sun®
microsystems

SunOS Reference Manual

Part Number: 800-3827-10
Revision A of 27 March, 1990

Thc Sun logo, Sun Microsystems, Sun Workstation, NFS, and TOPS are registered trademarks of Sun Microsystcms, Inc.

Sun, Sun-2, Sun-3, Sun-4, Sun386*i*, SPARCstation, SPARCserver, NeWS, NSE, OpenWindows, SPARC, SunInstall, SunLink, SunNet, SunOS, SunPro, and SunView are trademarks of Sun Microsystems, Inc.

UNIX is a registered trademark of AT&T; OPEN LOOK is a trademark of AT&T.

All othcr products or services mentioned in this document are identified by the trademarks or service marks of their respective companies or organizations, and Sun Microsystems, Inc. disclaims any responsibility for specifying which marks are owned by which companies or organizations.

This logo is a trademark of the X/Open Company Limited in the UK and other countries, and its use is licensed to Sun Microsystems, Inc. The use of this logo certifies SunOS 4.1 conformance with X/Opcn Portability Guide Issue 2 (XPG 2).

Restricted rights legend: use, duplication, or disclosure by the U.S. government is subject to restrictions set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013 and in similar clauses in the FAR and NASA FAR Supplement.

The Sun Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees.

This product is protected by one or more of the following U.S. patents: 4,777,485 4,688,190 4,527,232 4,745,407 4,679,014 4,435,792 4,719,569 4,550,368 in addition to foreign patents and applications pending.

This software and documentation is based in part on the Fourth Berkeley Software Distribution under license from the Rcgents of the University of California. We acknowledge the following individuals and institutions for their role in its development: The Regents of the University of California, the Electrical Engineering and Computer Sciences Dcpartment at the Berkeley Campus of the University of California, and Other Contributors.

NAME

intro – introduction to user-level library functions

DESCRIPTION

Section 3 describes user-level library routines. In this release, most user-library routines are listed in alphabetical order regardless of their subsection headings. (This eliminates having to search through several subsections of the manual.) However, due to their special-purpose nature, the routines from the following libraries are broken out into the indicated subsections:

- The Lightweight Processes Library, in subsection 3L.
- The Mathematical Library, in subsection 3M.
- The RPC Services Library, in subsection 3R.

A 3V section number means one or more of the following:

- The man page documents System V behavior only.
- The man page documents default SunOS behavior, and System V behavior as it differs from the default behavior. These System V differences are presented under **SYSTEM V** section headers.
- The man page documents behavior compliant with *IEEE Std 1003.1-1988* (POSIX.1).

The System V Library was formerly documented in a separate manual section. These man pages have been merged into the main portion of section 3. These man pages describe functions that may differ from the default SunOS functions. To use them, compile programs with **/usr/5bin/cc** instead of **/usr/bin/cc**.

Section 3 also documents the library interfaces for *X/Open Portability Guide, Issue 2* (XPG2) compatibility. Where these interfaces differ from the System V versions, the differences are noted. To use the XPG2 compatibility library interfaces, compile programs with **/usr/xpg2bin/cc**.

The libraries provide many different "standard" environments. These environments (including two that are not yet fully supported) are described on **ansic**(7V), **bsd**(7), **posix**(7V), **sunos**(7), **svidii**(7V), **svidiii**(7V), and **xopen**(7V).

The main C library, **/usr/lib/libc.a**, contains many of the functions described in this section, along with entry points for the system calls described in Section 2. This library also includes the Internet networking routines listed under the 3N subsection heading, and routines provided for compatibility with other UNIX operating systems, listed under 3C. Functions associated with the "standard I/O library" are listed under 3S.

User-level routines for access to data structures within the kernel and other processes are listed under 3K. To use these functions, compile programs with the **–lkvm** option for the C compiler, **cc**(1V).

Math library functions are listed under 3M. To use them, compile programs with the **–lm cc**(1V) option.

Various specialized libraries, the routines they contain, and the compiler options needed to link with them, are listed under 3X.

FILES

/usr/lib/libc.a	C Library (2, 3, 3N and 3C)
/usr/lib/lib*.a	other "standard" C libraries
/usr/lib/lib*.a	special-purpose C libraries
/usr/5bin/cc	

SEE ALSO

cc(1V), **ld**(1), **nm**(1), **intro**(2)

LIST OF LIBRARY FUNCTIONS

Name	Appears on Page	Description
a64l	a64l(3)	convert between long integer and base-64 ASCII string
abort	abort(3)	generate a fault
abs	abs(3)	integer absolute value
addexportent	exportent(3)	get exported file system information
addmntent	getmntent(3)	get file system descriptor file entry
aiocancel	aiocancel(3)	cancel an asynchronous operation
aioread	aioread(3)	asynchronous I/O operations
aiowait	aiowait(3)	wait for completion of asynchronous I/O operation
aiowrite	aioread(3)	asynchronous I/O operations
alarm	alarm(3V)	schedule signal after specified time
alloca	malloc(3V)	memory allocator
alphasort	scandir(3)	scan a directory
arc	plot(3X)	graphics interface
asctime	ctime(3V)	convert date and time
assert	assert(3V)	program verification
atof	strtod(3)	convert string to double-precision number
atoi	strtol(3)	convert string to integer
atol	strtol(3)	convert string to integer
audit_args	audit_args(3)	produce text audit message
audit_text	audit_args(3)	produce text audit message
auth_destroy	rpc_clnt_auth(3N)	library routines for client side RPC authentication
authdes_create	secure_rpc(3N)	library routines for secure remote procedure calls
authdes_getucred	secure_rpc(3N)	library routines for secure remote procedure calls
authnone_create	rpc_clnt_auth(3N)	library routines for client side RPC authentication
authunix_create	rpc_clnt_auth(3N)	library routines for client side RPC authentication
authunix_create_default	rpc_clnt_auth(3N)	library routines for client side RPC authentication
bcmp	bstring(3)	bit and byte string operations
bcopy	bstring(3)	bit and byte string operations
bindresvport	bindresvport(3N)	bind a socket to a privileged IP port
bsearch	bsearch(3)	binary search a sorted table
bstring	bstring(3)	bit and byte string operations
byteorder	byteorder(3N)	convert values between host and network byte order
bzero	bstring(3)	bit and byte string operations
calloc	malloc(3V)	memory allocator
callrpc	rpc_clnt_calls(3N)	library routines for client side calls
catclose	catopen(3C)	open/close a message catalog
catgetmsg	catgets(3C)	get message from a message catalog
catgets	catgets(3C)	get message from a message catalog
catopen	catopen(3C)	open/close a message catalog
cbc_crypt	des_crypt(3)	fast DES encryption
cfgetispeed	termios(3V)	terminal control functions
cfgetospeed	termios(3V)	terminal control functions
cfree	malloc(3V)	memory allocator
cfsetispeed	termios(3V)	terminal control functions
cfsetospeed	termios(3V)	terminal control functions
circle	plot(3X)	graphics interface
clearerr	ferror(3V)	stream status inquiries
clnt_broadcast	rpc_clnt_calls(3N)	library routines for client side calls
clnt_call	rpc_clnt_calls(3N)	library routines for client side calls
clnt_control	rpc_clnt_create(3N)	library routines creating and manipulating CLIENT handles

