/5410 in 132-column mode
4410v1,5410v1	AT&T 4410/5410 terminal in 80-column mode - version 1
4410v1–w,5410v1–w	AT&T 4410/5410 terminal in 132-column mode - version 1
4415,5420	AT&T 4415/5420 in 80-column mode
4415–nl,5420–nl	AT&T 4415/5420 without changing labels
4415–rv,5420–rv	AT&T 4415/5420 80 columns in reverse video
4415–rv–nl,5420–rv–nl	AT&T 4415/5420 reverse video without changing labels
4415–w,5420–w	AT&T 4415/5420 in 132-column mode
4415–w–nl,5420–w–nl	AT&T 4415/5420 in 132-column mode without changing labels
4415–w–rv,5420–w–rv	AT&T 4415/5420 132 columns in reverse video
4415–w–rv–nl,5420–w–rv–nl	AT&T 4415/5420 132 columns reverse video without changing labels
4418,5418	AT&T 5418 in 80-column mode
4418–w,5418–w	AT&T 5418 in 132-column mode
4420	AT&T Teletype Model 4420
4424	AT&T Teletype Model 4424
4424-2	AT&T Teletype Model 4424 in display function group ii
4425,5425	AT&T 4425/5425
4425–fk,5425–fk	AT&T 4425/5425 without function keys
4425–nl,5425–nl	AT&T 4425/5425 without changing labels in 80-column mode
4425–w,5425–w	AT&T 4425/5425 in 132-column mode
4425–w–fk,5425–w–fk	AT&T 4425/5425 without function keys in 132-column mode
4425–nl–w,5425–nl–w	AT&T 4425/5425 without changing labels in 132-column mode
4426	AT&T Teletype Model 4426S
450	DASI 450 (same as Diablo 1620)
450–12	DASI 450 in 12-pitch mode
500,att500	AT&T-IS 500 terminal
510,510a	AT&T 510/510a in 80-column mode
513bct,att513	AT&T 513 bct terminal
5320	AT&T 5320 hardcopy terminal
5420_2	AT&T 5420 model 2 in 80-column mode
5420_2–w	AT&T 5420 model 2 in 132-column mode
5620,dmd	AT&T 5620 terminal 88 columns
5620–24,dmd–24	AT&T Teletype Model DMD 5620 in a 24x80 layer
5620–34,dmd–34	AT&T Teletype Model DMD 5620 in a 34x80 layer
610,610bct	AT&T 610 bct terminal in 80-column mode
610–w,610bct–w	AT&T 610 bct terminal in 132-column mode
7300,pc7300,unix_pc	AT&T UNIX PC Model 7300
735,ti	Texas Instruments TI735 and TI725
745	Texas Instruments TI745
dumb	generic name for terminals that lack reverse

	line-feed and other special escape sequences
hp	Hewlett-Packard (same as 2645)
lp	generic name for a line printer
pt505	AT&T Personal Terminal 505 (22 lines)
pt505–24	AT&T Personal Terminal 505 (24-line mode)
sync	generic name for synchronous Teletype Model 4540-compatible terminals

Commands whose behavior depends on the type of terminal should accept arguments of the form –T*term* where *term* is one of the names given above; if no such argument is present, such commands should obtain the terminal type from the environment variable TERM, which, in turn, should contain *term*.

FILES

/usr/lib/terminfo/?/* compiled terminal description database

SEE ALSO

curses(3X), environ(5), infocmp(1M) man(1), profile(4), sh(1), stty(1), tabs(1), terminfo(4), tplot(1G), tput(1), vi(1)

NOTES

Not all programs follow the above naming conventions.

NAME

termcap – terminal capability data base

SYNOPSIS

/etc/termcap

DESCRIPTION

Termcap is a data base describing terminals, used, *e.g.*, by *vi*(1) and *curses*(3X). Terminals are described in *termcap* by giving a set of capabilities that they have and by describing how operations are performed. Padding requirements and initialization sequences are included in *termcap*. Refer to *curses(3X)* for additional information concerning terminal screen handling and optimization package.

Entries in *termcap* consist of a number of ':'-separated fields. The first entry for each terminal gives the names that are known for the terminal, separated by '|' characters. The first name is always two characters long and is used by older systems which store the terminal type in a 16-bit word in a system-wide data base. The second name given is the most common abbreviation for the terminal, the last name given should be a long name fully identifying the terminal, and all others are understood as synonyms for the terminal name. All names but the first and last should be in lower case and contain no blanks; the last name may well contain upper case and blanks for readability.

Terminal names (except for the last, verbose entry) should be chosen using the following conventions. The particular piece of hardware making up the terminal should have a root name chosen, thus "hp2621". This name should not contain hyphens. Modes that the hardware can be in or user preferences should be indicated by appending a hyphen and an indicator of the mode. Therefore, a "vt100" in 132-column mode would be "vt100-w". The following suffixes should be used where possible:

Suffix	**Meaning**	**Example**
-w	Wide mode (more than 80 columns)	vt100-w
-am	With automatic margins (usually default)	vt100-am
-nam	Without automatic margins	vt100-nam
-*n*	Number of lines on the screen	aaa-60
-na	No arrow keys (leave them in local)	concept100-na
-*n*p	Number of pages of memory	concept100-4p
-rv	Reverse video	concept100-rv

CAPABILITIES

The characters in the *Notes* field in the table have the following meanings (more than one may apply to a capability):

N indicates numeric parameter(s)
P indicates that padding may be specified
* indicates that padding may be based on the number of lines affected
o indicates capability is obsolete

"Obsolete" capabilities have no *terminfo* equivalents, since they were considered useless, or are subsumed by other capabilities. New software should not rely on them at all.

Name	**Type**	**Notes**	**Description**
ae	str	(P)	End alternate character set
AL	str	(NP*)	Add *n* new blank lines
al	str	(P*)	Add new blank line
am	bool		Terminal has automatic margins
as	str	(P)	Start alternate character set

bc	str	(o)	Backspace if not **^H**
bl	str	(P)	Audible signal (bell)
bs	bool	(o)	Terminal can backspace with **^H**
bt	str	(P)	Back tab
bw	bool		**le** (backspace) wraps from column 0 to last column
CC	str		Terminal settable command character in prototype
cd	str	(P*)	Clear to end of display
ce	str	(P)	Clear to end of line
ch	str	(NP)	Set cursor column (horizontal position)
cl	str	(P*)	Clear screen and home cursor
CM	str	(NP)	Memory-relative cursor addressing
cm	str	(NP)	Screen-relative cursor motion
co	num		Number of columns in a line (See BUGS section below)
cr	str	(P)	Carriage return
cs	str	(NP)	Change scrolling region (VT100)
ct	str	(P)	Clear all tab stops
cv	str	(NP)	Set cursor row (vertical position)
da	bool		Display may be retained above the screen
dB	num	(o)	Milliseconds of **bs** delay needed (default 0)
db	bool		Display may be retained below the screen
DC	str	(NP*)	Delete *n* characters
dC	num	(o)	Milliseconds of **cr** delay needed (default 0)
dc	str	(P*)	Delete character
dF	num	(o)	Milliseconds of **ff** delay needed (default 0)
DL	str	(NP*)	Delete *n* lines
dl	str	(P*)	Delete line
dm	str		Enter delete mode
dN	num	(o)	Milliseconds of **nl** delay needed (default 0)
DO	str	(NP*)	Move cursor down *n* lines
do	str		Down one line
ds	str		Disable status line
dT	num	(o)	Milliseconds of horizontal tab delay needed (default 0)
dV	num	(o)	Milliseconds of vertical tab delay needed (default 0)
ec	str	(NP)	Erase *n* characters
ed	str		End delete mode
ei	str		End insert mode
eo	bool		Can erase overstrikes with a blank
EP	bool	(o)	Even parity
es	bool		Escape can be used on the status line
ff	str	(P*)	Hardcopy terminal page eject
fs	str		Return from status line
gn	bool		Generic line type (*e.g.* dialup, switch)
hc	bool		Hardcopy terminal
HD	bool	(o)	Half-duplex
hd	str		Half-line down (forward 1/2 linefeed)
ho	str	(P)	Home cursor
hs	bool		Has extra "status line"
hu	str		Half-line up (reverse 1/2 linefeed)
hz	bool		Cannot print s (Hazeltine)
i1-i3	str		Terminal initialization strings (*terminfo* only)
IC	str	(NP*)	Insert *n* blank characters
ic	str	(P*)	Insert character
if	str		Name of file containing initialization string
im	str		Enter insert mode

in	bool		Insert mode distinguishes nulls
iP	str		Pathname of program for initialization (*terminfo* only)
ip	str	(P*)	Insert pad after character inserted
is	str		Terminal initialization string (*termcap* only)
it	num		Tabs initially every *n* positions
K1	str		Sent by keypad upper left
K2	str		Sent by keypad upper right
K3	str		Sent by keypad center
K4	str		Sent by keypad lower left
K5	str		Sent by keypad lower right
k0-k9	str		Sent by function keys 0-9
kA	str		Sent by insert-line key
ka	str		Sent by clear-all-tabs key
kb	str		Sent by backspace key
kC	str		Sent by clear-screen or erase key
kD	str		Sent by delete-character key
kd	str		Sent by down-arrow key
kE	str		Sent by clear-to-end-of-line key
ke	str		Out of "keypad transmit" mode
kF	str		Sent by scroll-forward/down key
kH	str		Sent by home-down key
kh	str		Sent by home key
kI	str		Sent by insert-character or enter-insert-mode key
kL	str		Sent by delete-line key
kl	str		Sent by left-arrow key
kM	str		Sent by insert key while in insert mode
km	bool		Has a "meta" key (shift, sets parity bit)
kN	str		Sent by next-page key
kn	num	(o)	Number of function (**k0–k9**) keys (default 0)
ko	str	(o)	Termcap entries for other non-function keys
kP	str		Sent by previous-page key
kR	str		Sent by scroll-backward/up key
kr	str		Sent by right-arrow key
kS	str		Sent by clear-to-end-of-screen key
ks	str		Put terminal in "keypad transmit" mode
kT	str		Sent by set-tab key
kt	str		Sent by clear-tab key
ku	str		Sent by up-arrow key
l0-l9	str		Labels on function keys if not "f*n*"
LC	bool	(o)	Lower-case only
LE	str	(NP)	Move cursor left *n* positions
le	str	(P)	Move cursor left one position
li	num		Number of lines on screen or page (See BUGS section below)
ll	str		Last line, first column
lm	num		Lines of memory if > **li** (0 means varies)
ma	str	(o)	Arrow key map (used by *vi* version 2 only)
mb	str		Turn on blinking attribute
md	str		Turn on bold (extra bright) attribute
me	str		Turn off all attributes
mh	str		Turn on half-bright attribute
mi	bool		Safe to move while in insert mode
mk	str		Turn on blank attribute (characters invisible)
ml	str	(o)	Memory lock on above cursor
mm	str		Turn on "meta mode" (8th bit)

mo	str		Turn off "meta mode"
mp	str		Turn on protected attribute
mr	str		Turn on reverse-video attibute
ms	bool		Safe to move in standout modes
mu	str	(o)	Memory unlock (turn off memory lock)
nc	bool	(o)	No correctly-working **cr** (Datamedia 2500, Hazeltine 2000)
nd	str		Non-destructive space (cursor right)
NL	bool	(o)	**\n** is newline, not line feed
nl	str	(o)	Newline character if not **\n**
ns	bool	(o)	Terminal is a CRT but doesn't scroll
nw	str	(P)	Newline (behaves like **cr** followed by **do**)
OP	bool	(o)	Odd parity
os	bool		Terminal overstrikes
pb	num		Lowest baud where delays are required
pc	str		Pad character (default NUL)
pf	str		Turn off the printer
pk	str		Program function key *n* to type string *s* (*terminfo* only)
pl	str		Program function key *n* to execute string *s* (*terminfo* only)
pO	str	(N)	Turn on the printer for *n* bytes
po	str		Turn on the printer
ps	str		Print contents of the screen
pt	bool	(o)	Has hardware tabs (may need to be set with **is**)
px	str		Program function key *n* to transmit string *s* (*terminfo* only)
r1-r3	str		Reset terminal completely to sane modes (*terminfo* only)
rc	str	(P)	Restore cursor to position of last **sc**
rf	str		Name of file containing reset codes
RI	str	(NP)	Move cursor right *n* positions
rp	str	(NP*)	Repeat character *c* *n* times
rs	str		Reset terminal completely to sane modes (*termcap* only)
sa	str	(NP)	Define the video attributes
sc	str	(P)	Save cursor position
se	str		End standout mode
SF	str	(NP*)	Scroll forward *n* lines
sf	str	(P)	Scroll text up
sg	num		Number of garbage chars left by **so** or **se** (default 0)
so	str		Begin standout mode
SR	str	(NP*)	Scroll backward *n* lines
sr	str	(P)	Scroll text down
st	str		Set a tab in all rows, current column
ta	str	(P)	Tab to next 8-position hardware tab stop
tc	str		Entry of similar terminal – must be last
te	str		String to end programs that use *termcap*
ti	str		String to begin programs that use *termcap*
ts	str	(N)	Go to status line, column *n*
UC	bool	(o)	Upper-case only
uc	str		Underscore one character and move past it
ue	str		End underscore mode
ug	num		Number of garbage chars left by **us** or **ue** (default 0)
ul	bool		Underline character overstrikes
UP	str	(NP*)	Move cursor up *n* lines
up	str		Upline (cursor up)
us	str		Start underscore mode
vb	str		Visible bell (must not move cursor)
ve	str		Make cursor appear normal (undo **vs**/**vi**)

vi	str		Make cursor invisible
vs	str		Make cursor very visible
vt	num		Virtual terminal number (not supported on all systems)
wi	str	(N)	Set current window
ws	num		Number of columns in status line
xb	bool		Beehive (f1=ESC, f2=^C)
xn	bool		Newline ignored after 80 cols (Concept)
xo	bool		Terminal uses xoff/xon (DC3/DC1) handshaking
xr	bool	(o)	Return acts like **ce cr nl** (Delta Data)
xs	bool		Standout not erased by overwriting (Hewlett-Packard)
xt	bool		Tabs ruin, magic **so** char (Teleray 1061)
xx	bool	(o)	Tektronix 4025 insert-line

A Sample Entry

The following entry, which describes the Concept–100, is among the more complex entries in the *termcap* file as of this writing.

```
ca | concept100 | c100 | concept | c104 | concept100-4p | HDS Concept–100:\
        :al=3*\E^R:am:bl=^G:cd=16*\E^C:ce=16\E^U:cl=2*^L:cm=\Ea%+ %+ :\
        :co#80:.cr=9^M:db:dc=16\E^A:dl=3*\E^B:do=^J:ei=\E\200:eo:im=\E^P:in:\
        :ip=16*:is=\EU\Ef\E7\E5\E8\El\ENH\EK\E\200\Eo&\200\Eo\47\E:k1=\E5:\
        :k2=\E6:k3=\E7:kb=^h:kd=\E<:ke=\Ex:kh=\E?:kl=\E>:kr=\E=:ks=\EX:\
        :ku=\E;:le=^H:li#24:mb=\EC:me=\EN\200:mh=\EE:mi:mk=\EH:mp=\EI:\
        :mr=\ED:nd=\E=:pb#9600:rp=0.2*\Er%.%+ :se=\Ed\Ee:sf=^J:so=\EE\ED:\
        :.ta=8\t:te=\Ev   \200\200\200\200\200\200\Ep\r\n:\
        :ti=\EU\Ev  8p\Ep\r:ue=\Eg:ul:up=\E;:us=\EG:\
        :vb=\Ek\200\200\200\200\200\200\200\200\200\200\200\200\200\200\EK:\
        :ve=\Ew:vs=\EW:vt#8:xn:\
        :bs:cr=^M:dC#9:dT#8:nl=^J:ta=^I:pt:
```

Entries may continue onto multiple lines by giving a \ as the last character of a line, and empty fields may be included for readability (here between the last field on a line and the first field on the next). Comments may be included on lines beginning with "#".