clnt_create	**rpc_clnt_create**(3N)	library routines creating and manipulating CLIENT handles
clnt_create_vers	**rpc_clnt_create**(3N)	library routines creating and manipulating CLIENT handles
clnt_destroy	**rpc_clnt_create**(3N)	library routines creating and manipulating CLIENT handles
clnt_freeres	**rpc_clnt_calls**(3N)	library routines for client side calls
clnt_geterr	**rpc_clnt_calls**(3N)	library routines for client side calls
clnt_pcreateerror	**rpc_clnt_create**(3N)	library routines creating and manipulating CLIENT handles
clnt_perrno	**rpc_clnt_calls**(3N)	library routines for client side calls
clnt_perror	**rpc_clnt_calls**(3N)	library routines for client side calls
clnt_spcreateerror	**rpc_clnt_create**(3N)	library routines creating and manipulating CLIENT handles
clnt_sperrno	**rpc_clnt_calls**(3N)	library routines for client side calls
clnt_sperror	**rpc_clnt_calls**(3N)	library routines for client side calls
clntraw_create	**rpc_clnt_create**(3N)	library routines creating and manipulating CLIENT handles
clnttcp_create	**rpc_clnt_create**(3N)	library routines creating and manipulating CLIENT handles
clntudp_bufcreate	**rpc_clnt_create**(3N)	library routines creating and manipulating CLIENT handles
clock	**clock**(3C)	report CPU time used
closedir	**directory**(3V)	directory operations
closelog	**syslog**(3)	control system log
closepl	**plot**(3X)	graphics interface
cont	**plot**(3X)	graphics interface
conv	**ctype**(3V)	character classification and conversion macros and functions
crypt	**crypt**(3)	password and data encryption
ctermid	**ctermid**(3V)	generate filename for terminal
ctime	**ctime**(3V)	convert date and time
ctype	**ctype**(3V)	character classification and conversion macros and functions
curses	**curses**(3V)	System V terminal screen handling and optimization package
cuserid	**cuserid**(3V)	get character login name of the user
dbm	**dbm**(3X)	data base subroutines
dbm_clearerr	**ndbm**(3)	data base subroutines
dbm_close	**ndbm**(3)	data base subroutines
dbm_delete	**ndbm**(3)	data base subroutines
dbm_error	**ndbm**(3)	data base subroutines
dbm_fetch	**ndbm**(3)	data base subroutines
dbm_firstkey	**ndbm**(3)	data base subroutines
dbm_nextkey	**ndbm**(3)	data base subroutines
dbm_open	**ndbm**(3)	data base subroutines
dbm_store	**ndbm**(3)	data base subroutines
dbmclose	**dbm**(3X)	data base subroutines
dbminit	**dbm**(3X)	data base subroutines
decimal_to_double	**decimal_to_floating**(3)	convert decimal record to floating-point value
decimal_to_extended	**decimal_to_floating**(3)	convert decimal record to floating-point value
decimal_to_single	**decimal_to_floating**(3)	convert decimal record to floating-point value
delete	**dbm**(3X)	data base subroutines
des_crypt	**des_crypt**(3)	fast DES encryption
des_setparity	**des_crypt**(3)	fast DES encryption
directory	**directory**(3V)	directory operations
dlclose	**dlopen**(3X)	simple programmatic interface to the dynamic linker
dlerror	**dlopen**(3X)	simple programmatic interface to the dynamic linker
dlopen	**dlopen**(3X)	simple programmatic interface to the dynamic linker
dlsym	**dlopen**(3X)	simple programmatic interface to the dynamic linker
dn_comp	**resolver**(3)	resolver routines
dn_expand	**resolver**(3)	resolver routines
double_to_decimal	**floating_to_decimal**(3)	convert floating-point value to decimal record
drand48	**drand48**(3)	generate uniformly distributed pseudo-random numbers

dysize	**ctime**(3V)	convert date and time
ecb_crypt	**des_crypt**(3)	fast DES encryption
econvert	**econvert**(3)	output conversion
ecvt	**econvert**(3)	output conversion
edata	**end**(3)	last locations in program
encrypt	**crypt**(3)	password and data encryption
end	**end**(3)	last locations in program
endac	**getacinfo**(3)	get audit control file information
endexportent	**exportent**(3)	get exported file system information
endfsent	**getfsent**(3)	get file system descriptor file entry
endgraent	**getgraent**(3)	get group adjunct file entry
endgrent	**getgrent**(3V)	get group file entry
endhostent	**gethostent**(3N)	get network host entry
endmntent	**getmntent**(3)	get file system descriptor file entry
endnetent	**getnetent**(3N)	get network entry
endnetgrent	**getnetgrent**(3N)	get network group entry
endprotoent	**getprotoent**(3N)	get protocol entry
endpwaent	**getpwaent**(3)	get password adjunct file entry
endpwent	**getpwent**(3V)	get password file entry
endrpcent	**getrpcent**(3N)	get RPC entry
endservent	**getservent**(3N)	get service entry
endttyent	**getttyent**(3)	get ttytab file entry
endusershell	**getusershell**(3)	get legal user shells
erand48	**drand48**(3)	generate uniformly distributed pseudo-random numbers
erase	**plot**(3X)	graphics interface
errno	**perror**(3)	system error messages
etext	**end**(3)	last locations in program
ether_aton	**ethers**(3N)	Ethernet address mapping operations
ether_hostton	**ethers**(3N)	Ethernet address mapping operations
ether_line	**ethers**(3N)	Ethernet address mapping operations
ether_ntoa	**ethers**(3N)	Ethernet address mapping operations
ether_ntohost	**ethers**(3N)	Ethernet address mapping operations
ethers	**ethers**(3N)	Ethernet address mapping operations
execl	**execl**(3V)	execute a file
execle	**execl**(3V)	execute a file
execlp	**execl**(3V)	execute a file
execv	**execl**(3V)	execute a file
execvp	**execl**(3V)	execute a file
exit	**exit**(3)	terminate a process after performing cleanup
exportent	**exportent**(3)	get exported file system information
extended_to_decimal	**floating_to_decimal**(3)	convert floating-point value to decimal record
fclose	**fclose**(3V)	close or flush a stream
fconvert	**econvert**(3)	output conversion
fcvt	**econvert**(3)	output conversion
fdopen	**fopen**(3V)	open a stream
feof	**ferror**(3V)	stream status inquiries
ferror	**ferror**(3V)	stream status inquiries
fetch	**dbm**(3X)	data base subroutines
fflush	**fclose**(3V)	close or flush a stream
ffs	**bstring**(3)	bit and byte string operations
fgetc	**getc**(3V)	get character or integer from stream
fgetgraent	**getgraent**(3)	get group adjunct file entry
fgetgrent	**getgrent**(3V)	get group file entry