Types of Capabilities

Capabilities in *termcap* are of three types: Boolean capabilities, which indicate particular features that the terminal has; numeric capabilities, giving the size of the display or the size of other attributes; and string capabilities, which give character sequences that can be used to perform particular terminal operations. All capabilities have two-letter codes. For instance, the fact that the Concept has *automatic margins* (*i.e.*, an automatic return and linefeed when the end of a line is reached) is indicated by the Boolean capability **am**. Hence the description of the Concept includes **am**.

Numeric capabilities are followed by the character '#' then the value. In the example above **co**, which indicates the number of columns the display has, gives the value '80' for the Concept.

Finally, string-valued capabilities, such as **ce** (clear-to-end-of-line sequence) are given by the two-letter code, an '=', then a string ending at the next following ':'. A delay in milliseconds may appear after the '=' in such a capability, which causes padding characters to be supplied by *tputs* after the remainder of the string is sent to provide this delay. The delay can be either a number, *e.g.* '20', or a number followed by an '*', *i.e.*, '3*'. An '*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-line padding required. (In the case of insert-character, the factor is still the number of *lines* affected; this is always 1 unless the terminal has **in** and the software uses it.) When an '*' is specified, it is sometimes useful to give a delay of the form '3.5' to specify a delay per line to tenths of milliseconds. (Only one decimal place is allowed.)

A number of escape sequences are provided in the string-valued capabilities for easy encoding of control characters there. **\E** maps to an ESC character, **^X** maps to a control-X for any appropriate X, and the sequences **\n \r \t \b \f** map to linefeed, return, tab, backspace, and formfeed, respectively. Finally, characters may be given as three octal digits after a ****, and the characters **^** and **** may be given as **\^** and \. If it is necessary to place a **:** in a capability it must be escaped in octal as **\072.** If it is necessary to place a NUL character in a string capability it must be encoded as **\200.** (The routines that deal with *termcap* use C strings and strip the high bits of the output very late, so that a **\200** comes out as a **\000** would.)

Sometimes individual capabilities must be commented out. To do this, put a period before the capability name. For example, see the first **cr** and **ta** in the example above.

Preparing Descriptions

We now outline how to prepare descriptions of terminals. The most effective way to prepare a terminal description is by imitating the description of a similar terminal in *termcap* and to build up a description gradually, using partial descriptions with *vi* to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the *termcap* file to describe it or bugs in *vi*. To easily test a new terminal description you can set the environment variable **TERMCAP** to the absolute pathname of a file containing the description you are working on and programs will look there rather than in */etc/termcap*. **TERMCAP** can also be set to the *termcap* entry itself to avoid reading the file when starting up a program.

To get the padding for insert-line right (if the terminal manufacturer did not document it), a severe test is to use *vi* to edit */etc/passwd* at 9600 baud, delete roughly 16 lines from the middle of the screen, then hit the 'u' key several times quickly. If the display messes up, more padding is usually needed. A similar test can be used for insert-character.

Basic Capabilities

The number of columns on each line of the display is given by the **co** numeric capability. If the display is a CRT, then the number of lines on the screen is given by the **li** capability. If the display wraps around to the beginning of the next line when the cursor reaches the right margin, then it should have the **am** capability. If the terminal can clear its screen, the code to do this is given by the **cl** string capability. If the terminal overstrikes (rather than clearing the position when a character is overwritten), it should have the **os** capability. If the terminal is a printing terminal, with no soft copy unit, give it both **hc** and **os**. (**os** applies to storage scope terminals, such as the Tektronix 4010 series, as well as to hard copy and APL terminals.) If there is a code to move the cursor to the left edge of the current row, give this as **cr**. (Normally this will be carriage-return, **^M**.) If there is a code to produce an audible signal (bell, beep, *etc.*), give this as **bl**.

If there is a code (such as backspace) to move the cursor one position to the left, that capability should be given as **le**. Similarly, codes to move to the right, up, and down should be given as **nd**, **up**, and **do**, respectively. These *local cursor motions* should not alter the text they pass over; for example, you would not normally use "nd= " unless the terminal has the **os** capability, because the space would erase the character moved over.

A very important point here is that the local cursor motions encoded in *termcap* have undefined behavior at the left and top edges of a CRT display. Programs should never attempt to backspace around the left edge, unless **bw** is given, and never attempt to go up off the top using local cursor motions.

In order to scroll text up, a program goes to the bottom left corner of the screen and sends the **sf** (index) string. To scroll text down, a program goes to the top left corner of the screen and sends the **sr** (reverse index) string. The strings **sf** and **sr** have undefined behavior when not on their respective corners of the screen. Parameterized versions of the scrolling sequences are **SF** and **SR**, which have the same semantics as **sf** and **sr** except that they take one parameter and scroll that many lines. They also have undefined behavior except at the appropriate corner of the screen.

The **am** capability tells whether the cursor sticks at the right edge of the screen when text is output there, but this does not necessarily apply to **nd** from the last column. Leftward local motion is defined from the left edge only when **bw** is given; then an **le** from the left edge will move to the right edge of the previous row. This is useful for drawing a box around the edge of the screen, for example. If the terminal has switch-selectable automatic margins, the *termcap* description usually assumes that this feature is on, *i.e.*, **am**. If the terminal has a command that moves to the first column of the next line, that command can be given as **nw** (newline). It is permissible for this to clear the remainder of the current line, so if the terminal has no correctly-working CR and LF it may still be possible to craft a working **nw** out of one or both of them.

These capabilities suffice to describe hardcopy and "glass-tty" terminals. Thus the Teletype model 33 is described as

```
        T3 | tty33 | 33 | tty | Teletype model 33:\
                :bl=^G:co#72:cr=^M:do=^J:hc:os:
```

and the Lear Siegler ADM–3 is described as

```
        l3 | adm3 | 3 | LSI ADM-3:\
                :am:bl=^G:cl=^Z:co#80:cr=^M:do=^J:le=^H:li#24:sf=^J:
```

Parameterized Strings

Cursor addressing and other strings requiring parameters are described by a parameterized string capability, with *printf*(3S)-like escapes %x in it, while other characters are passed through unchanged. For example, to address the cursor the **cm** capability is given, using two parameters: the row and column to move to. (Rows and columns are numbered from zero and refer to the physical screen visible to the user, not to any unseen memory. If the terminal has memory-relative cursor addressing, that can be indicated by an analogous **CM** capability.)

The % encodings have the following meanings:

%%	output '%'
%d	output value as in *printf* %d
%2	output value as in *printf* %2d
%3	output value as in *printf* %3d
%.	output value as in *printf* %c
%+*x*	add *x* to value, then do %.
%>*xy*	if value > *x*1then add *y*, no output
%r	reverse order of two parameters, no output
%i	increment by one, no output
%n	exclusive-or all parameters with 0140 (Datamedia 2500)
%B	BCD (16*(value/10)) + (value%10), no output
%D	Reverse coding (value – 2*(value%16)), no output (Delta Data)

Consider the Hewlett-Packard 2645, which, to get to row 3 and column 12, needs to be sent "\E&a12c03Y" padded for 6 milliseconds. Note that the order of the row and column coordinates is reversed here and that the row and column are sent as two-digit integers. Thus its **cm** capability is "cm=6\E&%r%2c%2Y".

The Microterm ACT-IV needs the current row and column sent simply encoded in binary preceded by a **^T**, "cm=^T%.%.". Terminals that use "%." need to be able to backspace the cursor (**le**) and to move the cursor up one line on the screen (**up**). This is necessary because it is not always safe to transmit **\n**, **^D**, and **\r**, as the system may change or discard them. (Programs using *termcap* must set terminal modes so that tabs are not expanded, so **\t** is safe to send. This turns out to be essential for the Ann Arbor 4080.)

A final example is the Lear Siegler ADM–3a, which offsets row and column by a blank character, thus "cm=\E=%+ %+ ".

Row or column absolute cursor addressing can be given as single parameter capabilities **ch** (horizontal position absolute) and **cv** (vertical position absolute). Sometimes these are shorter than the more general two-parameter sequence (as with the Hewlett-Packard 2645) and can be used in preference to **cm**. If there are parameterized local motions (*e.g.*, move *n* positions to the right) these can be given as **DO**, **LE**, **RI**, and **UP** with a single parameter indicating how many positions to move. These are primarily useful if the terminal does not have **cm**, such as the Tektronix 4025.

Cursor Motions

If the terminal has a fast way to home the cursor (to the very upper left corner of the screen), this can be given as **ho**. Similarly, a fast way of getting to the lower left-hand corner can be given as **ll**; this may involve going up with **up** from the home position, but a program should never do this itself (unless **ll** does), because it can make no assumption about the effect of moving up from the home position. Note that the home position is the same as cursor address (0,0): to the top left corner of the screen, not of memory. (Therefore, the "\EH" sequence on Hewlett-Packard terminals cannot be used for **ho**.)

Area Clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as **ce**. If the terminal can clear from the current position to the end of the display, this should be given as **cd**. **cd** must only be invoked from the first column of a line. (Therefore, it can be simulated by a request to delete a large number of lines, if a true **cd** is not available.)

Insert/Delete Line

If the terminal can open a new blank line before the line containing the cursor, this should be given as **al**; this must be invoked only from the first position of a line. The cursor must then appear at the left of the newly blank line. If the terminal can delete the line that the cursor is on, this should be given as **dl**; this must only be used from the first position on the line to be deleted. Versions of **al** and **dl** which take a single parameter and insert or delete that many lines can be given as **AL** and **DL**. If the terminal has a settable scrolling region (like the VT100), the command to set this can be described with the **cs** capability, which takes two parameters: the top and bottom lines of the scrolling region. The cursor position is, alas, undefined after using this command. It is possible to get the effect of insert or delete line using this command — the **sc** and **rc** (save and restore cursor) commands are also useful. Inserting lines at the top or bottom of the screen can also be done using **sr** or **sf** on many terminals without a true insert/delete line, and is often faster even on terminals with those features.

If the terminal has the ability to define a window as part of memory which all commands affect, it should be given as the parameterized string **wi**. The four parameters are the starting and ending lines in memory and the starting and ending columns in memory, in that order. (This *terminfo* capability is described for completeness. It is unlikely that any *termcap*-using program will support it.)

If the terminal can retain display memory above the screen, then the **da** capability should be given; if display memory can be retained below, then **db** should be given. These indicate that deleting a line or scrolling may bring non-blank lines up from below or that scrolling back with **sr** may bring down non-blank lines.

Insert/Delete Character

There are two basic kinds of intelligent terminals with respect to insert/delete character that can be described using *termcap*. The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept–100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated or expanded to two untyped blanks. You can determine the kind of terminal you have by clearing the screen then typing text separated by cursor motions. Type "abc   def" using local cursor motions (not spaces) between the "abc" and the "def". Then position the cursor before the "abc" and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the "abc" shifts over to the "def" which then move together around the end of the current line and onto the next as you insert, then you have the second type of terminal and should give the capability **in**, which stands for "insert null". While these are two logically separate attributes (one line *vs.* multi-line insert mode, and special treatment of untyped spaces), we have seen no terminals whose insert mode cannot be described with the single attribute.

Termcap can describe both terminals that have an insert mode and terminals that send a simple sequence to open a blank position on the current line. Give as **im** the sequence to get into insert mode. Give as **ei** the sequence to leave insert mode. Now give as **ic** any sequence that needs to be sent just before each character to be inserted. Most terminals with a true insert mode will not give **ic**; terminals that use a sequence to open a screen position should give it here. (If your terminal has both, insert mode is usually preferable to **ic**. Do not give both unless the terminal actually requires both to be used in combination.) If post-insert padding is needed, give this as a number of milliseconds in **ip** (a string option). Any other sequence that may need to be sent after insertion of a single character can also be given in **ip**. If your terminal needs to be placed into an 'insert mode' and needs a special code preceding each inserted character, then both **im**/**ei** and **ic** can be given, and both will be used. The **IC** capability, with one parameter *n*, will repeat the effects of **ic** *n* times.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (*e.g.*, if there is a tab after the insertion position). If your terminal allows motion while in insert mode, you can give the capability **mi** to speed up inserting in this case. Omitting **mi** will affect only speed. Some terminals (notably Datamedia's) must not have **mi** because of the way their insert mode works.

Finally, you can specify **dc** to delete a single character, **DC** with one parameter *n* to delete *n* characters, and delete mode by giving **dm** and **ed** to enter and exit delete mode (which is any mode the terminal needs to be placed in for **dc** to work).

Highlighting, Underlining, and Visible Bells

If your terminal has one or more kinds of display attributes, these can be represented in a number of different ways. You should choose one display form as *standout mode*, representing a good high-contrast, easy-on-the-eyes format for highlighting error messages and other attention getters. (If you have a choice, reverse video plus half-bright is good, or reverse video alone.) The sequences to enter and exit standout mode are given as **so** and **se**, respectively. If the code to change into or out of standout mode leaves one or even two blank spaces or garbage characters on the screen, as the TVI 912 and Teleray 1061 do, then **sg** should be given to tell how many characters are left.

Codes to begin underlining and end underlining can be given as **us** and **ue**, respectively. Underline mode change garbage is specified by **ug**, similar to **sg**. If the terminal has a code to underline the current character and move the cursor one position to the right, such as the Microterm Mime, this can be given as **uc**.

Other capabilities to enter various highlighting modes include **mb** (blinking), **md** (bold or extra bright), **mh** (dim or half-bright), **mk** (blanking or invisible text), **mp** (protected), **mr** (reverse video), **me** (turn off *all* attribute modes), **as** (enter alternate character set mode), and **ae** (exit alternate character set mode). Turning on any of these modes singly may or may not turn off other modes.