fgetpwaent	**getpwaent**(3)	get password adjunct file entry
fgetpwent	**getpwent**(3V)	get password file entry
fgets	**gets**(3S)	get a string from a stream
file_to_decimal	**string_to_decimal**(3)	parse characters into decimal record
fileno	**ferror**(3V)	stream status inquiries
firstkey	**dbm**(3X)	data base subroutines
floatingpoint	**floatingpoint**(3)	IEEE floating point definitions
fopen	**fopen**(3V)	open a stream
fprintf	**printf**(3V)	formatted output conversion
fputc	**putc**(3S)	put character or word on a stream
fputs	**puts**(3S)	put a string on a stream
fread	**fread**(3S)	buffered binary input/output
free	**malloc**(3V)	memory allocator
freopen	**fopen**(3V)	open a stream
fscanf	**scanf**(3V)	formatted input conversion
fseek	**fseek**(3S)	reposition a stream
ftell	**fseek**(3S)	reposition a stream
ftime	**time**(3V)	get date and time
ftok	**ftok**(3)	standard interprocess communication package
ftw	**ftw**(3)	walk a file tree
func_to_decimal	**string_to_decimal**(3)	parse characters into decimal record
fwrite	**fread**(3S)	buffered binary input/output
gcd	**mp**(3X)	multiple precision integer arithmetic
gconvert	**econvert**(3)	output conversion
gcvt	**econvert**(3)	output conversion
get_myaddress	**secure_rpc**(3N)	library routines for secure remote procedure calls
getacdir	**getacinfo**(3)	get audit control file information
getacflg	**getacinfo**(3)	get audit control file information
getacinfo	**getacinfo**(3)	get audit control file information
getacmin	**getacinfo**(3)	get audit control file information
getauditflagsbin	**getauditflags**(3)	convert audit flag specifications
getauditflagschar	**getauditflags**(3)	convert audit flag specifications
getc	**getc**(3V)	get character or integer from stream
getchar	**getc**(3V)	get character or integer from stream
getcwd	**getcwd**(3V)	get pathname of current working directory
getenv	**getenv**(3V)	return value for environment name
getexportent	**exportent**(3)	get exported file system information
getexportopt	**exportent**(3)	get exported file system information
getfauditflags	**getfauditflags**(3)	generates the process audit state
getfsent	**getfsent**(3)	get file system descriptor file entry
getfsfile	**getfsent**(3)	get file system descriptor file entry
getfsspec	**getfsent**(3)	get file system descriptor file entry
getfstype	**getfsent**(3)	get file system descriptor file entry
getgraent	**getgraent**(3)	get group adjunct file entry
getgranam	**getgraent**(3)	get group adjunct file entry
getgrent	**getgrent**(3V)	get group file entry
getgrgid	**getgrent**(3V)	get group file entry
getgrnam	**getgrent**(3V)	get group file entry
gethostbyaddr	**gethostent**(3N)	get network host entry
gethostbyname	**gethostent**(3N)	get network host entry
gethostent	**gethostent**(3N)	get network host entry
getlogin	**getlogin**(3V)	get login name
getmntent	**getmntent**(3)	get file system descriptor file entry

getnetbyaddr	**getnetent**(3N)	get network entry
getnetbyname	**getnetent**(3N)	get network entry
getnetent	**getnetent**(3N)	get network entry
getnetgrent	**getnetgrent**(3N)	get network group entry
getnetname	**secure_rpc**(3N)	library routines for secure remote procedure calls
getopt	**getopt**(3)	get option letter from argument vector
getpass	**getpass**(3V)	read a password
getprotobyname	**getprotoent**(3N)	get protocol entry
getprotobynumber	**getprotoent**(3N)	get protocol entry
getprotoent	**getprotoent**(3N)	get protocol entry
getpublickey	**publickey**(3R)	get public or secret key
getpw	**getpw**(3)	get name from uid
getpwaent	**getpwaent**(3)	get password adjunct file entry
getpwanam	**getpwaent**(3)	get password adjunct file entry
getpwent	**getpwent**(3V)	get password file entry
getpwnam	**getpwent**(3V)	get password file entry
getpwuid	**getpwent**(3V)	get password file entry
getrpcbyname	**getrpcent**(3N)	get RPC entry
getrpcbynumber	**getrpcent**(3N)	get RPC entry
getrpcent	**getrpcent**(3N)	get RPC entry
gets	**gets**(3S)	get a string from a stream
getsecretkey	**publickey**(3R)	get public or secret key
getservbyname	**getservent**(3N)	get service entry
getservbyport	**getservent**(3N)	get service entry
getservent	**getservent**(3N)	get service entry
getsubopt	**getsubopt**(3)	parse sub options from a string.
gettext	**gettext**(3)	retrieve a message string, get and set text domain
getttyent	**getttyent**(3)	get ttytab file entry
getttynam	**getttyent**(3)	get ttytab file entry
getusershell	**getusershell**(3)	get legal user shells
getw	**getc**(3V)	get character or integer from stream
getwd	**getwd**(3)	get current working directory pathname
gmtime	**ctime**(3V)	convert date and time
grpauth	**pwdauth**(3)	password authentication routines
gsignal	**ssignal**(3)	software signals
gtty	**stty**(3C)	set and get terminal state
hasmntopt	**getmntent**(3)	get file system descriptor file entry
hcreate	**hsearch**(3)	manage hash search tables
hdestroy	**hsearch**(3)	manage hash search tables
host2netname	**secure_rpc**(3N)	library routines for secure remote procedure calls
hsearch	**hsearch**(3)	manage hash search tables
htonl	**byteorder**(3N)	convert values between host and network byte order
htons	**byteorder**(3N)	convert values between host and network byte order
index	**string**(3)	string operations
inet	**inet**(3N)	Internet address manipulation
inet_addr	**inet**(3N)	Internet address manipulation
inet_lnaof	**inet**(3N)	Internet address manipulation
inet_makeaddr	**inet**(3N)	Internet address manipulation
inet_netof	**inet**(3N)	Internet address manipulation
inet_network	**inet**(3N)	Internet address manipulation
inet_ntoa	**inet**(3N)	Internet address manipulation
initgroups	**initgroups**(3)	initialize supplementary group IDs
initstate	**random**(3)	better random number generator