If there is a sequence to set arbitrary combinations of mode, this should be given as **sa** (set attributes), taking 9 parameters. Each parameter is either 0 or 1, as the corresponding attributes is on or off. The 9 parameters are, in order: standout, underline, reverse, blink, dim, bold, blank, protect, and alternate character set. Not all modes need be supported by **sa**, only those for which corresponding attribute commands exist. (It is unlikely that a *termcap*-using program will support this capability, which is defined for compatibility with *terminfo*.)

Terminals with the "magic cookie" glitches (**sg** and **ug**), rather than maintaining extra attribute bits for each character cell, instead deposit special "cookies", or "garbage characters", when they receive mode-setting sequences, which affect the display algorithm.

Some terminals, such as the Hewlett-Packard 2621, automatically leave standout mode when they move to a new line or when the cursor is addressed. Programs using standout mode should exit standout mode on such terminals before moving the cursor or sending a newline. On terminals where this is not a problem, the **ms** capability should be present to say that this overhead is unnecessary.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement), this can be given as **vb**; it must not move the cursor.

If the cursor needs to be made more visible than normal when it is not on the bottom line (to change, for example, a non-blinking underline into an easier-to-find block or blinking underline), give this sequence as **vs**. If there is a way to make the cursor completely invisible, give that as **vi**. The capability **ve**, which undoes the effects of both of these modes, should also be given.

If your terminal correctly displays underlined characters (with no special codes needed) even though it does not overstrike, then you should give the capability **ul**. If overstrikes are erasable with a blank, this should be indicated by giving **eo**.

Keypad

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local mode (this applies, for example, to the unshifted Hewlett-Packard 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as **ks** and **ke**. Otherwise the keypad is assumed to always transmit. The codes sent by the left-arrow, right-arrow, up-arrow, down-arrow, and home keys can be given as **kl**, **kr**, **ku**, **kd**, and **kh**, respectively. If there are function keys such as f0, f1, ..., f9, the codes they send can be given as **k0**, **k1**, **k9**. If these keys have labels other than the default f0 through f9, the labels can be given as **l0**, **l1**, **l9**. The codes transmitted by certain other special keys can be given: **kH** (home down), **kb** (backspace), **ka** (clear all tabs), **kt** (clear the tab stop in this column), **kC** (clear screen or erase), **kD** (delete character), **kL** (delete line), **kM** (exit insert mode), **kE** (clear to end of line), **kS** (clear to end of screen), **kI** (insert character or enter insert mode), **kA** (insert line), **kN** (next page), **kP** (previous page), **kF** (scroll forward/down), **kR** (scroll backward/up), and **kT** (set a tab stop in this column). In addition, if the keypad has a 3 by 3 array of keys including the four arrow keys, then the other five keys can be given as **K1**, **K2**, **K3**, **K4**, and **K5**. These keys are useful when the effects of a 3 by 3 directional pad are needed. The obsolete **ko** capability formerly used to describe "other" function keys has been completely supplanted by the above capabilities.

The **ma** entry is also used to indicate arrow keys on terminals that have single-character arrow keys. It is obsolete but still in use in version 2 of *vi* which must be run on some minicomputers due to memory limitations. This field is redundant with **kl**, **kr**, **ku**, **kd**, and **kh**. It consists of groups of two characters. In each group, the first character is what an arrow key sends, and the second character is the corresponding *vi* command. These commands are **h** for **kl**, **j** for **kd**, **k** for **ku**, **l** for **kr**, and **H** for **kh**. For example, the Mime would have "ma=^Hh^Kj^Zk^Xl" indicating arrow keys left (^H), down (^K), up (^Z), and right (^X). (There is no home key on the Mime.)

Tabs and Initialization

If the terminal needs to be in a special mode when running a program that uses these capabilities, the codes to enter and exit this mode can be given as **ti** and **te**. This arises, for example, from terminals like the Concept with more than one page of memory. If the terminal has only memory-relative cursor addressing and not screen-relative cursor addressing, a screen-sized window must be fixed into the display for cursor addressing to work properly. This is also used for the Tektronix 4025, where **ti** sets the command character to be the one used by *termcap*.

Other capabilities include **is**, an initialization string for the terminal, and **if**, the name of a file containing long initialization strings. These strings are expected to set the terminal into modes consistent with the rest of the *termcap* description. They are normally sent to the terminal by the *tset* program each time the user logs in. They will be printed in the following order: **is**; setting tabs using **ct** and **st**; and finally **if**. (*Terminfo* uses **i1-i2** instead of **is** and runs the program **iP** and prints **i3** after the other initializations.) A pair of sequences that does a harder reset from a totally unknown state can be analogously given as **rs** and **if**. These strings are output by the *reset* program, which is used when the terminal gets into a wedged state. (*Terminfo* uses **r1-r3** instead of **rs**.) Commands are normally placed in **rs** and **rf** only if they produce annoying effects on the screen and are not necessary when logging in. For example, the command to set the VT100 into 80-column mode would normally be part of **is**, but it causes an annoying glitch of the screen and is not normally needed since the terminal is usually already in 80-column mode.

If the terminal has hardware tabs, the command to advance to the next tab stop can be given as **ta** (usually ^I). A "backtab" command which moves leftward to the previous tab stop can be given as **bt**. By convention, if the terminal driver modes indicate that tab stops are being expanded by the computer rather than being sent to the terminal, programs should not use **ta** or **bt** even if they are present, since the user may not have the tab stops properly set. If the terminal has hardware tabs that are initially set every *n* positions when the terminal is powered up, then the numeric parameter **it** is given, showing the number of positions between tab stops. This is normally used by the *tset* command to determine whether to set the driver mode for hardware tab expansion, and whether to set the tab stops. If the terminal has tab stops that can be saved in nonvolatile memory, the *termcap* description can assume that they are properly set.

If there are commands to set and clear tab stops, they can be given as **ct** (clear all tab stops) and **st** (set a tab stop in the current column of every row). If a more complex sequence is needed to set the tabs than can be described by this, the sequence can be placed in **is** or **if**.

Delays

Certain capabilities control padding in the terminal driver. These are primarily needed by hardcopy terminals and are used by the *tset* program to set terminal driver modes appropriately. Delays embedded in the capabilities **cr**, **sf**, **le**, **ff**, and **ta** will cause the appropriate delay bits to be set in the terminal driver. If **pb** (padding baud rate) is given, these values can be ignored at baud rates below the value of **pb**. For 4.2BSD *tset*, the delays are given as numeric capabilities **dC**, **dN**, **dB**, **dF**, and **dT** instead.

Miscellaneous

If the terminal requires other than a NUL (zero) character as a pad, this can be given as **pc**. Only the first character of the **pc** string is used.

If the terminal has commands to save and restore the position of the cursor, give them as **sc** and **rc**.

If the terminal has an extra "status line" that is not normally used by software, this fact can be indicated. If the status line is viewed as an extra line below the bottom line, then the capability **hs** should be given. Special strings to go to a position in the status line and to return from the status line can be given as **ts** and **fs**. (**fs** must leave the cursor position in the same place that it was before **ts**. If necessary, the **sc** and **rc** strings can be included in **ts** and **fs** to get this effect.) The capability **ts** takes one parameter, which is the column number of the status line to which the cursor is to be moved. If escape sequences and other special commands such as tab work while in the status line, the flag **es** can be given. A string that turns off the status line (or otherwise erases its contents) should be given as **ds**. The status line is normally assumed to be the same width as the rest of the screen, *i.e.*, **co**. If the status line is a different width (possibly because the terminal does not allow an entire line to be loaded), then its width in columns can be indicated with the numeric parameter **ws**.

If the terminal can move up or down half a line, this can be indicated with **hu** (half-line up) and **hd** (half-line down). This is primarily useful for superscripts and subscripts on hardcopy terminals. If a hardcopy terminal can eject to the next page (form feed), give this as **ff** (usually **^L**).

If there is a command to repeat a given character a given number of times (to save time transmitting a large number of identical characters), this can be indicated with the parameterized string **rp**. The first parameter is the character to be repeated and the second is the number of times to repeat it. (This is a *terminfo* feature that is unlikely to be supported by a program that uses *termcap*.)

If the terminal has a settable command character, such as the Tektronix 4025, this can be indicated with **CC**. A prototype command character is chosen which is used in all capabilities. This character is given in the **CC** capability to identify it. The following convention is supported on some UNIX systems: The environment is to be searched for a **CC** variable, and if found, all occurrences of the prototype character are replaced by the character in the environment variable. This use of the **CC** environment variable is a very bad idea, as it conflicts with *make*(1).

Terminal descriptions that do not represent a specific kind of known terminal, such as *switch*, *dialup*, *patch*, and *network*, should include the **gn** (generic) capability so that programs can complain that they do not know how to talk to the terminal. (This capability does not apply to *virtual* terminal descriptions for which the escape sequences are known.)

If the terminal uses xoff/xon (DC3/DC1) handshaking for flow control, give **xo**. Padding information should still be included so that routines can make better decisions about costs, but actual pad characters will not be transmitted.

If the terminal has a "meta key" which acts as a shift key, setting the 8th bit of any character transmitted, then this fact can be indicated with **km**. Otherwise, software will assume that the 8th bit is parity and it will usually be cleared. If strings exist to turn this "meta mode" on and off, they can be given as **mm** and **mo**.

If the terminal has more lines of memory than will fit on the screen at once, the number of lines of memory can be indicated with **lm**. An explicit value of 0 indicates that the number of lines is not fixed, but that there is still more memory than fits on the screen.

If the terminal is one of those supported by the UNIX system virtual terminal protocol, the terminal number can be given as **vt**.

Media copy strings which control an auxiliary printer connected to the terminal can be given as **ps**: print the contents of the screen; **pf**: turn off the printer; and **po**: turn on the printer. When the printer is on, all text sent to the terminal will be sent to the printer. It is undefined whether the text is also displayed on the terminal screen when the printer is on. A variation **pO** takes one parameter and leaves the printer on for as many characters as the value of the parameter, then turns the printer off. The parameter should not exceed 255. All text, including **pf**, is transparently passed to the printer while **pO** is in effect.

Strings to program function keys can be given as **pk**, **pl**, and **px**. Each of these strings takes two parameters: the function key number to program (from 0 to 9) and the string to program it with. Function key numbers out of this range may program undefined keys in a terminal-dependent manner. The differences among the capabilities are that **pk** causes pressing the given key to be the same as the user typing the given string; **pl** causes the string to be executed by the terminal in local mode; and **px** causes the string to be transmitted to the computer. Unfortunately, due to lack of a definition for string parameters in *termcap*, only *terminfo* supports these capabilities.

Glitches and Braindamage

Hazeltine terminals, which do not allow '' characters to be displayed, should indicate **hz**.

The **nc** capability, now obsolete, formerly indicated Datamedia terminals, which echo **\r \n** for carriage return then ignore a following linefeed.

Terminals that ignore a linefeed immediately after an **am** wrap, such as the Concept, should indicate **xn**.

If **ce** is required to get rid of standout (instead of merely writing normal text on top of it), **xs** should be given.

Teleray terminals, where tabs turn all characters moved over to blanks, should indicate **xt** (destructive tabs). This glitch is also taken to mean that it is not possible to position the cursor on top of a "magic cookie", and that to erase standout mode it is necessary to use delete and insert line.

The Beehive Superbee, which is unable to correctly transmit the ESC or ^C characters, has **xb**, indicating that the "f1" key is used for ESC and "f2" for ^C. (Only certain Superbees have this problem, depending on the ROM.)

Other specific terminal problems may be corrected by adding more capabilities of the form **x***x*.

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **tc** can be given with the name of the similar terminal. This capability must be *last*, and the combined length of the entries must not exceed 1024. The capabilities given before **tc** override those in the terminal type invoked by **tc**. A capability can be canceled by placing **xx@** to the left of the **tc** invocation, where *xx* is the capability. For example, the entry

```
hn | 2621–nl:ks@:ke@:tc=2621:
```

defines a "2621–nl" that does not have the **ks** or **ke** capabilities, hence does not turn on the function key labels when in visual mode. This is useful for different modes for a terminal, or for different user preferences.

AUTHOR

William Joy
Mark Horton added underlining and keypad support

FILES

/etc/termcap file containing terminal descriptions

SEE ALSO

ex(1), more(1), tset(1), ul(1), vi(1), curses(3X), printf(3S), term(7).

CAVEATS AND BUGS

Note: *termcap* was replaced by *terminfo* in UNIX System V Release 2.0. The transition will be relatively painless if capabilities flagged as "obsolete" are avoided.

Lines and columns are now stored by the kernel as well as in the termcap entry. Most programs now use the kernel information primarily; the information in this file is used only if the kernel does not have any information.

Vi allows only 256 characters for string capabilities, and the routines in *termlib*(3) do not check for overflow of this buffer. The total length of a single entry (excluding only escaped newlines) may not exceed 1024.

Not all programs support all entries.

NAME

types – primitive system data types

SYNOPSIS

#include <sys/types.h>

DESCRIPTION

The data types defined in the include file are used in UNIX system code; some data of these types are accessible to user code:

```
typedef struct { int r[1]; } *     physadr;
typedef long            daddr_t;      /* <disk address> type */
typedef char *          caddr_t;      /* ?<core address> type */
typedef unsigned char * ucaddr_t;     /* Unsigned char addr */
typedef unsigned char   unchar;
typedef unsigned char   u_char;
typedef unsigned short  ushort;
typedef unsigned short  u_short;
typedef unsigned int    uint;
typedef unsigned int    u_int;
typedef unsigned long   ulong;
typedef unsigned long   u_long;
typedef ushort          ino_t;        /* <inode> type */
typedef short           cnt_t;        /* ?<count> type */
typedef long            time_t;       /* <time> type */
typedef int             label_t[12];
typedef short           dev_t;        /* <old device number> type */
typedef long            off_t;        /* ?<offset> type */
typedef unsigned long   paddr_t;      /* <physical address> type */
typedef int             key_t;        /* IPC key type */
typedef unsigned char   use_t;        /* use count for swap. */
typedef short           sysid_t;
typedef short           index_t;
typedef short           lock_t;       /* lock work for busy wait */
typedef unsigned int    size_t;       /* len param for string funcs */
```

The form *daddr_t* is used for disk addresses except in an i-node on disk, see *fs*(4). Times are encoded in seconds since 00:00:00 GMT, January 1, 1970. The major and minor parts of a device code specify kind and unit number of a device and are installation-dependent. Offsets are measured in bytes from the beginning of a file. The *label_t* variables are used to save the processor state while another process is running.

SEE ALSO

fs(4).

NAME

values – manifest constants for Stardent 1500/3000 architecture

DESCRIPTION

This file contains a set of manifest constants defined for the Stardent 1500/3000 architecture.

Integers are represented in two's complement binary, with the sign represented by the high-order bit.

BITS(type)	The number of bits in a specified type (e.g., int).
HIBITS	The value of a short integer with only the high-order bit set (0x8000).
HIBITL	The value of a long integer with only the high-order bit set (0x80000000).
HIBITI	The value of a regular integer with only the high-order bit set (same as HIBITL).
MAXSHORT	The maximum value of a signed short integer (0x7FFF = 32767).
MAXLONG	The maximum value of a signed long integer (0x7FFFFFFF = 2147483647).
MAXINT	The maximum value of a signed regular integer (same as MAXLONG).
MAXFLOAT	The maximum value of a single-precision floating-point number.
LN_MAXFLOAT	The natural log of MAXFLOAT.
MAXDOUBLE	The maximum value of a double-precision floating-point number.
LN_MAXDOUBLE	The natural log of MAXDOUBLE.
MINFLOAT	The minimum positive value of a single-precision floating-point number.
LN_MINFLOAT	The natural log of MINFLOAT.
MINDOUBLE	The minimum positive value of a double-precision floating-point number.
LN_MINDOUBLE	The natural log of MINDOUBLE.
FSIGNIF	The number of significant bits in the mantissa of a single-precision floating-point number.
DSIGNIF	The number of significant bits in the mantissa of a double-precision floating-point number.
_EXPBASE	The exponent base (2).
_IEEE	Nonzero if IEEE representation is used (1).

_DEXPLEN	The number of bits for the exponent of a double (11).
_FEXPLEN	The number of bits for the exponent of a float (8).
_HIDDENBIT	Nonzero if the most significant bit of the mantissa is implicit (1).
DMAXEXP	The maximum exponent of a double (as returned by frexp()).
FMAXEXP	The maximum exponent of a float (as returned by frexp()).
DMINEXP	The minimum exponent of a double (as returned by frexp()).
FMINEXP	The minimum exponent of a float (as returned by frexp())
DMAXPOWTWO	The largest power of two exactly representable as a double.
FMAXPOWTWO	The largest power of two exactly representable as a float.

NAME

varargs – get variable argument lists

SYNOPSIS

# include <varargs.h>

va_alist

va_dcl

void va_start(pvar)
va_list pvar;

type va_arg(pvar,type)
va_list pvar;

type *va_argp(pvar,type)
va_list pvar;

DESCRIPTION

This set of macros allows portable procedures to be written that accept variable argument lits. Routines that have variable argument lists [such as printf(3S)] but do not use varargs are inherently nonportable, as different machines use different argument-passing conventions.

va_alist is used as the parameter list in a function header.

va_dcl is a declaration for *va_alist.*

va_list is a type defined for the variable used to traverse the list.

va_start is called to initialize pvar to the beginning of the list.

va_arg returns the next argument in the list pointed to by *pvar. type* is the type the argument is expected to be. Different types can be mixed, but it is up to the routine to know what type of argument is expected, as it cannot be determined at runtime.

va_end is used to clean up.

Multiple traversals, each bracketed by *va_start ... va_end,* are possible.

In addition to the above, portable features, there are several features that are specific to the Stardent 1500/3000:

> *va_argp(pvar,type)* is used to return a pointer to the next argument, stored in memory.
>
> *va_doublep(pvar)* is used to get pointers to double precision values (see below).

EXAMPLE

This example shows a possible portable implementation of *execl(2).*

```
#include <varargs.h>
#define MAXARGS 100

/*      execl is called by
        execl( file, arg1, arg2, ..., (char *)0 );
*/

execl(va_dcl va_alist)
{
        va_list ap;

        char *file;
        char *args[MAXARGS];
        int argno = 0;

        va_start(ap);
        file = va_arg(ap,char *);
        while((args[argno++] = va_arg(ap,char *)) != (char *)0)
                ;
        va_end(ap);
        return execv(file,args);
}
```

On the Stardent 1500/3000, the first four integer/pointer arguments and the first (two doubles on P3; four doubles on P2) four float/double function arguments are passed in registers. This complicates the implementation of *varargs*. In particular, a function that uses *varargs* will always assume that it might have floating point arguments passed to it, and will save the floating point registers.

Under some circumstances (such as the *execl* example, above) we know that the defined function can never be passed a floating-point number. In this case, the pragma

```
#pragma NO_FLOAT
```

at the head of the file will suppress the use of floating point. Note that this pragma affects the entire file in which it appears.

Under even more obscure circumstances (e.g., *printf*), it is desirable to avoid using floating point operations until you are certain that a floating point computation is being done. This can be done by postponing the storing of the floating point registers until you are certain that floating point will be used (e.g., when a % *f* format is encountered). In this case, the NO_FLOAT pragma can be used in the routine where the *varargs* is seen; this routine must then call another routine (compiled without NO_FLOAT), passing it the *va_list* variable, and this routine can then use *va_doublep* to obtain the double precision values.

This manual page is too small to contain all the details.

NAME

vmath – include definitions for vector math routines

SYNOPSIS

```
#include<vmath.h>
```

DESCRIPTION

This file is an alternative to *math.h*. In addition to all the declarations found in *math.h*, this file contains pragmas to the C compiler indicating those routines which have vector versions in *libm*, so that including this file will cause these routines to vectorize. Without these declarations, the C compiler assumes that references to math intrinsic functions may be belong to the programmer and not those defined in *libm*, and hence will not vectorize them. Use of this include file almost always requires the inclusion of *libm* on the link line.

SEE ALSO

intro(3), math(5), values(5)

NAME

intro – introduction to special files

DESCRIPTION

This section describes various special files that refer to specific hardware peripherals, and UNIX system device drivers. STREAMS [see *intro*(2)] software drivers, modules and the STREAMS-generic set of *ioctl*(2) system calls are also described.

For hardware related files, the names of the entries are generally derived from names for the hardware, as opposed to the names of the special files themselves. Characteristics of both the hardware device and the corresponding UNIX system device driver are discussed where applicable.

Disk device file names are in the following format:

/dev/{r}dsk/c#d#s#
/dev/{r}dsk/k#d#s# (for the second I/O)

where **r** indicates a raw interface to the disk, the **c#** indicates the controller number, **d#** indicates the device attached to the controller and **s#** indicates the section number of the partitioned device.

NAME

arp – Address Resolution Protocol

SYNOPSIS

pseudo-device ether

DESCRIPTION

ARP is a protocol used to dynamically map between DARPA Internet and 10Mb/s Ethernet addresses. It is used by all the 10Mb/s Ethernet interface drivers. It is not specific to Internet protocols or to 10Mb/s Ethernet, but this implementation currently supports only that combination.

ARP caches Internet-Ethernet address mappings. When an interface requests a mapping for an address not in the cache, ARP queues the message which requires the mapping and broadcasts a message on the associated network requesting the address mapping. If a response is provided, the new mapping is cached and any pending message is transmitted. ARP will queue at most one packet while waiting for a mapping request to be responded to; only the most recently "transmitted" packet is kept.

To facilitate communications with systems which do not use ARP, *ioctl*s are provided to enter and delete entries in the Internet-to-Ethernet tables. Usage:

```
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <net/if.h>
struct arpreq arpreq;

ioctl(s, SIOCSARP, (caddr_t)&arpreq);
ioctl(s, SIOCGARP, (caddr_t)&arpreq);
ioctl(s, SIOCDARP, (caddr_t)&arpreq);
```

Each ioctl takes the same structure as an argument. SIOCSARP sets an ARP entry, SIOCGARP gets an ARP entry, and SIOCDARP deletes an ARP entry. These ioctls may be applied to any socket descriptor *s*, but only by the super-user. The *arpreq* structure contains:

```
/* ARP ioctl request */
struct arpreq {
        struct sockaddr      arp_pa; /* protocol address */
        struct sockaddr      arp_ha; /* hardware address */
        int                  arp_flags; /* flags */
};
/*  arp_flags field values */
#define ATF_COM              0x02   /* completed entry (arp_ha valid) */
#define  ATF_PERM            0x04   /* permanent entry */
#define  ATF_PUBL            0x08   /* publish (respond for other host) */
#define  ATF_USETRAILERS     0x10   /* send trailer packets to host */
```

The address family for the *arp_pa* sockaddr must be AF_INET; for the *arp_ha* sockaddr it must be AF_UNSPEC. The only flag bits which may be written are ATF_PERM, ATF_PUBL and ATF_USETRAILERS. ATF_PERM causes the entry to be permanent if the ioctl call succeeds. The peculiar nature of the ARP tables may cause the ioctl to fail if more than 8 (permanent) Internet host addresses hash to the same slot. ATF_PUBL specifies that the ARP code should respond to ARP requests for the indicated host coming from other machines. This allows a host to act as an "ARP server," which may be useful in convincing an ARP-only machine to talk to a non-ARP machine.

ARP is also used to negotiate the use of trailer IP encapsulations; trailers are an alternate encapsulation used to allow efficient packet alignment for large packets despite variable-sized headers. Hosts which wish to receive trailer encapsulations so indicate by sending gratuitous ARP translation replies along with replies to IP requests; they are also sent in reply to IP translation replies. The negotiation is thus fully symmetrical, in that either or both hosts may request trailers. The ATF_USETRAILERS flag is used to record the receipt of such a reply, and enables the transmission of trailer packets to that host.

ARP watches passively for hosts impersonating the local host (i.e. a host which responds to an ARP mapping request for the local host's address).

DIAGNOSTICS

duplicate IP address!! sent from ethernet address: %x:%x:%x:%x:%x:%x. ARP has discovered another host on the local network which responds to mapping requests for its own Internet address.

SEE ALSO

ec(4), de(4), il(4), inet(4F), arp(8C), ifconfig(8C)
"An Ethernet Address Resolution Protocol," RFC826, Dave Plummer, Network Information Center, SRI.
"Trailer Encapsulations," RFC893, S.J. Leffler and M.J. Karels, Network Information Center, SRI.

BUGS

ARP packets on the Ethernet use only 42 bytes of data; however, the smallest legal Ethernet packet is 60 bytes (not including CRC). Some systems may not enforce the minimum packet size, others will.

NAME

/dev/centrnix – Centronix Parallel Port Driver

DESCRIPTION

The Centronix driver provides an interface to a user program. The driver can be accessed via the *open(2)*,write(2), and *close(2)* system calls. Data is passed transparently to the device connected to the parallel port, meaning that no translation of any sort is performed on the data.

Because the Tektronix 4693D color plotter uses the signals differently from other Centronix printers, the driver provides two different classes to the user program. If the driver is opened with a minor device number with the 0x10 (hex 10) bit on, it is opened in Tektronix mode. Otherwise it is opened in normal mode. If a Tektronix 4693D is connected to the parallel port, then the Tektronix mode must be used.

NAME

clone – open any minor device on a STREAMS driver

DESCRIPTION

clone is a STREAMS software driver that finds and opens an unused minor device on another STREAMS driver. The minor device passed to *clone* during the open is interpreted as the major device number of another STREAMS driver for which an unused minor device is to be obtained. Each such open results in a separate *stream* to a previously unused minor device.

The *clone* driver consists solely of an open function. This open function performs all of the necessary work so that subsequent system calls (including *close(2)*) require no further involvement of *clone*.

clone will generate an ENXIO error, without opening the device, if the minor device number provided does not correspond to a valid major device, or if the driver indicated is not a STREAMS driver.

CAVEATS

Multiple opens of the same minor device cannot be done through the *clone* interface. Executing *stat(2)* on the file system node for a cloned device yields a different result from executing *fstat(2)* using a file descriptor obtained from opening the node.

NAME

console – console interface

DESCRIPTION

The console provides the operator interface to the console.

The file */dev/console* is the system console, and refers to an asynchronous serial data line originating from the system board. This special file implements the features described in *termio*(7).

FILE

/dev/console

SEE ALSO

termio(7).

NAME

dlb – dial box

DESCRIPTION

dlb controls the Stardent 1500/3000 digit-8 dial box which is made by Honeywell. *dlb* is a STREAMS module, pushed on top of an RS-232 line driver.

The user can not send data to the dial box, but can receive data. The data are encapsulated in the structure **gin** format as defined in **<machine/gin.h>** before being sent toward the user.

```
struct gin {
          unsigned long         gin_msec;
          unsigned long         gin_mux_id;
          unsigned short        gin_id;
          char                  gin_dev;
          char                  gin_dim;
          long                  gin_buttons;
          short                 gin_data[NDIM_GIN];
};
```

gin_id is GID_DIALS. *gin_dev* is GDEV_REL|GDEV_SUBDEV. *gin_dim* is two (2). *gin_buttons* is zero (0). The first data (*gin_data*[0]) indicates which dial (value from 0 to 7). The second data (*gin_data*[1]) specifies the relative movement (delta) for that dial (value 0 to 255).

For the standard configuration, the dial box is connected to the serial port "d" (that is, tty line 3). Thus, **/dev/dials** is linked to **/dev/tty3**. The user conventionally opens **/dev/dials**, (via *open*(2)), pops off the tty line discipline, (via I_POP ioctl), and pushes the **dlb**(7) module, (via I_PUSH ioctl).

COMMAND FUNCTIONS

dlb(7) recognizes the following *ioctl* commands. The argument (the M_DATA block following the M_IOCTL) has the format of the structure **ginioctl**, defined in **<machine/gin.h>**:

```
struct ginioctl {
          int       g_cmd;
          int       g_id;/* muxid under gin*/
          int       g_arg1;
};
```

G_MSTHRESH	Set the threshold (delta) value which must be exceeded before the delta value is reported to the user. All eight (8) dials are set to the same value, a 32-bit integer. The default threshold is 1.
G_MGTHRESH	Retrieve the threshold value.
G_RATE	Set the rate at which data packets are transmitted to the user. *g_arg1* specifies the rate in samples per second. Default is 20 samples per second.

FILES

/dev/dials

SEE ALSO

streamio(7), gin(7)
DIGIT Technical Specification, Honeywell.

NAME

dsk – SCSI disk interface

DESCRIPTION

All disk devices on the SCSI bus are accessed by the files in */dev/dsk* for block mode access or */dev/rdsk* for character mode access. Each device name is of the form cXdYsZ where *X* is the SCSI controller (0 for A, 1 for B), *Y* is the device target number and Z is the disk partition number, *vh* for the disk header or *vol* for the entire disk volume.

FILES

/dev/{r}dsk/c?d?s?
/dev/{r}dsk/c?d?vh
/dev/{r}dsk/c?d?vol

SEE ALSO

mt(7)

NAME

et – generic ethernet driver

DESCRIPTION

et is a STREAMS general ethernet driver. It is general in the sense that it can support many different hardware devices, such as the AMD's LANCE device or Interphase EAGLE board.

COMMAND FUNCTIONS

et recognizes the following *ioctl* commands.

ET_GFACADDR	Get manufacturer assigned ethernet address.
ET_SADDRESS	Set ethernet address to be used.
ET_GADDRESS	Get ethernet address currently being used.
ET_SETTYPE	Set the ethernet type field.
ET_GETTYPE	Get the ethernet type currently being used.
ET_MCASTOP	Multicast operations. This *ioctl* takes the following arguments:

```
struct et_mcast {
        int        em_cmd;              /* command */
        int        em_naddr;            /* number of addresses */
        eaddr_t    em_addr[MAXMCAST];
};
```

where *em_cmd* can be one of the following values:

EM_SET	Set multicast addresses. All existing multicast addresses are cleared first.
EM_CLR	Clear all multicast addresses.
EM_ADD	Add multicast addresses to the current set of addresses.
EM_DEL	Delete multicast addresses.
EM_GET	Get the multicast addresses currently being used.