innetgr	**getnetgrent**(3N)	get network group entry
insque	**insque**(3)	insert/remove element from a queue
isalnum	**ctype**(3V)	character classification and conversion macros and functions
isalpha	**ctype**(3V)	character classification and conversion macros and functions
isascii	**ctype**(3V)	character classification and conversion macros and functions
isatty	**ttyname**(3V)	find name of a terminal
iscntrl	**ctype**(3V)	character classification and conversion macros and functions
isdigit	**ctype**(3V)	character classification and conversion macros and functions
isgraph	**ctype**(3V)	character classification and conversion macros and functions
islower	**ctype**(3V)	character classification and conversion macros and functions
isprint	**ctype**(3V)	character classification and conversion macros and functions
ispunct	**ctype**(3V)	character classification and conversion macros and functions
issecure	**issecure**(3)	indicates whether system is running secure
isspace	**ctype**(3V)	character classification and conversion macros and functions
isupper	**ctype**(3V)	character classification and conversion macros and functions
isxdigit	**ctype**(3V)	character classification and conversion macros and functions
itom	**mp**(3X)	multiple precision integer arithmetic
jrand48	**drand48**(3)	generate uniformly distributed pseudo-random numbers
key_decryptsession	**secure_rpc**(3N)	library routines for secure remote procedure calls
key_encryptsession	**secure_rpc**(3N)	library routines for secure remote procedure calls
key_gendes	**secure_rpc**(3N)	library routines for secure remote procedure calls
key_setsecret	**secure_rpc**(3N)	library routines for secure remote procedure calls
kvm_close	**kvm_open**(3K)	specify a kernel to examine
kvm_getcmd	**kvm_getu**(3K)	get the u-area or invocation arguments for a process
kvm_getproc	**kvm_nextproc**(3K)	read system process structures
kvm_getu	**kvm_getu**(3K)	get the u-area or invocation arguments for a process
kvm_nextproc	**kvm_nextproc**(3K)	read system process structures
kvm_nlist	**kvm_nlist**(3K)	get entries from kernel symbol table
kvm_open	**kvm_open**(3K)	specify a kernel to examine
kvm_read	**kvm_read**(3K)	copy data to or from a kernel image or running system
kvm_setproc	**kvm_nextproc**(3K)	read system process structures
kvm_write	**kvm_read**(3K)	copy data to or from a kernel image or running system
l3tol	**l3tol**(3C)	convert between 3-byte integers and long integers
l64a	**a64l**(3)	convert between long integer and base-64 ASCII string
label	**plot**(3X)	graphics interface
lcong48	**drand48**(3)	generate uniformly distributed pseudo-random numbers
ldaclose	**ldclose**(3X)	close a COFF file
ldahread	**ldahread**(3X)	read the archive header of a member of a COFF archive file
ldaopen	**ldopen**(3X)	open a COFF file for reading
ldclose	**ldclose**(3X)	close a COFF file
ldfcn	**ldfcn**(3)	common object file access routines
ldfhread	**ldfhread**(3X)	read the file header of a COFF file
ldgetname	**ldgetname**(3X)	retrieve symbol name for COFF file symbol table entry
ldlinit	**ldlread**(3X)	manipulate line number entries of a COFF file function
ldlitem	**ldlread**(3X)	manipulate line number entries of a COFF file function
ldlread	**ldlread**(3X)	manipulate line number entries of a COFF file function
ldlseek	**ldlseek**(3X)	seek to line number entries of a section of a COFF file
ldnlseek	**ldlseek**(3X)	seek to line number entries of a section of a COFF file
ldnrseek	**ldrseek**(3X)	seek to relocation entries of a section of a COFF file
ldnshread	**ldshread**(3X)	read an indexed/named section header of a COFF file
ldnsseek	**ldsseek**(3X)	seek to an indexed/named section of a COFF file
ldohseek	**ldohseek**(3X)	seek to the optional file header of a COFF file
ldopen	**ldopen**(3X)	open a COFF file for reading

ldrseek	**ldrseek**(3X)	seek to relocation entries of a section of a COFF file
ldshread	**ldshread**(3X)	read an indexed/named section header of a COFF file
ldsseek	**ldsseek**(3X)	seek to an indexed/named section of a COFF file
ldtbindex	**ldtbindex**(3X)	compute the index of a symbol table entry of a COFF file
ldtbread	**ldtbread**(3X)	read an indexed symbol table entry of a COFF file
ldtbseek	**ldtbseek**(3X)	seek to the symbol table of a COFF file
lfind	**lsearch**(3)	linear search and update
line	**plot**(3X)	graphics interface
linemod	**plot**(3X)	graphics interface
localdtconv	**localdtconv**(3)	get date and time formatting conventions
localeconv	**localeconv**(3)	get numeric and monetary formatting conventions
localtime	**ctime**(3V)	convert date and time
lockf	**lockf**(3)	record locking on files
longjmp	**setjmp**(3V)	non-local goto
lrand48	**drand48**(3)	generate uniformly distributed pseudo-random numbers
lsearch	**lsearch**(3)	linear search and update
ltol3	**l3tol**(3C)	convert between 3-byte integers and long integers
madd	**mp**(3X)	multiple precision integer arithmetic
madvise	**madvise**(3)	provide advice to VM system
malloc	**malloc**(3V)	memory allocator
malloc_debug	**malloc**(3V)	memory allocator
malloc_verify	**malloc**(3V)	memory allocator
mallocmap	**malloc**(3V)	memory allocator
mblen	**mblen**(3)	multibyte character handling
mbstowcs	**mblen**(3)	multibyte character handling
mbtowc	**mblen**(3)	multibyte character handling
mcmp	**mp**(3X)	multiple precision integer arithmetic
mdiv	**mp**(3X)	multiple precision integer arithmetic
memalign	**malloc**(3V)	memory allocator
memccpy	**memory**(3)	memory operations
memchr	**memory**(3)	memory operations
memcmp	**memory**(3)	memory operations
memcpy	**memory**(3)	memory operations
memory	**memory**(3)	memory operations
memset	**memory**(3)	memory operations
mfree	**mp**(3X)	multiple precision integer arithmetic
min	**mp**(3X)	multiple precision integer arithmetic
mkstemp	**mktemp**(3)	make a unique file name
mktemp	**mktemp**(3)	make a unique file name
mlock	**mlock**(3)	lock (or unlock) pages in memory
mlockall	**mlockall**(3)	lock (or unlock) address space
moncontrol	**monitor**(3)	prepare execution profile
monitor	**monitor**(3)	prepare execution profile
monstartup	**monitor**(3)	prepare execution profile
mout	**mp**(3X)	multiple precision integer arithmetic
move	**plot**(3X)	graphics interface
mp	**mp**(3X)	multiple precision integer arithmetic
mrand48	**drand48**(3)	generate uniformly distributed pseudo-random numbers
msub	**mp**(3X)	multiple precision integer arithmetic
msync	**msync**(3)	synchronize memory with physical storage
mtox	**mp**(3X)	multiple precision integer arithmetic
mult	**mp**(3X)	multiple precision integer arithmetic
munlock	**mlock**(3)	lock (or unlock) pages in memory

munlockall	**mlockall**(3)	lock (or unlock) address space
ndbm	**ndbm**(3)	data base subroutines
netname2host	**secure_rpc**(3N)	library routines for secure remote procedure calls
netname2user	**secure_rpc**(3N)	library routines for secure remote procedure calls
nextkey	**dbm**(3X)	data base subroutines
nice	**nice**(3V)	change nice value of a process
nl_init	**setlocale**(3V)	set international environment
nl_langinfo	**nl_langinfo**(3C)	language information
nlist	**nlist**(3V)	get entries from symbol table
nrand48	**drand48**(3)	generate uniformly distributed pseudo-random numbers
ntohl	**byteorder**(3N)	convert values between host and network byte order
ntohs	**byteorder**(3N)	convert values between host and network byte order
on_exit	**on_exit**(3)	name termination handler
opendir	**directory**(3V)	directory operations
openlog	**syslog**(3)	control system log
openpl	**plot**(3X)	graphics interface
optarg	**getopt**(3)	get option letter from argument vector
optind	**getopt**(3)	get option letter from argument vector
passwd2des	**xcrypt**(3R)	hex encryption and utility routines
pause	**pause**(3V)	stop until signal
pclose	**popen**(3S)	open or close a pipe (for I/O) from or to a process
perror	**perror**(3)	system error messages
plock	**plock**(3)	lock process, text, or data segment in memory
plot	**plot**(3X)	graphics interface
point	**plot**(3X)	graphics interface
popen	**popen**(3S)	open or close a pipe (for I/O) from or to a process
pow	**mp**(3X)	multiple precision integer arithmetic
printf	**printf**(3V)	formatted output conversion
prof	**prof**(3)	profile within a function
psignal	**psignal**(3)	system signal messages
publickey	**publickey**(3R)	get public or secret key
putc	**putc**(3S)	put character or word on a stream
putchar	**putc**(3S)	put character or word on a stream
putenv	**putenv**(3)	change or add value to environment
putpwent	**putpwent**(3)	write password file entry
puts	**puts**(3S)	put a string on a stream
putw	**putc**(3S)	put character or word on a stream
pwdauth	**pwdauth**(3)	password authentication routines
qsort	**qsort**(3)	quicker sort
rand	**rand**(3V)	simple random number generator
random	**random**(3)	better random number generator
rcmd	**rcmd**(3N)	routines for returning a stream to a remote command
re_comp	**regex**(3)	regular expression handler
re_exec	**regex**(3)	regular expression handler
readdir	**directory**(3V)	directory operations
realloc	**malloc**(3V)	memory allocator
realpath	**realpath**(3)	return the canonicalized absolute pathname
regex	**regex**(3)	regular expression handler
regexp	**regexp**(3)	regular expression compile and match routines
registerrpc	**rpc_svc_calls**(3N)	library routines for registerring servers
remexportent	**exportent**(3)	get exported file system information
remque	**insque**(3)	insert/remove element from a queue
res_init	**resolver**(3)	resolver routines

res_mkquery	**resolver**(3)	resolver routines
res_send	**resolver**(3)	resolver routines
resolver	**resolver**(3)	resolver routines
rewind	**fseek**(3S)	reposition a stream
rewinddir	**directory**(3V)	directory operations
rexec	**rexec**(3N)	return stream to a remote command
rindex	**string**(3)	string operations
rpc	**rpc**(3N)	library routines for remote procedure calls
rpc_createrr	**rpc_clnt_create**(3N)	library routines creating and manipulating CLIENT handles
rpow	**mp**(3X)	multiple precision integer arithmetic
rresvport	**rcmd**(3N)	routines for returning a stream to a remote command
rtime	**rtime**(3N)	get remote time
ruserok	**rcmd**(3N)	routines for returning a stream to a remote command
scandir	**scandir**(3)	scan a directory
scanf	**scanf**(3V)	formatted input conversion
seconvert	**econvert**(3)	output conversion
seed48	**drand48**(3)	generate uniformly distributed pseudo-random numbers
seekdir	**directory**(3V)	directory operations
setac	**getacinfo**(3)	get audit control file information
setbuf	**setbuf**(3V)	assign buffering to a stream
setbuffer	**setbuf**(3V)	assign buffering to a stream
setegid	**setuid**(3V)	set user and group ID
seteuid	**setuid**(3V)	set user and group ID
setexportent	**exportent**(3)	get exported file system information
setfsent	**getfsent**(3)	get file system descriptor file entry
setgid	**setuid**(3V)	set user and group ID
setgraent	**getgraent**(3)	get group adjunct file entry
setgrent	**getgrent**(3V)	get group file entry
sethostent	**gethostent**(3N)	get network host entry
setjmp	**setjmp**(3V)	non-local goto
setkey	**crypt**(3)	password and data encryption
setlinebuf	**setbuf**(3V)	assign buffering to a stream
setlocale	**setlocale**(3V)	set international environment
setlogmask	**syslog**(3)	control system log
setmntent	**getmntent**(3)	get file system descriptor file entry
setnetent	**getnetent**(3N)	get network entry
setnetgrent	**getnetgrent**(3N)	get network group entry
setprotoent	**getprotoent**(3N)	get protocol entry
setpwaent	**getpwaent**(3)	get password adjunct file entry
setpwent	**getpwent**(3V)	get password file entry
setpwfile	**getpwent**(3V)	get password file entry
setrgid	**setuid**(3V)	set user and group ID
setrpcent	**getrpcent**(3N)	get RPC entry
setruid	**setuid**(3V)	set user and group ID
setservent	**getservent**(3N)	get service entry
setstate	**random**(3)	better random number generator
setttyent	**getttyent**(3)	get ttytab file entry
setuid	**setuid**(3V)	set user and group ID
setusershell	**getusershell**(3)	get legal user shells
setvbuf	**setbuf**(3V)	assign buffering to a stream
sfconvert	**econvert**(3)	output conversion
sgconvert	**econvert**(3)	output conversion
sigaction	**sigaction**(3V)	examine and change signal action

sigaddset	**sigsetops**(3V)	manipulate signal sets
sigdelset	**sigsetops**(3V)	manipulate signal sets
sigemptyset	**sigsetops**(3V)	manipulate signal sets
sigfillset	**sigsetops**(3V)	manipulate signal sets
sigfpe	**sigfpe**(3)	signal handling for specific SIGFPE codes
siginterrupt	**siginterrupt**(3V)	allow signals to interrupt system calls
sigismember	**sigsetops**(3V)	manipulate signal sets
siglongjmp	**setjmp**(3V)	non-local goto
signal	**signal**(3V)	simplified software signal facilities
sigsetjmp	**setjmp**(3V)	non-local goto
sigsetops	**sigsetops**(3V)	manipulate signal sets
single_to_decimal	**floating_to_decimal**(3)	convert floating-point value to decimal record
sleep	**sleep**(3V)	suspend execution for interval
space	**plot**(3X)	graphics interface
sprintf	**printf**(3V)	formatted output conversion
srand48	**drand48**(3)	generate uniformly distributed pseudo-random numbers
srand	**rand**(3V)	simple random number generator
srandom	**random**(3)	better random number generator
sscanf	**scanf**(3V)	formatted input conversion
ssignal	**ssignal**(3)	software signals
stdio	**stdio**(3V)	standard buffered input/output package
store	**dbm**(3X)	data base subroutines
strcasecmp	**string**(3)	string operations
strcat	**string**(3)	string operations
strchr	**string**(3)	string operations
strcmp	**string**(3)	string operations
strcoll	**strcoll**(3)	compare or transform strings using collating information
strcpy	**string**(3)	string operations
strcspn	**string**(3)	string operations
strdup	**string**(3)	string operations
strftime	**ctime**(3V)	convert date and time
string_to_decimal	**string_to_decimal**(3)	parse characters into decimal record
strlen	**string**(3)	string operations
strncasecmp	**string**(3)	string operations
strncat	**string**(3)	string operations
strncmp	**string**(3)	string operations
strncpy	**string**(3)	string operations
strpbrk	**string**(3)	string operations
strptime	**ctime**(3V)	convert date and time
strrchr	**string**(3)	string operations
strspn	**string**(3)	string operations
strstr	**string**(3)	string operations
strtod	**strtod**(3)	convert string to double-precision number
strtok	**string**(3)	string operations
strtol	**strtol**(3)	convert string to integer
strxfrm	**strcoll**(3)	compare or transform strings using collating information
stty	**stty**(3C)	set and get terminal state
svc_destroy	**rpc_svc_create**(3N)	library routines for dealing with the creation of server handles
svc_fds	**rpc_svc_reg**(3N)	library routines for RPC servers
svc_fdset	**rpc_svc_reg**(3N)	library routines for RPC servers
svc_freeargs	**rpc_svc_reg**(3N)	library routines for RPC servers
svc_getargs	**rpc_svc_reg**(3N)	library routines for RPC servers
svc_getcaller	**rpc_svc_reg**(3N)	library routines for RPC servers