ET_SETMODE Direct the generic ethernet module to do, with the following arguments:

ER_ETHERRAW	Specify a raw ethernet I/O.
ER_ETCTYPE	On raw ethernet output, the ethernet type field should use the value stored in the driver (its value was assigned by a previous ET_SETF *ioctl*).
ER_ETCSRC	On raw ethernet output, the source ethernet address field should use the value associated with this device.
ER_ETALLTYPE	On input, accept packets with all ethernet types.

ET_GETMODE Get the mode of the generic ethernet module.

ET_SETLMODE Set the mode of the lower level hardware device to:

ETC_PRM	Enable promiscuous receiving mode.
ETC_INTL	Enable internal loopback.
ETC_DRTY	Disable transmit retry.
ETC_COLL	Force collision on next transmit.

ETC_DTCR	Disable transmit CRC generation.
ETC_DTX	Disable transmitter.
ETC_DRX	Disable receiver.

ET_GETLMODE Get the mode of the lower level hardware device.

Only root is allowed to open an ethernet device or issued *ioctl* commands. Note that *ioctl* operations (e.g. promiscuous mode or multicast operations) may not be supported by all lower level devices. Modes of the generic ethernet module and the lower level devices are initialized to zero (0).

FILES

/dev/etc0 – et clone device for controller 0
/dev/etc1 – et clone device for controller 1
/dev/etc2 – et clone device for controller 2
/dev/etm0 – et control channel for controller 0
/dev/etm1 – et control channel for controller 1
/dev/etm2 – et control channel for controller 2

SEE ALSO

streamio(7), clone(7), et(7)

NAME

fcast – Dupont 4CAST Digital Color Printer Driver

DESCRIPTION

The fcast printer driver provides an interface to a user program through open(2), write(2), ioctl(2) and close(2). The printer accepts color plane interleaved image data in the order of Yellow, Magenta, Cyan and Black (YMCK). The driver requires the user program to send a complete color plane for each print (write) command. To print a full color image, the user must issue four separate write calls.

A print format defines various attributes of the printing image, such as, upper left x, y coordinates, image size in width and length, image depth (bit per pixel) and paper size. The Look Up Tables (four sections, one for each color) are used for pixel intensity transformation while the image is being printed.

FUNCTIONS

open Open the printer device for writing.

```
#include <machine/4cast.h>
int open (/dev/fcast, O_WRONLY)
```

Only one user can open the device at one time. The device must be opened as WRITE ONLY. The driver loads the print format and the Look Up Tables defined by system initialization or by previous *ioctl* calls. The color code is set to Yellow. The alternate print format, Look Up Tables or color code can be defined through *ioctl* calls after the device is opened (see description under *ioctl*). On failure, *errno* is set to:

[EBUSY]	The device is busy.
[EIO]	Physical printer error. Use FCAST_STATUS in *ioctl* to get more error information.

write Send one image plane to the printer.

```
int write (fd, buf, nbytes)
int fd;
char *buf;
unsigned long nbytes;
```

The image data stored in *buf* should be one of the four color planes (YMCK). The parameter *nbytes* is equal to number of pixels per line times total number of lines to print. *nbytes* must be less than the width times the length defined in the last *Print Format.* The driver advances the color code to the next color in sequence after write completes. On failure, *errno* is set to:

[EINVAL]	*nbytes* exceeds the defined format.
[EIO]	Physical printer error. Use FCAST_STATUS in *ioctl* to get more error information.

ioctl Miscellaneous printer control commands.

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

Commands are described as follow:

FCAST_SETLUT Set the color Look Up Tables. The Look Up Tables are set up as a one to one mapping upon system initialization. The structure of the Look Up Tables are defined in <machine/4cast.h>:

```
typedef struct {
        unsigned long colorcode:3;
        unsigned long bitsppixel:5;
        unsigned long transferlen:24;
        unsigned char data[1 << FCAST_BPPIXEL];
} LUTRequest;

        colorcode = FCAST_YELLOW;
        bitsppixel = FCAST_BPPIXEL;
        transferlen = FCAST_MAX_LUTSIZE;
        data = look up table for yellow;

        colorcode = FCAST_MAGENTA;
        bitsppixel = FCAST_BPPIXEL;
        transferlen = FCAST_MAX_LUTSIZE;
        data = look up table for magenta;

        colorcode = FCAST_CYAN;
        bitsppixel = FCAST_BPPIXEL;
        transferlen = FCAST_MAX_LUTSIZE;
        data = look up table for cyan;

        colorcode = FCAST_BLACK;
        bitsppixel = FCAST_BPPIXEL;
        transferlen = FCAST_MAX_LUTSIZE;
        data = look up table for black;
```

It is optional to load either single section or multiple sections with one *ioctl* call. On failure *errno* is set to:

[EINVAL] Invalid Look Up Table parameters.

[EFAULT] Invalid user address.

FCAST_GETLUT Get the color Look Up Tables. On failure *errno* is set to:

[EFAULT] Invalid user address.

FCAST_SETFORMAT Set the print format. The Print Format is set up upon system initialization with the default values shown as follow:

```
typedef struct {
        unsigned short :8;
        unsigned short papersize :8;
        unsigned short :16;
        unsigned short uleft_X;
        unsigned short uleft_Y;
        unsigned short width;
        unsigned short length;
        unsigned short :8;
        unsigned short csf:1;
        unsigned short :7;
        unsigned short :3;
        unsigned short bitppixel:5;
        unsigned short :8;
        unsigned short :16;
} PrintFormat;

        cs = 0;
        uleft_X = 0;
        uleft_Y = 0;
        papersize = FCAST_FREE;
        width = FCAST_WIDTH;
        length = FCAST_MAXLEN;
        bitppixel = FCAST_BPPIXEL;
```

On failure *errno* is set to:

[EINVAL] Invalid print format parameter.

[EFAULT] Invalid user address.

FCAST_GETFORMAT
Get the print format. On failure *errno* is set to:

[EFAULT] Invalid user address.

FCAST_SETCOLOR Set the next color to print. The ink ribbon will be wasted if this function is used to set the color other than the standard color sequence YMCK. On failure *errno* is set to:

[EINVAL] Invalid color code.

[EFAULT] Invalid user address.

FCAST_GETCOLOR Get the next color to print. On failure *errno* is set to:

[EFAULT] Invalid user address.

FCAST_REZERO Rezero printer unit and eject paper. This command is also issued when the device is closed. On failure *errno* is set to:

[EIO] Printer command REZERO failed.

FCAST_STATUS Get status from last command. The structure of the status data is defined in <machine/4cast.h>:

```
typedef struct {
        intsense_key;
        intfcast_key;
} fcast_status;
```

On failure *errno* is set to:

[EFAULT] Invalid user address.

close Close the printer device.

```
int close (fd)
int fd;
```

The driver performs a *Rezero* unit command to eject the paper and reset the printer on close.

FILES

/dev/fcast

BUGS

4CAST printer does not disconnect from the SCSI bus, and may hold the bus up to 45 seconds. Due to the I/O space limit, the maximum number of bytes to print must be less than 16 megabytes.

NAME

gin – graphics input multiplexor

DESCRIPTION

gin is a STREAMS multiplexor. It provides a graphics input interface between the raw input devices and the user level programs.

The input devices are keyboard, mouse, tablet, and dial box. The *gin* multiplexor translates data from hardware specific format to the common format, defined in **<machine/gin.h>**:

```
struct gin {
        unsigned long       gin_msec;
        unsigned long       gin_mux_id;
        unsigned short      gin_id;
        char                gin_dev;
        char                gin_dim;
        long                gin_buttons;
        short               gin_data[NDIM_GIN];
};
```

gin_mux_id is filled in by the multiplexor to identify each lower input stream. The value is the one obtained from the I_LINK *ioctl*(2) when the lower device is linked under the multiplexor. All other values in the *gin* structure are filled in by the individual lower driver. *gin_msec* is the time stamp. *gin_id* identifies the type of the lower device. *gin_dev* provides more information (such as sub-device). Both *gin_id* and *gin_dev* are defined in **<machine/gin.h>**. *gin_dim* indicates the number of dimensions. *gin_buttons* specifies the button (for mouse) positions. *gin_data* contains up to 8 16-bit data.

The user programs can only *read*(2), but not *write*(2), the *gin* device. The *gin* device only recognizes the generic STREAMS I_LINK and I_UNLINK ioctl. All other ioctls use the generic I_STR ioctl, with a common data format:

```
struct ginioctl {
        int     g_cmd;
        int     g_id;
        int     g_arg1;
};
```

The ioctls are passed to the designated (indexed by *g_id*) lower stream for processing. To support multiple graphics displays, a *gin* device is provided for each graphics display.

FILES

/dev/gin – *gin* clone device
/dev/gin0 – *gin* clone device for display 0
/dev/gin1 – *gin* clone device for display 1

SEE ALSO

streamio(7)

NAME

ip – internet protocol (IP) multiplexor

DESCRIPTION

ip is a (N by M) STREAMS multiplexor. The upstream side (N) is clonable (see *clone*(7)) and normally linked under *tcp*(7), *udp*(2), and *raw*(7). The downstream side is linked to various network interfaces, e.g. the ethernet interface *et*(7). *ip* implements the IP protocol as well as the Internet Control Message Protocol (ICMP).

COMMAND FUNCTIONS

ip recognizes the following *ioctl* commands.

I_LINK — Link another stream under *ip*. I_LINK can only happen at channel 0 (the *ip* control channel). Error codes are:

[EPERM]	The user does not have the "root" privilege.
[ENOSPC]	Too many hardware interfaces have been linked under. Resources are exhausted. Currently the limit is set to 10.

I_UNLINK — Unlink the *ip* stream. I_UNLINK can only happen at channel 0 (the *ip* control channel). Error codes are:

[EPERM]	The user does not have the "root" privilege.

IPIOC_SETPROTO — Set the protocol number for input multiplexing. For the assigned number for a particular protocol, see **<netinet/in.h>**.
When network data arrive at the *ip*(7), the protocol type (in the IP header) is used for input multiplexing. If a match is found, data are sent upwardly to the matched stream. If no match exists and the *raw*(7) multiplexor exists, data are then sent to *raw*(7). Error codes are:

[EADDRINUSE]	There is another *ip* clone already handling this protocol.

FILES

/dev/ip	ip clone device
/dev/ip0	ip control channel

SEE ALSO

streamio(7), clone(7), raw(7), udp(7), tcp(7), et(7)

NAME

kbd – keyboard driver

DESCRIPTION

kbd interfaces to the synchronous serial keyboard (8042). This driver is a STREAMS driver; thus it can be linked (via I_LINK ioctl) under the *gin* multiplexor. *kbd* can operate in either ASCII or non-ASCII mode. In the former, each key stroke is represented by a 7-bit ASCII code. For example, pressing and releasing "a" generates the number 0x61. In the latter case, each key stroke is represented by the "raw" scan code from the 8042. For example, pressing and releasing "a" gets 0x1c (scan code for "a"), 0xf0, and 0x1c (break code for "a"). The scan codes can be found in **<machine/kb.h>**, or in the IBM PC manual.

When the system is booted up, the console driver (*/dev/console*) internally opens this keyboard driver and sets its mode to ASCII. Thus keyboard input can be read from the console driver. Later the user (e.g. X11 server) can explicitly open */dev/kbd* and link it under the *gin*(7) multiplexor. Once this is done, the keyboard is in non-ASCII mode and the data is encapsulated in the form of the structure of *gin*, which is defined in **<machine/gin.h>**.

COMMANDS FUNCTIONS

kbd recognizes the following ioctl commands. They have to be encoded in I_STR format using the *ginioctl* structure. (See *gin(7)*.)

G_ENABLE Enable the keyboard device.

G_DISABLE Disable the keyboard device.

G_KASCII Set to ASCII mode. This is the default mode when the system is booted up.

G_KGIN Set to non-ASCII mode. This is the default mode when the *kbd* is explicitly opened.

G_KTYPE Set the typematic rate and delay. The typematic rate are coded in bits 4, 3, 2, 1, and 0. Bits 6 and 5 are the delay parameter. Bit 7 (the most significant bit) is always 0. The delay is equal to 1 plus the binary value of bits 6 and 5 multiplied by 250 milliseconds ± 20%. The period (interval from one typematic output to the next) can be calculated from:

period = (8 + 'bits 2, 1, 0') * (2 ** 'bits 4, 3') * 0.00417

The typematic rate (scan code per second) is 1/period.

The default values for the typematic rate is 10 characters per second and the delay is 500 milliseconds (± 20%).

G_KWRITE Write the data byte to the keyboard. This is used to turn on/off the LED on the keyboard. Writing a "1" turn the LED on. Note that only the three least significant bits are meaningful: bit 0 for scroll lock, bit 1 for num lock, and bit 2 for Caps lock.

G_RESET Re-enable the keyboard device and set the parameters to the default value.

FILES

/dev/kbd – clone device
/dev/kbd0 – keyboard for display 0
/dev/kbd1 – keyboard for display 1

SEE ALSO

streamio(7), gin(7), console(7)
IBM PC AT Technical Reference, IBM Corporation.

NAME

klog – kernel console message logging device

DESCRIPTION

klog is a special file to which the kernel sends all internally generated messages destined for the system console. It is intended for use solely by the syslog daemon so all kernel-generated console traffic can be logged to a disk file.

This special file allows *read*(2), but not *write*(2) system calls. The first read after an open is special. The kernel remembers all messages logged to the console from the time of the last system boot until *klog* is opened. At the time of the open, all of these messages are put onto the *klog* upstream queue. The first reads to *klog* are satisfied from this queue, until all queued messages have been read. From that time forward all reads will block until the kernel generates a new console message.

This special device file is a streams driver, so all stream operations are allowed. For instance, poll(2) will work with this special device file.

FILES

/dev/klog

SEE ALSO

poll(2), syslog(3), streamio(7), syslogd(8)

NAME

mem, kmem – core memory

DESCRIPTION

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

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

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

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

The per-process data for the current process begins at 0x80880000.

FILES

/dev/mem
/dev/kmem

WARNING

Some of */dev/kmem* cannot be read because of write-only addresses or unequipped memory addresses.

NAME

mouse – mouse driver

DESCRIPTION

The mouse driver interfaces to the quadrature three button mouse. This driver is a STREAMS driver; thus it can be linked (via I_LINK ioctl) under the *gin* multiplexor. Every clock tick (10 milliseconds), the mouse device is scanned. If the position or the buttons have been changed from the previous reading, new data are generated and sent upstream. Data are encupsulated in the form of the structure of *gin*, which is defined in **<machine/gin.h>**.

COMMANDS FUNCTIONS

mouse recognizes the following ioctl commands. They have to be encoded in I_STR using the *ginioctl* structure. (See *gin*(7).)

G_ENABLE	Enable the mouse device.
G_DISABLE	Disable the mouse device.
G_MSTHRESH	Set the threshold value. Mouse position (either x- or y- coordinate) is considered changed when the two consecutive readings differ at least this threshold value. The default threshold is 2.
G_MGTHRESH	Get the threshold value.
G_RESET	Re-enable the mouse device and set the threshold to the default value.

FILES

/dev/mouse – clone device
/dev/mouse0 – mouse for display 0
/dev/mouse1 – mouse for display 1

SEE ALSO

streamio(7), gin(7)

NAME

null – the null file

DESCRIPTION

Data written on the null special file, */dev/null*, is discarded.

Reads from a null special file always return 0 bytes.

FILES

/dev/null

NAME

raw – raw interface to internal network protocol

DESCRIPTION

raw is a (N by 1) STREAMS multiplexor. The upstream side (N) is clonable (see *clone*(7)). The downstream side is the *ip*(7) multiplexor. *raw*, emulating the functionality of SOCK_RAW as in the BSD system, provides access to the underlying network protocol. Each *raw* stream supports two abstractions: *socket* abstraction as used in the BSD systems and *Transport Level Interface (TLI)* abstraction as used in System V Release 3.

In the socket abstraction, *raw*(7) can operate in either "connection" or "connectionless" mode. In the former, the user has to *connect*(2) to the peer first. Then the data transfer can take place via the normal *read*(2) or *write*(2) system calls. In the latter, the each data transfer may address to a different peer. User uses the *putmsg*(2) and *getmsg*(2) system calls for data transfer. Each transfer unit (called a "message" in the STREAMS parlance) contains an M_PROTO block which encodes the address, followed by M_DATA blocks. The address is specified in the format of *struct sockaddr_in* as described in **<netinet/in.h>**.

In the TLI abstraction, each transfer contains an M_PROTO block and some M_DATA blocks. The M_PROTO has the type T_UNITDATA_REQ for output and T_UNITDATA_IND for input. The address is specified in the format of *struct sockaddr_in* as described in **<netinet/in.h>**.

When network data arrive at the *ip*(7), if the protocol type (in the IP header) is 6, data is sent up to the *tcp*(7); if the type is 11, data is sent up to the *udp*(7); otherwise, data is sent up the *raw*(7) multiplexor. Once here, data is further multiplexed the following way. If a channel has been opened and bound to (via RAWIOC_SETPROTO ioctl) that particular type, then data is sent there. Otherwise, data is directed to the channel which was bound to "wildcard" (that is, "0") protocol type. Data is discarded at the *raw*(7) multiplexor if no s"wildcard" channel.

COMMAND FUNCTIONS

raw recognizes the following *ioctl* commands; their interpretations are described in *tcp*(7).

I_PUSH	
	I_POP
	SOCK_ON
	SOCK_BIND
	SOCK_CONNECT
	SOCK_SHUTDOWN
	SOCK_GETNAME
	SOCK_GETPEER
	SOCK_SETOPT
	SOCK_GETOPT
RAWIOC_SETPROTO	Set the protocol number for the channel.

FILES

/dev/raw	raw clone device
/dev/raw0	raw control channel

SEE ALSO

read(2), write(2), getmsg(2), putmsg(2), socket(2), bind(2), connect(2), send(2), sendto(2), recv(2), recvfrom(2)
streamio(7), tcp(7), udp(7), ip(7)

NAME

/dev/bell – speaker driver /dev/sound – speaker driver

DESCRIPTION

The speaker driver provides a low-level interface to the built-in speaker. To use the speaker, a user program must first open (with the *open(2)* system call) */dev/sound* or */dev/bell.* The speaker is an exclusive use device, therefore, only one process can open the speaker at the one time. Once the speaker is opened successfully by one process, any further attempt to open the speaker results in a failure with EBUSY set in *errno.* Once the speaker is open, the user program accesses the speaker via the *write(2)* system call. The write system call to the speaker deals with the following structure, defined in **<sys/machine/soundreg.h>**:

```
	struct	sound {
			int		sn_vol;
			int		sn_dur;
			int		sn_freq;
	};
```

Every write must be of exactly *sizeof(struct sound)* bytes. An error is returned from the write system call with EINVAL set in *errno* if this count is not correct. The write system call specifies the values of these fields. *Sn_vol* specifies the volume, and takes on an integer value 0, 1, or 2, with 2 being the loudest. *Sn_dur* specifies the duration in milli-seconds. Finally, *sn_freq* specifies the frequency in HZ. As usual, an error is returned with EFAULT set in *errno* if the user program specifies an illegal address.

NAME

streamio – STREAMS ioctl commands

SYNOPSIS

#include <stropts.h>
int ioctl (fildes, command, arg)
int fildes, command;

DESCRIPTION

STREAMS [see *intro*(2)] ioctl commands are a subset of *ioctl*(2) system calls which perform a variety of control functions on *streams*. The arguments *command* and *arg* are passed to the file designated by *fildes* and are interpreted by the *stream head*. Certain combinations of these arguments may be passed to a module or driver in the *stream*.

fildes is an open file descriptor that refers to a *stream*. *command* determines the control function to be performed as described below. *arg* represents additional information that is needed by this command. The type of *arg* depends upon the command, but it is generally an integer or a pointer to a *command*-specific data structure.

Since these STREAMS commands are a subset of *ioctl*, they are subject to the errors described there. In addition to those errors, the call will fail with *errno* set to EINVAL, without processing a control function, if the *stream* referenced by *fildes* is linked below a multiplexor, or if *command* is not a valid value for a *stream*.

Also, as described in *ioctl*, STREAMS modules and drivers can detect errors. In this case, the module or driver sends an error message to the *stream head* containing an error value. This causes subsequent system calls to fail with *errno* set to this value.

COMMAND FUNCTIONS

The following *ioctl* commands, with error values indicated, are applicable to all STREAMS files:

I_PUSH Pushes the module whose name is pointed to by *arg* onto the top of the current *stream*, just below the *stream head*. It then calls the open routine of the newly-pushed module. On failure, *errno* is set to one of the following values:

[EINVAL]	Invalid module name.
[EFAULT]	*arg* points outside the allocated address space.
[ENXIO]	Open routine of new module failed.
[ENXIO]	Hangup received on *fildes*.

I_POP Removes the module just below the *stream head* of the *stream* pointed to by *fildes*. *arg* should be 0 in an I_POP request. On failure, *errno* is set to one of the following values:

[EINVAL]	No module present in the *stream*.
[ENXIO]	Hangup received on *fildes*.

I_LOOK Retrieves the name of the module just below the *stream head* of the *stream* pointed to by *fildes*, and places it in a null terminated character string pointed at by *arg*. The buffer pointed to by *arg* should be at least FMNAMESZ+1 bytes long. An "#include <sys/conf.h>" declaration is required. On failure, *errno* is set to one of the following values:

[EFAULT]	*arg* points outside the allocated address space.
[EINVAL]	No module present in *stream*.

I_FLUSH This request flushes all input and/or output queues, depending on the value of *arg*. Legal *arg* values are:

FLUSHR	Flush read queues.
FLUSHW	Flush write queues.
FLUSHRW	Flush read and write queues.

On failure, *errno* is set to one of the following values:

[EAGAIN]	Unable to allocate buffers for flush message.
[EINVAL]	Invalid *arg* value.
[ENXIO]	Hangup received on *fildes*.

I_SETSIG Informs the *stream head* that the user wishes the kernel to issue the SIGPOLL signal [see *signal*(2) and *sigset*(2)] when a particular event has occurred on the *stream* associated with *fildes*. I_SETSIG supports an asynchronous processing capability in STREAMS. The value of *arg* is a bitmask that specifies the events for which the user should be signaled. It is the bitwise-OR of any combination of the following constants:

S_INPUT	A non-priority message has arrived on a *stream head* read queue, and no other messages existed on that queue before this message was placed there. This is set even if the message is of zero length.
S_HIPRI	A priority message is present on the *stream head* read queue. This is set even if the message is of zero length.
S_OUTPUT	The write queue just below the *stream head* is no longer full. This notifies the user that there is room on the queue for sending (or writing) data downstream.
S_MSG	A STREAMS signal message that contains the SIGPOLL signal has reached the front of the *stream head* read queue.

A user process may choose to be signaled only of priority messages by setting the *arg* bitmask to the value S_HIPRI.

Processes that wish to receive SIGPOLL signals must explicitly register to receive them using I_SETSIG. If several processes register to receive this signal for the same event on the same Stream, each process will be signaled when the event occurs.

If the value of *arg* is zero, the calling process will be unregistered and will not receive further SIGPOLL signals. On failure, *errno* is set to one of the following values:

[EINVAL]	*arg* value is invalid or *arg* is zero and process is not registered to receive the SIGPOLL signal.
[EAGAIN]	Allocation of a data structure to store the signal request failed.

I_GETSIG Returns the events for which the calling process is currently registered to be sent a SIGPOLL signal. The events are returned as a bitmask pointed to by *arg*, where the events are those specified in the description of I_SETSIG above. On failure, *errno* is set to one of the following values:

[EINVAL]	Process not registered to receive the SIGPOLL signal.
[EFAULT]	*arg* points outside the allocated address space.

I_FIND
: This request compares the names of all modules currently present in the *stream* to the name pointed to by *arg*, and returns 1 if the named module is present in the *stream*. It returns 0 if the named module is not present. On failure, *errno* is set to one of the following values:

[EFAULT]	*arg* points outside the allocated address space.
[EINVAL]	*arg* does not contain a valid module name.

I_PEEK
: This request allows a user to retrieve the information in the first message on the *stream head* read queue without taking the message off the queue. *arg* points to a *strpeek* structure which contains the following members:

```
struct strbuf    ctlbuf;
struct strbuf    databuf;
long             flags;
```

The *maxlen* field in the *ctlbuf* and *databuf strbuf* structures [see *getmsg*(2)] must be set to the number of bytes of control information and/or data information, respectively, to retrieve. If the user sets *flags* to RS_HIPRI, I_PEEK will only look for a priority message on the *stream head* read queue.

I_PEEK returns 1 if a message was retrieved, and returns 0 if no message was found on the *stream head* read queue, or if the RS_HIPRI flag was set in *flags* and a priority message was not present on the *stream head* read queue. It does not wait for a message to arrive. On return, *ctlbuf* specifies information in the control buffer, *databuf* specifies information in the data buffer, and *flags* contains the value 0 or RS_HIPRI. On failure, *errno* is set to the following value:

[EFAULT]	*arg* points, or the buffer area specified in *ctlbuf* or *databuf* is, outside the allocated address space.

I_SRDOPT
: Sets the read mode using the value of the argument *arg*. Legal *arg* values are:

RNORM	Byte-stream mode, the default.
RMSGD	Message-discard mode.
RMSGN	Message-nondiscard mode.

Read modes are described in *read*(2). On failure, *errno* is set to the following value:

[EINVAL]	*arg* is not one of the above legal values.

I_GRDOPT
: Returns the current read mode setting in an *int* pointed to by the argument *arg*. Read modes are described in *read*(2). On failure, *errno* is set to the following value:

[EFAULT]	*arg* points outside the allocated address space.

I_NREAD
: Counts the number of data bytes in data blocks in the first message on the *stream head* read queue, and places this value in the location pointed to by *arg*. The return value for the command is the number of messages on the *stream head* read queue. For example, if zero is returned in *arg*, but the *ioctl* return value is greater than zero, this

indicates that a zero-length message is next on the queue. On failure, *errno* is set to the following value:

[EFAULT]	*arg* points outside the allocated address space.

I_FDINSERT creates a message from user specified buffer(s), adds information about another *stream* and sends the message downstream. The message contains a control part and an optional data part. The data and control parts to be sent are distinguished by placement in separate buffers, as described below.

arg points to a *strfdinsert* structure which contains the following members:

```
struct strbuf     ctlbuf;
struct strbuf     databuf;
long              flags;
int               fd;
int               offset;
```

The *len* field in the *ctlbuf strbuf* structure [see *putmsg*(2)] must be set to the size of a pointer plus the number of bytes of control information to be sent with the message. *fd* specifies the file descriptor of the other *stream* and *offset*, which must be word-aligned, specifies the number of bytes beyond the beginning of the control buffer where I_FDINSERT will store a pointer to the *fd stream*'s driver read queue structure. The *len* field in the *databuf strbuf* structure must be set to the number of bytes of data information to be sent with the message or zero if no data part is to be sent.

flags specifies the type of message to be created. A non-priority message is created if *flags* is set to 0, and a priority message is created if *flags* is set to RS_HIPRI. For non-priority messages, I_FDINSERT will block if the *stream* write queue is full due to internal flow control conditions. For priority messages, I_FDINSERT does not block on this condition. For non-priority messages, I_FDINSERT does not block when the write queue is full and O_NDELAY is set. Instead, it fails and sets *errno* to EAGAIN.

I_FDINSERT 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. On failure, *errno* is set to one of the following values:

[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.
[EFAULT]	*arg* points, or the buffer area specified in *ctlbuf* or *databuf* is, outside the allocated address space.
[EINVAL]	One of the following: *fd* in the *strfdinsert* structure is not a valid, open *stream* file descriptor; the size of a pointer plus *offset* is greater than the *len* field for the buffer specified through *ctlptr*; *offset* does not specify a properly-aligned location in the data buffer; an undefined value is stored in *flags*.

[ENXIO]	Hangup received on *fildes*.
[ERANGE]	The *len* field for the buffer specified through *databuf* does not fall within the range specified by the maximum and minimum packet sizes of the topmost *stream* module, or the *len* field for the buffer specified through *databuf* is larger than the maximum configured size of the data part of a message, or the *len* field for the buffer specified through *ctlbuf* is larger than the maximum configured size of the control part of a message.

I_STR Constructs an internal STREAMS ioctl message from the data pointed to by *arg*, and sends that message downstream.

This mechanism is provided to send user *ioctl* requests to downstream modules and drivers. It allows information to be sent with the ioctl, and will return to the user any information sent upstream by the downstream recipient. I_STR blocks until the system responds with either a positive or negative acknowledgement message, or until the request "times out" after some period of time. If the request times out, it fails with *errno* set to ETIME.

At most, one I_STR can be active on a *stream*. Further I_STR calls will block until the active I_STR completes at the *stream head*. The default timeout interval for these requests is 15 seconds. The O_NDELAY [see *open*(2)] flag has no effect on this call.