svc_getreq	**rpc_svc_reg**(3N)	library routines for RPC servers
svc_getreqset	**rpc_svc_reg**(3N)	library routines for RPC servers
svc_register	**rpc_svc_calls**(3N)	library routines for registerring servers
svc_run	**rpc_svc_reg**(3N)	library routines for RPC servers
svc_sendreply	**rpc_svc_reg**(3N)	library routines for RPC servers
svc_unregister	**rpc_svc_calls**(3N)	library routines for registerring servers
svcerr_auth	**rpc_svc_err**(3N)	library routines for server side remote procedure call errors
svcerr_decode	**rpc_svc_err**(3N)	library routines for server side remote procedure call errors
svcerr_noproc	**rpc_svc_err**(3N)	library routines for server side remote procedure call errors
svcerr_noprog	**rpc_svc_err**(3N)	library routines for server side remote procedure call errors
svcerr_progvers	**rpc_svc_err**(3N)	library routines for server side remote procedure call errors
svcerr_systemerr	**rpc_svc_err**(3N)	library routines for server side remote procedure call errors
svcerr_weakauth	**rpc_svc_err**(3N)	library routines for server side remote procedure call errors
svcfd_create	**rpc_svc_create**(3N)	library routines for dealing with the creation of server handles
svcraw_create	**rpc_svc_create**(3N)	library routines for dealing with the creation of server handles
svctcp_create	**rpc_svc_create**(3N)	library routines for dealing with the creation of server handles
svcudp_bufcreate	**rpc_svc_create**(3N)	library routines for dealing with the creation of server handles
swab	**swab**(3)	swap bytes
sys_siglist	**psignal**(3)	system signal messages
syslog	**syslog**(3)	control system log
system	**system**(3)	issue a shell command
t_accept	**t_accept**(3N)	accept a connect request
t_alloc	**t_alloc**(3N)	allocate a library structure
t_bind	**t_bind**(3N)	bind an address to a transport endpoint
t_close	**t_close**(3N)	close a transport endpoint
t_connect	**t_connect**(3N)	establish a connection with another transport user
t_error	**t_error**(3N)	produce error message
t_free	**t_free**(3N)	free a library structure
t_getinfo	**t_getinfo**(3N)	get protocol-specific service information
t_getstate	**t_getstate**(3N)	get the current state
t_listen	**t_listen**(3N)	listen for a connect request
t_look	**t_look**(3N)	look at the current event on a transport endpoint
t_open	**t_open**(3N)	establish a transport endpoint
t_optmgmt	**t_optmgmt**(3N)	manage options for a transport endpoint
t_rcv	**t_rcv**(3N)	receive normal or expedited data sent over a connection
t_rcvconnect	**t_rcvconnect**(3N)	receive the confirmation from a connect request
t_rcvdis	**t_rcvdis**(3N)	retrieve information from disconnect
t_rcvrel	**t_rcvrel**(3N)	acknowledge receipt of an orderly release indication
t_rcvudata	**t_rcvudata**(3N)	receive a data unit
t_rcvuderr	**t_rcvuderr**(3N)	receive a unit data error indication
t_snd	**t_snd**(3N)	send normal or expedited data over a connection
t_snddis	**t_snddis**(3N)	send user-initiated disconnect request
t_sndrel	**t_sndrel**(3N)	initiate an orderly release
t_sndudata	**t_sndudata**(3N)	send a data unit
t_sync	**t_sync**(3N)	synchronize transport library
t_unbind	**t_unbind**(3N)	disable a transport endpoint
tcdrain	**termios**(3V)	terminal control functions
tcflow	**termios**(3V)	terminal control functions
tcflush	**termios**(3V)	terminal control functions
tcgetattr	**termios**(3V)	terminal control functions
tcgetpgrp	**tcgetpgrp**(3V)	get, set foreground process group ID
tcsendbreak	**termios**(3V)	terminal control functions
tcsetattr	**termios**(3V)	terminal control functions

tcsetpgrp	**tcgetpgrp**(3V)	get, set foreground process group ID
tdelete	**tsearch**(3)	manage binary search trees
telldir	**directory**(3V)	directory operations
tempnam	**tmpnam**(3S)	create a name for a temporary file
termcap	**termcap**(3X)	terminal independent operation routines
termios	**termios**(3V)	terminal control functions
textdomain	**gettext**(3)	retrieve a message string, get and set text domain
tfind	**tsearch**(3)	manage binary search trees
tgetent	**termcap**(3X)	terminal independent operation routines
tgetflag	**termcap**(3X)	terminal independent operation routines
tgetnum	**termcap**(3X)	terminal independent operation routines
tgetstr	**termcap**(3X)	terminal independent operation routines
tgoto	**termcap**(3X)	terminal independent operation routines
time	**time**(3V)	get date and time
timegm	**ctime**(3V)	convert date and time
timelocal	**ctime**(3V)	convert date and time
times	**times**(3V)	get process times
timezone	**timezone**(3C)	get time zone name given offset from GMT
tmpfile	**tmpfile**(3S)	create a temporary file
tmpnam	**tmpnam**(3S)	create a name for a temporary file
toascii	**ctype**(3V)	character classification and conversion macros and functions
tolower	**ctype**(3V)	character classification and conversion macros and functions
toupper	**ctype**(3V)	character classification and conversion macros and functions
tputs	**termcap**(3X)	terminal independent operation routines
tsearch	**tsearch**(3)	manage binary search trees
ttyname	**ttyname**(3V)	find name of a terminal
ttyslot	**ttyslot**(3V)	find the slot in the utmp file of the current process
twalk	**tsearch**(3)	manage binary search trees
tzset	**ctime**(3V)	convert date and time
tzsetwall	**ctime**(3V)	convert date and time
ualarm	**ualarm**(3)	schedule signal after interval in microseconds
ulimit	**ulimit**(3C)	get and set user limits
ungetc	**ungetc**(3S)	push character back into input stream
user2netname	**secure_rpc**(3N)	library routines for secure remote procedure calls
usleep	**usleep**(3)	suspend execution for interval in microseconds
utime	**utime**(3V)	set file times
valloc	**malloc**(3V)	memory allocator
values	**values**(3)	machine-dependent values
varargs	**varargs**(3)	handle variable argument list
vfprintf	**vprintf**(3V)	print formatted output of a varargs argument list
vlimit	**vlimit**(3C)	control maximum system resource consumption
vprintf	**vprintf**(3V)	print formatted output of a varargs argument list
vsprintf	**vprintf**(3V)	print formatted output of a varargs argument list
vsyslog	**vsyslog**(3)	log message with a varargs argument list
vtimes	**vtimes**(3C)	get information about resource utilization
wcstombs	**mblen**(3)	multibyte character handling
wctomb	**mblen**(3)	multibyte character handling
xcrypt	**xcrypt**(3R)	hex encryption and utility routines
xdecrypt	**xcrypt**(3R)	hex encryption and utility routines
xdr	**xdr**(3N)	library routines for external data representation
xdr_accepted_reply	**rpc_xdr**(3N)	XDR library routines for remote procedure calls
xdr_array	**xdr_complex**(3N)	library routines for translating complex data types
xdr_authunix_parms	**rpc_xdr**(3N)	XDR library routines for remote procedure calls