To send requests downstream, *arg* must point to a *strioctl* structure which contains the following members:

```
int     ic_cmd;        /* downstream command */
int     ic_timout;     /* ACK/NAK timeout */
int     ic_len;        /* length of data arg */
char    *ic_dp;        /* ptr to data arg */
```

ic_cmd is the internal ioctl command intended for a downstream module or driver and *ic_timout* is the number of seconds (-1 = infinite, 0 = use default, >0 = as specified) an I_STR request will wait for acknowledgement before timing out. *ic_len* is the number of bytes in the data argument and *ic_dp* is a pointer to the data argument. The *ic_len* field has two uses: on input, it contains the length of the data argument passed in, and on return from the command, it contains the number of bytes being returned to the user (the buffer pointed to by *ic_dp* should be large enough to contain the maximum amount of data that any module or the driver in the *stream* can return).

The *stream head* will convert the information pointed to by the *strioctl* structure to an internal ioctl command message and send it downstream. On failure, *errno* is set to one of the following values:

[EAGAIN]	Unable to allocate buffers for the *ioctl* message.
[EFAULT]	*arg* points, or the buffer area specified by *ic_dp* and *ic_len* (separately for data sent and data returned) is, outside the allocated address space.
[EINVAL]	*ic_len* is less than 0 or *ic_len* is larger than the maximum configured size of the data part of a message or *ic_timout* is less than -1.

[ENXIO]	Hangup received on *fildes*.
[ETIME]	A downstream *ioctl* timed out before acknowledgement was received.

An I_STR can also fail while waiting for an acknowledgement if a message indicating an error or a hangup is received at the *stream head*. In addition, an error code can be returned in the positive or negative acknowledgement message, in the event the ioctl command sent downstream fails. For these cases, I_STR will fail with *errno* set to the value in the message.

I_SENDFD Requests the *stream* associated with *fildes* to send a message, containing a file pointer, to the *stream head* at the other end of a *stream* pipe. The file pointer corresponds to *arg*, which must be an integer file descriptor.

I_SENDFD converts *arg* into the corresponding system file pointer. It allocates a message block and inserts the file pointer in the block. The user id and group id associated with the sending process are also inserted. This message is placed directly on the read queue [see *intro*(2)] of the *stream head* at the other end of the *stream* pipe to which it is connected. On failure, *errno* is set to one of the following values:

[EAGAIN]	The sending *stream* is unable to allocate a message block to contain the file pointer.
[EAGAIN]	The read queue of the receiving *stream head* is full and cannot accept the message sent by I_SENDFD.
[EBADF]	*arg* is not a valid, open file descriptor.
[EINVAL]	*fildes* is not connected to a *stream* pipe.
[ENXIO]	Hangup received on *fildes*.

I_RECVFD Retrieves the file descriptor associated with the message sent by an I_SENDFD *ioctl* over a *stream* pipe. *arg* is a pointer to a data buffer large enough to hold an *strrecvfd* data structure containing the following members:

```
int fd;
unsigned short uid;
unsigned short gid;
char fill[8];
```

fd is an integer file descriptor. *uid* and *gid* are the user id and group id, respectively, of the sending *stream*.

If O_NDELAY is not set [see *open*(2)], I_RECVFD will block until a message is present at the *stream head*. If O_NDELAY is set, I_RECVFD will fail with *errno* set to EAGAIN if no message is present at the *stream head*.

If the message at the *stream head* is a message sent by an I_SENDFD, a new user file descriptor is allocated for the file pointer contained in the message. The new file descriptor is placed in the *fd* field of the *strrecvfd* structure. The structure is copied into the user data buffer pointed to by *arg*. On failure, *errno* is set to one of the following values:

[EAGAIN]	A message was not present at the *stream head* read queue, and the O_NDELAY flag is set.
[EBADMSG]	The message at the *stream head* read queue was not a message containing a passed file descriptor.
[EFAULT]	*arg* points outside the allocated address space.
[EMFILE]	NOFILES file descriptors are currently open.
[ENXIO]	Hangup received on *fildes*.

The following two commands are used for connecting and disconnecting multiplexed STREAMS configurations.

I_LINK Connects two *streams*, where *fildes* is the file descriptor of the *stream* connected to the multiplexing driver, and *arg* is the file descriptor of the *stream* connected to another driver. The *stream* designated by *arg* gets connected below the multiplexing driver. I_LINK requires the multiplexing driver to send an acknowledgement message to the *stream head* regarding the linking operation. This call returns a multiplexor ID number (an identifier used to disconnect the multiplexor, see I_UNLINK) on success, and a -1 on failure. On failure, *errno* is set to one of the following values:

[ENXIO]	Hangup received on *fildes*.
[ETIME]	Time out before acknowledgement message was received at *stream head*.
[EAGAIN]	Unable to allocate STREAMS storage to perform the I_LINK.
[EBADF]	*arg* is not a valid, open file descriptor.
[EINVAL]	*fildes stream* does not support multiplexing.
[EINVAL]	*arg* is not a *stream*, or is already linked under a multiplexor.
[EINVAL]	The specified link operation would cause a "cycle" in the resulting configuration; that is, if a given *stream head* is linked into a multiplexing configuration in more than one place.

An I_LINK can also fail while waiting for the multiplexing driver to acknowledge the link request, if a message indicating an error or a hangup is received at the *stream head* of *fildes*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_LINK will fail with *errno* set to the value in the message.

I_UNLINK Disconnects the two *streams* specified by *fildes* and *arg*. *fildes* is the file descriptor of the *stream* connected to the multiplexing driver. *arg* is the multiplexor ID number that was returned by the *ioctl* I_LINK command when a *stream* was linked below the multiplexing driver. If *arg* is -1, then all Streams which were linked to *fildes* are disconnected. As in I_LINK, this command requires the multiplexing driver to acknowledge the unlink. On failure, *errno* is set to one of the following values:

[ENXIO]	Hangup received on *fildes*.

[ETIME]	Time out before acknowledgement message was received at *stream head*.
[EAGAIN]	Unable to allocate buffers for the acknowledgement message.
[EINVAL]	Invalid multiplexor ID number.

An I_UNLINK can also fail while waiting for the multiplexing driver to acknowledge the link request, if a message indicating an error or a hangup is received at the *stream head* of *fildes*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_UNLINK will fail with *errno* set to the value in the message.

SEE ALSO

close(2), fcntl(2), intro(2), ioctl(2), open(2), read(2), getmsg(2), poll(2), putmsg(2), signal(2), sigset(2), write(2).

DIAGNOSTICS

Unless specified otherwise above, the return value from *ioctl* is 0 upon success and -1 upon failure with *errno* set as indicated.

NAME

tb – tablet

DESCRIPTION

tb controls the Hitachi Data Tablet. *tb* is a STREAMS module, pushed on top of an RS-232 line driver.

The user can not send data to the tablet, but can receive data. The data are encapsulated in the structure gin format as defined in **<machine/gin.h>**:

```
struct gin {
          unsigned long          gin_msec;
          unsigned long          gin_mux_id;
          unsigned short         gin_id;
          char                   gin_dev;
          char                   gin_dim;
          long                   gin_buttons;
          short                  gin_data[NDIM_GIN];
};
```

gin_id is GID_TBL. *gin_dev* is GDEV_ABS. *gin_dim* is two (2). *gin_buttons* indicates which button is being changed. The first data (*gin_data*[0]) specifies the x-coordinate delta and the second data (*gin_data*[1]) for the y-coordinate.

The default mode settings are:

> resolution: 0.001 inch. output mode: "incremental" mode in that data will be output when the movement of the pen or cursor in the x/y direction is above 0.1 millimeter greater than the set resolution.
>
> output rate: fastest (limited by the RS-232C baud rate).
>
> origin: origin is set at the left bottom corner of the tablet. Thus, all xy coordinate data are positive.
>
> format: packet binary data format is used between the tablet hardware and the *tb* module.

For the standard configuration, the tablet is connected to the serial port "c" (that is, tty line 2). Thus, **/dev/tb** is linked to **/dev/tty2**. The user conventionally opens **/dev/tb**, (via *open*(2)), pops off the tty line discipline, (via I_POP ioctl), and pushes the tb(7) module, (via I_PUSH ioctl).

FILES

/dev/tb

SEE ALSO

streamio(7), gin(7)
Hitachi Data Table Digitizer HDG-0812 HDG-1217 Instruction Manual, Hitachi Seiko, Ltd.

NAME

tcp – internet transmission control protocol (TCP) multiplexor

DESCRIPTION

tcp is a (N by 1) STREAMS multiplexor. The upstream side (N) is clonable (see *clone*(7)). The downstream side is the *ip*(7) multiplexor. *tcp* implements the TCP protocol which provides reliable, flow-controlled, byte-stream, connection-based, two-way transmission of data. *tcp* uses the standard Internet address format and, in addition, provides a per-host collection of "port addresses". Thus, a *tcp* endpoint is identified by the <internet-addr, port> pair. A circuit can then uniquely be specified by two endpoints. a *tcp* endpoint can be either "active" or "passive". The active side initiates connections to the passive side.

tcp supports two abstractions: the *socket* abstraction as in the BSD systems and the *Transport Level Interface (TLI)* abstraction as in System V Release 3. A connection example, for the socket abstraction, follows. A user server program:

– creates (via *open*(2)) a passive *tcp* channel,

– sets it to the socket abstraction (via SOCK_ON ioctl),

– binds (via SOCK_BIND ioctl) to a "listening" address, which can be "wildcard" (INADDR_ANY) as defined in **<netinet/in.h>**,

– listens (via SOCK_LISTEN ioctl) to the network,

– accepts (via SOCK_ACCEPT ioctl) incoming connection requests.

Only active TCP endpoints can initiate connection (via SOCK_CONNECT). Once the connection is established, *read*(2) and *write*(2) system calls are used to transfer data. That is, at the *tcp* upper multiplexor level, data is encoded in the M_DATA STREAMS messages.

In the TLI abstraction, *tcp* implements the kernel level transport provider interface (TPI). This abstraction is used in conjunction with the user level TLI library. Connection setup and takedown procedure is similar, in principle, to that for the socket abstraction, but differs syntactically. For example, a connection request is not presented as an ioctl command; TPI expects an M_PROTO message which contains "T_conn_req" information. Each transfer contains an M_PROTO block and one or more M_DATA blocks. The M_PROTO has the type T_DATA_REQ for output and T_DATA_IND for input. There is no address information in the M_PROTO block, only "MORE_flag" to indicate the continuation of the transport service data unit.

COMMAND FUNCTIONS

tcp recognizes the following *ioctl* commands.

I_LINK
: Link another stream (*ip*) under *tcp*. I_LINK can only happen at channel 0 (the *tcp* control channel). Error codes are:

[EPERM]	The user does not have the "root" privilege.
[EINVAL]	The *tcp* multiplexor has been linked.

I_UNLINK
: Unlink the *ip* stream. I_UNLINK can only happen at channel 0 (the *tcp* control channel). Error codes are:

[EPERM]	The user does not have the "root" privilege.
[EINVAL]	The *tcp* multiplexor is not linked.

SOCK_ON
: Set to "socket" mode.

SOCK_OFF
: Set to "TLI" mode. This is the default mode.

The following are applicable to the TLI abstraction only.

TF_RDWR	Set to "tirdwr" mode such that when data arrives, only M_DATA blocks will be sent up (instead of with the leading M_PROTO block).

The following are applicable to the socket abstraction only. Most of the commands take an argument which is either an integer or a pointer to an address in the form of *struct sockaddr_in* as defined in **<netinet/in.h>**. Other definitions and data structures can be found in **<sys/socklib.h>**.

SOCK_BIND — Bind the specified address to the local endpoint. Error codes are:

[EISCONN]	This channel was bound before.
[EAFNOSUPPORT]	The specified address family is not supported. Currently only the Internet family is supported.
[EADDRINUSE]	Try to bind to a reserved port which was used by another channel and the user does not have the root privilege.
[EADDRNOTAVAIL]	All available ports are currently used.

SOCK_LISTEN — Indicate the willingness to accept connection and set the limit for simultaneous pending connections. Error codes are:

[EISCONN]	The channel is already listening.
[EINVAL]	The argument (limit) is not an integer.
[EINVAL]	The listening channel is not bound yet.
[EBUSY]	Another channel is already listening in the same address.

SOCK_REJECT — (The listener) refuse the connection request. There are potential many pending requests. The socket abstraction always refuses the first request, the one in the front of the queue. The TLI abstraction has the option to refuse any request. Error codes are:

[EINVAL]	There is no pending connection request.

SOCK_ACCEPT — Accept a connection.
This interface resembles the I_FDINSERT STREAMS ioctl. An accepting scenario is as follow. When the connection request comes in to the listening channel, The following structure encoded in the M_PROTO is sent upstream to indicate a connection request just arrives.

```
typedef struct {
        ulong   operation;          /* SOCK_CONNIND here */
        ulong   fildes;             /* fd is accepted on */
        ulong   seq;                /* sequence number */
        int     family;             /* address family */
        union a {                   /* foreign address */
                struct sockaddr_in in;
                struct sockaddr_un un;
        } rem_addr;
} socketop_t;
```

Once the server decides to accept the connection, it first opens a new *tcp* channel and puts the file descriptor in *fildes*. Then the server issues this ioctl (SOCK_ACCEPT), again using the argument *socketop_t*. The pending connection request is then associated with the new channel. A circuit is now considered established.
If the SOCK_ACCEPT is done before any connection request, the ioctl is held and not acknowledged. This effectively puts the caller to sleep (until the connection comes).
Error codes are:

[EINVAL]	The argument does not point to a correct *socketop_t* structure.
[EINVAL]	*fildes* is not a valid stream file descriptor or not associated with any *tcp*(7) channel.
[EINVAL]	The accepting channel is not in the listening state.

SOCK_CONNECT Connect to another tcp endpoint whose address is specified in the argument. Error codes are:

[EINVAL]	Address format is wrong.
[EADDRNOTAVAIL]	If the local endpoint was not bound previously, now attempt to bind first. Return this error if no free port is available.
[EISCONN]	This channel is already in the "connect" state.
[EAFNOSUPPORT]	The specified address family is not supported. Currently only the Internet family is supported.
[ENOBUFS]	Can not allocate internal resources used for the connection.

SOCK_SHUTDOWN Shutdown either the receiving side (argument is 0), or the sending side (argument is 1), or both sides (argument is 2).

SOCK_GETNAME Return the address of our own endpoint.

SOCK_GETPEER Return the address of the peer endpoint which is connected to us.

SOCK_SETOPT Set socket options. For options that are recognized, see either **<sys/socket.h>** or *setsockopt*(2) manual page.

SOCK_GETOPT Get socket options. For options that are recognized, see either **<sys/socket.h>** or *getsockopt*(2) manual page.

SOCK_SENDOOB There may be one or more M_DATA blocks following the M_IOCTL. These data blocks are sent to peer as the out-of-band data. Normally only one byte of OOB data is present.

SOCK_RECVOOB Retrieve the out-of-band data (only one byte). If there is no OOB data pending, the ioctl returns error code EWOULDBLOCK.