xdr_bool	**xdr_simple**(3N)	library routines for translating simple data types
xdr_bytes	**xdr_complex**(3N)	library routines for translating complex data types
xdr_callhdr	**rpc_xdr**(3N)	XDR library routines for remote procedure calls
xdr_callmsg	**rpc_xdr**(3N)	XDR library routines for remote procedure calls
xdr_char	**xdr_simple**(3N)	library routines for translating simple data types
xdr_destroy	**xdr_create**(3N)	library routines for XDR stream creation
xdr_double	**xdr_simple**(3N)	library routines for translating simple data types
xdr_enum	**xdr_simple**(3N)	library routines for translating simple data types
xdr_float	**xdr_simple**(3N)	library routines for translating simple data types
xdr_free	**xdr_simple**(3N)	library routines for translating simple data types
xdr_getpos	**xdr_admin**(3N)	library routines for management of the XDR stream
xdr_inline	**xdr_admin**(3N)	library routines for management of the XDR stream
xdr_int	**xdr_simple**(3N)	library routines for translating simple data types
xdr_long	**xdr_simple**(3N)	library routines for translating simple data types
xdr_opaque	**xdr_complex**(3N)	library routines for translating complex data types
xdr_opaque_auth	**rpc_xdr**(3N)	XDR library routines for remote procedure calls
xdr_pamp	**portmap**(3N)	library routines for RPC bind service
xdr_pmaplist	**portmap**(3N)	library routines for RPC bind service
xdr_pointer	**xdr_complex**(3N)	library routines for translating complex data types
xdr_reference	**xdr_complex**(3N)	library routines for translating complex data types
xdr_rejected_reply	**rpc_xdr**(3N)	XDR library routines for remote procedure calls
xdr_replymsg	**rpc_xdr**(3N)	XDR library routines for remote procedure calls
xdr_setpos	**xdr_admin**(3N)	library routines for management of the XDR stream
xdr_short	**xdr_simple**(3N)	library routines for translating simple data types
xdr_string	**xdr_complex**(3N)	library routines for translating complex data types
xdr_u_char	**xdr_simple**(3N)	library routines for translating simple data types
xdr_u_int	**xdr_simple**(3N)	library routines for translating simple data types
xdr_u_long	**xdr_simple**(3N)	library routines for translating simple data types
xdr_u_short	**xdr_simple**(3N)	library routines for translating simple data types
xdr_union	**xdr_complex**(3N)	library routines for translating complex data types
xdr_vector	**xdr_complex**(3N)	library routines for translating complex data types
xdr_void	**xdr_simple**(3N)	library routines for translating simple data types
xdr_wrapstring	**xdr_complex**(3N)	library routines for translating complex data types
xdrmem_create	**xdr_create**(3N)	library routines for XDR stream creation
xdrrec_create	**xdr_create**(3N)	library routines for XDR stream creation
xdrrec_endofrecord	**xdr_admin**(3N)	library routines for management of the XDR stream
xdrrec_eof	**xdr_admin**(3N)	library routines for management of the XDR stream
xdrrec_readbytes	**xdr_admin**(3N)	library routines for management of the XDR stream
xdrrec_skiprecord	**xdr_admin**(3N)	library routines for management of the XDR stream
xdrstdio_create	**xdr_create**(3N)	library routines for XDR stream creation
xencrypt	**xcrypt**(3R)	hex encryption and utility routines
xprt_register	**rpc_svc_calls**(3N)	library routines for registerring servers
xprt_unregister	**rpc_svc_calls**(3N)	library routines for registerring servers
xtom	**mp**(3X)	multiple precision integer arithmetic
yp_all	**ypclnt**(3N)	NIS client interface
yp_bind	**ypclnt**(3N)	NIS client interface
yp_first	**ypclnt**(3N)	NIS client interface
yp_get_default_domain	**ypclnt**(3N)	NIS client interface
yp_master	**ypclnt**(3N)	NIS client interface
yp_match	**ypclnt**(3N)	NIS client interface
yp_next	**ypclnt**(3N)	NIS client interface
yp_order	**ypclnt**(3N)	NIS client interface
yp_unbind	**ypclnt**(3N)	NIS client interface

yp_update	**ypupdate**(3N)	changes NIS information
ypclnt	**ypclnt**(3N)	NIS client interface
yperr_string	**ypclnt**(3N)	NIS client interface
ypprot_err	**ypclnt**(3N)	NIS client interface

NAME

a641, l64a – convert between long integer and base-64 ASCII string

SYNOPSIS

```
long a641(s)
char *s;

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

DESCRIPTION

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

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

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

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

BUGS

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

NAME

abort – generate a fault

SYNOPSIS

abort( )

DESCRIPTION

abort( ) first closes all open files if possible, then sends an IOT signal to the process. This signal usually results in termination with a core dump, which may be used for debugging.

It is possible for **abort( )** to return control if **SIGIOT** is caught or ignored, in which case the value returned is that of the **kill**(2V) system call.

SEE ALSO

adb(1), **exit**(2V), **kill**(2V), **signal**(3V)

DIAGNOSTICS

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

NAME

abs – integer absolute value

SYNOPSIS

```
abs(i)
int i;
```

DESCRIPTION

abs( ) returns the absolute value of its integer operand.

SEE ALSO

ieee_functions(3M) for **fabs**( )

BUGS

Applying the **abs**() function to the most negative integer generates a result which is the most negative integer. That is, **abs(0x80000000)** returns **0x80000000** as a result.

NAME

aiocancel – cancel an asynchronous operation

SYNOPSIS

#include <sys/asynch.h>

int aiocancel(resultp)
aio_result_t *resultp;

DESCRIPTION

aiocancel( ) cancels the asynchronous operation associated with the result buffer pointed to by *resultp*. It may not be possible to immediately cancel an operation which is in progress and in this case, **aiocancel( )** will not wait to cancel it.

Upon successful completion, **aiocancel( )** will return 0 and the requested operation will be canceled. The application will not receive the **SIGIO** completion signal for an asynchronous operation which is successfully canceled.

RETURN VALUES

aiocancel( ) returns:

0	on success.
−1	on failure and sets **errno** to indicate the error.

ERRORS

aiocancel( ) will fail if any of the following are true:

EACCES The parameter *resultp* does not correspond to an outstanding asynchronous operation.

The operation could not be cancelled.

EFAULT The parameter *resultp* points to an address that is outside of the address space of the requesting process.

SEE ALSO

aioread(3), aiowait(3)

NAME

aioread, aiowrite – asynchronous I/O operations

SYNOPSIS

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

int aioread(fd, bufp, bufs, offset, whence, resultp)
int fd;
char *bufp;
int bufs;
int offset;
int whence;
aio_result_t *resultp;

int aiowrite(fd, bufp, bufs, offset, whence, resultp)
int fd;
char *bufp;
int bufs;
int offset;
int whence;
aio_result_t *resultp;
```

DESCRIPTION

aioread() initiates one asynchronous **read(2V)** and returns control to the calling program. The **read()** continues concurrently with other activity of the process. An attempt is made to read *bufs* bytes of data from the object referenced by the descriptor *fd* into the buffer pointed to by *bufp*.

aiowrite() initiates one asynchronous **write(2V)** and returns control to the calling program. The **write()** continues concurrently with other activity of the process. An attempt is made to write *bufs* bytes of data from the buffer pointed to by *bufp* to the object referenced by the descriptor *fd*.

On objects capable of seeking, the I/O operation starts at the position specified by *whence* and *offset*. These parameters have the same meaning as the corresponding parameters to the **lseek(2V)** function. On objects not capable of seeking the I/O operation always start from the current position and the parameters *whence* and *offset* are ignored. The seek pointer for objects capable of seeking is not updated by **aioread()** or **aiowrite()**. Sequential asynchronous operations on these devices must be managed by the application using the *whence* and *offset* parameters.