FILES

/dev/tcp	tcp clone device
/dev/tcp0	tcp control channel

SEE ALSO

read(2), write(2), getmsg(2), putmsg(2), socket(2), bind(2), connect(2), send(2), sendto(2), recv(2), recvfrom(2), getsockopt(2), setsockopt(2), listen(2), accept(2), shutdown(2), getsockname(2), getpeername(2)

streamio(7), clone(7), raw(7), udp(7), ip(7)

NAME

termio – general terminal interface

DESCRIPTION

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

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

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

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

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

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

INTR	(Rubout or ASCII DEL) generates an *interrupt* signal which is sent to all processes with the associated control terminal. Normally, each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location; see *signal*(2).
QUIT	(Control-| or ASCII FS) generates a *quit* signal. Its treatment is identical to the interrupt signal except that, unless a receiving process has made other arrangements, it will not only be terminated but a core image file (called **core**) will be created in the current working directory.
SWTCH	(Control-z or ASCII SUB) is used by the job control facility, *shl*, to change the current layer to the control layer.

ERASE	(#) erases the preceding character. It will not erase beyond the start of a line, as delimited by a NL, EOF, or EOL character.
KILL	(@) deletes the entire line, as delimited by a NL, EOF, or EOL character.
EOF	(Control-d or ASCII EOT) may be used to generate an end-of-file from a terminal. When received, all the characters waiting to be read are immediately passed to the program, without waiting for a new-line, and the EOF is discarded. Thus, if there are no characters waiting, which is to say the EOF occurred at the beginning of a line, zero characters will be passed back, which is the standard end-of-file indication.
NL	(ASCII LF) is the normal line delimiter. It can not be changed or escaped.
EOL	(ASCII NUL) is an additional line delimiter, like NL. It is not normally used.
EOL2	is another additional line delimiter.
STOP	(Control-s or ASCII DC3) can be used to temporarily suspend output. It is useful with CRT terminals to prevent output from disappearing before it can be read. While output is suspended, STOP characters are ignored and not read.
START	(Control-q or ASCII DC1) is used to resume output which has been suspended by a STOP character. While output is not suspended, START characters are ignored and not read. The start/stop characters can not be changed or escaped.

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

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

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

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

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

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

```
0   VINTR    DEL
```

1	VQUIT	FS
2	VERASE	#
3	VKILL	@
4	VEOF	EOT
5	VEOL	NUL
6	reserved	
7	SWTCH	

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

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

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

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

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

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

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

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

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

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

The initial input control value is all-bits-clear.

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

OPOST	0000001	Postprocess output.
OLCUC	0000002	Map lower case to upper on output.
ONLCR	0000004	Map NL to CR-NL on output.
OCRNL	0000010	Map CR to NL on output.
ONOCR	0000020	No CR output at column 0.
ONLRET	0000040	NL performs CR function.
OFILL	0000100	Use fill characters for delay.
OFDEL	0000200	Fill is DEL, else NUL.
NLDLY	0000400	Select new-line delays:
NL0	0	
NL1	0000400	
CRDLY	0003000	Select carriage-return delays:
CR0	0	
CR1	0001000	
CR2	0002000	
CR3	0003000	
TABDLY	0014000	Select horizontal-tab delays:
TAB0	0	
TAB1	0004000	
TAB2	0010000	
TAB3	0014000	Expand tabs to spaces.
BSDLY	0020000	Select backspace delays:
BS0	0	
BS1	0020000	
VTDLY	0040000	Select vertical-tab delays:
VT0	0	
VT1	0040000	
FFDLY	0100000	Select form-feed delays:
FF0	0	
FF1	0100000	

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

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

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

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

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

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

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

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

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

The actual delays depend on line speed and system load.

The initial output control value is all bits clear.

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

CBAUD	0000017	Baud rate:
B0	0	Hang up
B50	0000001	50 baud
B75	0000002	75 baud
B110	0000003	110 baud
B134	0000004	134 baud
B150	0000005	150 baud
B200	0000006	200 baud
B300	0000007	300 baud
B600	0000010	600 baud
B1200	0000011	1200 baud
B1800	0000012	1800 baud
B2400	0000013	2400 baud
B4800	0000014	4800 baud
B9600	0000015	9600 baud
B19200	0000016	19200 baud
EXTA	0000016	External A
B38400	0000017	38400 baud
EXTB	0000017	External B
CSIZE	0000060	Character size:
CS5	0	5 bits
CS6	0000020	6 bits
CS7	0000040	7 bits
CS8	0000060	8 bits
CSTOPB	0000100	Send two stop bits, else one.
CREAD	0000200	Enable receiver.
PARENB	0000400	Parity enable.
PARODD	0001000	Odd parity, else even.
HUPCL	0002000	Hang up on last close.
CLOCAL	0004000	Local line, else dial-up.
RCV1EN	0010000	
XMT1EN	0020000	
LOBLK	0040000	Block layer output.

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

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

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

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

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

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

If LOBLK is set, the output of a job control layer will be blocked when it is not the current layer. Otherwise the output generated by that layer will be multiplexed onto the current layer.

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

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

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

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

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

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

for:	*use*:
`	\'
|	\!
~	\^
{	\(
}	\)
\	\\

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

If ECHO is set, characters are echoed as received.

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

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

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

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

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

The commands using this form are:

TCGETA	Get the parameters associated with the terminal and store in the *termio* structure referenced by **arg**.
TCSETA	Set the parameters associated with the terminal from the structure referenced by **arg**. The change is immediate.
TCSETAW	Wait for the output to drain before setting the new parameters. This form should be used when changing parameters that will affect output.
TCSETAF	Wait for the output to drain, then flush the input queue and set the new parameters.

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

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

The commands using this form are:

TCSBRK	Wait for the output to drain. If *arg* is 0, then send a break (zero bits for 0.25 seconds).
TCXONC	Start/stop control. If *arg* is 0, suspend output; if 1, restart suspended output.
TCFLSH	If *arg* is 0, flush the input queue; if 1, flush the output queue; if 2, flush both the input and output queues.

FILES

/dev/tty*

SEE ALSO

fork(2), ioctl(2), setpgrp(2), signal(2), stty(1).

NAME

tigr – 1500/3000 graphics interface

DESCRIPTION

The character special device files /dev/tigr* form the interface between the operating system and graphics programs. The operating system uses the device files to control the Stardent 1500/3000 graphics hardware. Operations on the files result in the transfer of commands and data between system memory and the graphics hardware. Other file operations result in synchronization among processes, hardware, and the driver.

The file */dev/tigr0* is used internally by the operating system as its interface for console output when not running the X11 window system. Other minor device files */dev/tigr[1-9]* may be used by any process. In the most common case, the X11 window server will use */dev/tigr1*, and the other minor devices */dev/tigr[2-9]* will be available for applications (such as applications using Doré). Most standard clients of the X11 window server (including xterm, xclock, xload, etc.) deal only with the window server through a communications channel [such as DISPLAY='hostname':0, or DISPLAY=unix:0] and have no direct interaction with the graphics hardware. While wanting to coexist within a window system, Doré applications usually want higher speed or more functionality (such as depth buffering) than provided by typical window servers, and find it necessary to use the *tigr direct graphics* interface.

Each minor device is an exclusive-open device: a close() must be performed between any two open()s. There are no read(), write(), or lseek() calls; ioctl() calls do all the work. The Stardent 1500/3000 graphics board has a one-megabyte address space for drawing commands and data, along with an address translation table which allows mapping to any 256 4Kbyte pages of system memory. (See the *Hardware Reference Manual,* part number 340-0011.) Although certain functions can be performed without using the graphics address space, all drawing commands must pass through the graphics address space. The */dev/tigr* driver creates and manages a shared-memory interface among the application, the driver, and the graphics hardware. Since the pages of the shared memory region are "locked down," they provide high bandwidth, low latency, and low overhead for graphics operations.

As defined in *<machine/tigr.h>*, the shared memory contains one header control structure *cbmem,* two circular buffers (*cmd* and *rpl* for command and reply), and arbitrary data regions. The *cbmem* structure lies at the low-address end of the region. It contains an identifying tag, the length of the region, graphics synchronization locks/keys, and the pointers for the circular buffers. One circular buffer, the command buffer, handles communication from the user to the kernel. The other (reply) buffer takes care of communication from the kernel to the user. All "pointers" within the tigr shared memory region are actually byte displacements from the lowest address in the region, making them region-relative. The *cbmem* structure must be aligned on an 8-byte address boundary.

Four pointers contained in a *cbhdr* structure control each circular buffer. The *fwa* and *lwa* pointers describe the lowest address, and one beyond the highest address, used for the buffer. The *fwa* and *lwa* must be constant for the life of the buffer. Two other pointers, *in* and *out,* vary as the buffer is used. *in* points to the location which will receive the next byte inserted into the buffer. *out* points to the location containing the next byte to be removed from the buffer. Whenever *in==out*, the buffer is empty. If *in>out*, then the buffer contains *in–out* bytes beginning at *out*. If *in<out*, then the buffer contains *in–out+lwa–fwa* total bytes; *lwa–out* bytes from *out* to *lwa,followed*by *in–fwa* bytes from *fwa* to *in*.

FILES

/dev/tigrx – a minor device for display 0
/dev/tigr1x – a minor device for display 1

NAME

tty – controlling terminal interface

DESCRIPTION

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

FILES

/dev/tty
/dev/tty*

SEE ALSO

console(7)

NAME

udom – Unix domain IPC driver

DESCRIPTION

udom is a clonable STREAMS driver. It implements the *BSD* UNIX domain IPC in that two processes not necessarily having a common ancestor can communicate. Communicating parties refer to its peer by name or address. In the *BSD* implementation, this name must be a path name within the file system name space. This is not true in *udom*, where the name is not tied to the file system. For instance, two processes can each open */dev/udom*. (they get different *udom* channels because of cloning). One process binds the name *fred* to it. Another process can then send data to the channel *fred*.

udom presents two abstractions for communication: "byte stream" or "datagram." In the byte stream mode, a client creates an endpoint (via *open*(2)), connects to peer (via SOCK_CONNECT ioctl), and sends or receives data using *read*(2), *write*(2), *send*(2), or *recv*(2). The server, similar to the situation in the Internet Domain, creates an endpoint, binds a well-known address (via SOCK_BIND ioctl), listens to it (via SOCK_LISTEN ioctl), and accepts connection (via SOCK_ACCEPT ioctl). In the datagram mode, the communicating parties use *sendto*(2) and *recvfrom*(2) to transfer data. Data sent to the *udom* driver is in the form of M_PROTO block followed by one or more M_DATA blocks. The M_PROTO block encodes the address whose format is defined in **<sys/un.h>**.

COMMAND FUNCTIONS

udom(7) recognizes the following *ioctl* commands.

SOCK_BIND — Bind the specified address to the local endpoint. Error codes are:

[EINVAL]	This endpoint has already been bound.
[EINVAL]	Address format is incorrect.
[EADDRINUSE]	Try to bind to an address which was used by another channel.

SOCK_LISTEN — Indicate the willingness to accept connection and set the limit for simultaneous pending connections. Error codes are:

[EINVAL]	The listening channel is not bound yet.

SOCK_ACCEPT — Accept a connection.
This interface resembles the I_FDINSERT STREAMS ioctl. An accepting scenario is as follow. When the connection request comes in to the listening channel, The following structure encoded in the M_PROTO is sent upstream to indicate a connection request just arrives.

```
typedef struct {
        ulong  operation;        /* SOCK_CONNIND here */
        ulong  fildes;           /* fd is accepted on */
        ulong  seq;              /* sequence number */
        int    family;           /* address family */
        union a {                /* foreign address */
                struct sockaddr_in in;
                struct sockaddr_un un;
        } rem_addr;
} socketop_t;
```

Once the server decides to accept the connection, it first opens a new *udom* channel and puts the file descriptor in *fildes*. Then the server issues this ioctl (SOCK_ACCEPT), again using the argument *socketop_t*. The pending connection request is then

associated with the new channel. A connection is now considered established.
If the SOCK_ACCEPT is done before any connection request, the ioctl is held and not acknowledged. This effectively puts the caller to sleep (until the connection comes).
Error codes are:

[EINVAL]	The argument does not point to a correct *socketop_t* structure.
[EINVAL]	*fildes* is not a valid stream file descriptor or not associated with any *udom*(7) channel.

SOCK_CONNECT Connect to another udom endpoint whose address is specified in the argument. Error codes are:

[EINVAL]	The address format is wrong.
[EADDRNOTAVAIL]	No channels have been bound to the address we want to connect.
[ECONNREFUSED]	The listening channel already has maximum number (default to 5) of pending connection requests.
[ENOBUFS]	Can not allocate internal resources used for the connection.

SOCK_GETNAME Return the address of our own endpoint.

SOCK_GETPEER Return the address of the peer endpoint which is connected to us.

FILES

/dev/udom udom clone device

SEE ALSO

read(2), write(2), getmsg(2), putmsg(2), socket(2), bind(2), connect(2), send(2), sendto(2), recv(2), recvfrom(2), listen(2), accept(2), getsockname(2), getpeername(2) streamio(7), clone(7), tcp(7)

NAME

udp – user datagram protocol (UDP) interface

DESCRIPTION

udp is a (N by 1) STREAMS multiplexor. The upstream side (N) is clonable (see *clone*(7)). The downstream side is the *ip*(7) multiplexor. *udp* provides a simple, unreliable datagram protocol (UDP) service. Each *udp* stream supports two abstractions: *socket* abstraction as used in the BSD systems and *Transport Level Interface (TLI)* abstraction as used in the System V Release 3.

In the socket abstraction, *udp*(7) can operate in either "connection" or "connectionless" mode. In the former, the user has to *connect*(2) to the peer first. Then the data transfer can take place via the normal *read*(2) or *write*(2) system calls. In the latter, the each data transfer may address to a different peer. User uses the *putmsg*(2) and *getmsg*(2) system calls for data transfer. Each transfer unit (called a "message" in the STREAMS parlance) contains an M_PROTO block which encodes the address, followed by M_DATA blocks. The address is specified in the format of *struct sockaddr_in* as described in **<netinet/in.h>**.

In the TLI abstraction, each transfer contains an M_PROTO block and some M_DATA blocks. The M_PROTO has the type T_UNITDATA_REQ for output and T_UNITDATA_IND for input. The address is specified in the format of *struct sockaddr_in* as described in **<netinet/in.h>**.

COMMAND FUNCTIONS

udp recognizes the following *ioctl* commands; their interpretations are described in *tcp(7)*.

I_PUSH
I_POP
SOCK_ON
SOCK_BIND
SOCK_CONNECT
SOCK_SHUTDOWN
SOCK_GETNAME
SOCK_GETPEER

FILES

/dev/udp	udp clone device
/dev/udp0	udp control channel

SEE ALSO

read(2), write(2), getmsg(2), putmsg(2), socket(2), bind(2), connect(2), send(2), sendto(2), recv(2), recvfrom(2)
streamio(7), tcp(7), clone(7), ip(7)