The result of the asynchronous operation is stored in the structure pointed to by *resultp*:

```
int aio_return;        /* return value of read() or write() */
int aio_errno;         /* value of errno for read() or write() */
```

Upon completion of the operation both *aio_return* and *aio_errno* are set to reflect the result of the operation. AIO_INPROGRESS is not a value used by the system so the client may detect a change in state by initializing *aio_return* to this value.

Notification of the completion of an asynchronous I/O operation may be obtained synchronously through the **aiowait(3)** function, or asynchronously through the signal mechanism. Asynchronous notification is accomplished by generating the SIGIO signal. The delivery of this instance of the SIGIO signal is reliable in that a signal delivered while the handler is executing is not lost. If the client ensures that **aiowait(3)** returns nothing (using a polling timeout) before returning from the signal handler, no asynchronous I/O notifications are lost. The **aiowait(3)** function is the only way to dequeue an asynchronous notification. Note: SIGIO may have several meanings simultaneously: for example, that a descriptor generated SIGIO and an asynchronous operation completed. Further, issuing an asynchronous request successfully guarantees that space exists to queue the completion notification.

close(2V), **exit(2V)** and **execve(2V)** will block until all pending asynchronous I/O operations can be cancelled by the system.

It is an error to use the same result buffer in more than one outstanding request. These structures may only be reused after the system has completed the operation.

RETURN VALUES

aioread( ) and **aiowrite**( ) return:

0 on success.

–1 on failure and set **errno** to indicate the error.

ERRORS

EBADF *fd* is not a valid file descriptor open for reading.

EFAULT At least one of *bufp* or *resultp* points to an address out side the address space of the requesting process.

EINVAL The parameter *resultp* is currently being used by an outstanding asynchronous request.

EPROCLIM The number of asynchronous requests that the system can handle at any one time has been exceeded

SEE ALSO

close(2V), **execve**(2V), **exit**(2V), **lseek**(2V), **open**(2V), **read**(2V), **sigvec**(2), **write**(2V), **aiocancel**(3), **aiowait**(3)

NAME

aiowait – wait for completion of asynchronous I/O operation

SYNOPSIS

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

aio_result_t *aiowait(timeout)
struct timeval *timeout;
```

DESCRIPTION

aiowait( ) suspends the calling process until one of its outstanding asynchronous I/O operations completes. This provides a synchronous method of notification.

If *timeout* is a non-zero pointer, it specifies a maximum interval to wait for the completion of an asynchronous I/O operation. If *timeout* is a zero pointer, then **aiowait( )** blocks indefinitely. To effect a poll, the *timeout* parameter should be non-zero, pointing to a zero-valued *timeval* structure. The *timeval* structure is defined in **<sys/time.h>** as:

```
struct timeval {
        long tv_sec;            /* seconds          */
        long tv_usec;           /* and microseconds */
};
```

NOTES

aiowait( ) is the only way to dequeue an asynchronous notification. It may be used either inside a **SIGIO** signal handler or in the main program. Note: one **SIGIO** signal may represent several queued events.

RETURN VALUES

On success, **aiowait( )** returns a pointer to the result structure used when the completed asynchronous I/O operation was requested. On failure, it returns –1 and sets **errno** to indicate the error. **aiowait( ) returns** 0 if the time limit expires.

ERRORS

EFAULT	*timeout* points to an address outside the address space of the requesting process.
EINTR	A signal was delivered before an asynchronous I/O operation completed.
	The time limit expired.
EINVAL	There are no outstanding asynchronous I/O requests.

SEE ALSO

aiocancel(3), **aioread**(3)

NAME

alarm – schedule signal after specified time

SYNOPSIS

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

DESCRIPTION

alarm() sends the signal **SIGALRM** (see **sigvec(2)**), to the invoking process after *seconds* seconds. Unless caught or ignored, the signal terminates the process.

alarm() requests are not stacked; successive calls reset the alarm clock. If the argument is 0, any **alarm()** request is canceled. Because of scheduling delays, resumption of execution of when the signal is caught may be delayed an arbitrary amount. The longest specifiable delay time is 2147483647 seconds.

The return value is the amount of time previously remaining in the alarm clock.

SEE ALSO

sigpause(2V), **sigvec(2)**, **signal(3V)**, **sleep(3V)**, **ualarm(3)**, **usleep(3)**

WARNINGS

alarm() is slightly incompatible with the default version of **sleep(3V)**. The alarm signal is not sent when one would expect for programs that wait one second of clock time between successive calls to **sleep()**. Each **sleep()** call postpones the alarm signal that would have been sent during the requested sleep period for one second. Use System V **sleep(3V)** to avoid this delay.

NAME

assert – program verification

SYNOPSIS

#include <assert.h>

assert(expression)

DESCRIPTION

assert( ) is a macro that indicates *expression* is expected to be true at this point in the program. If *expression* is false (0), it displays a diagnostic message on the standard output and exits (see **exit**(2V)). Compiling with the **cc**(1V) option **–DNDEBUG**, or placing the preprocessor control statement

#define NDEBUG

before the "**#include <assert.h>**" statement effectively deletes **assert**( ) from the program.

SYSTEM V DESCRIPTION

The System V version of **assert**( ) calls **abort**(3) rather than **exit**( ).

SEE ALSO

cc(1V), **exit**(2V), **abort**(3)

DIAGNOSTICS

Assertion failed: file *f* **line** *n*

The expression passed to the **assert**( ) statement at line *n* of source file *f* was false.

SYSTEM V DIAGNOSTICS

Assertion failed: *expression*, **file** *f*, **line** *n*

The *expression* passed to the **assert**( ) statement at line *n* of source file *f* was false.

NAME

audit_args, audit_text – produce text audit message

SYNOPSIS

```
#include <sys/label.h>
#include <sys/audit.h>

audit_args(event, argc, argv)
int event;
int argc;
char **argv;

audit_text(event, error, retval, argc, argv)
int event;
int error;
int retval;
int argc;
char **argv;
```

DESCRIPTION

These functions provide text interfaces to the **audit**(2) system call. In both calls, the *event* parameter identifies the event class of the action, and *argc* is the number of strings found in the vector *argv*. The **error** parameter is used to determine the failure or success of the audited operation. A negative value is always audited. A zero value is audited as a successful event. A positive value is audited as an event failure. The *retval* parameter is the return value or exit code that the invoking program will have.

audit_args( ) is equivalent to **audit_text**( ) with **error** and *retval* parameters of –1.

SEE ALSO

audit(2)

NAME

bindresvport – bind a socket to a privileged IP port

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>

int bindresvport(sd, sin)
int sd;
struct sockaddr_in *sin;
```

DESCRIPTION

bindresvport( ) is used to bind a socket descriptor to a privileged IP port, that is, a port number in the range 0-1023. The routine returns 0 if it is successful, otherwise −1 is returned and **errno** set to reflect the cause of the error. This routine differs with **rresvport** (see **rcmd**(3N)) in that this works for any IP socket, whereas **rresvport( )** only works for TCP.

Only root can bind to a privileged port; this call will fail for any other users.

SEE ALSO

rcmd(3N)

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, in which case it prints out the string and its length, or it 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)
struct node *node1, *node2;
{
        return strcmp(node1->string, 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 declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

SEE ALSO

hsearch(3), **lsearch**(3), **qsort**(3), **tsearch**(3)

DIAGNOSTICS

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

NAME

bstring, bcopy, bcmp, bzero, ffs – bit and byte string operations

SYNOPSIS

```
void
bcopy(b1, b2, length)
char *b1, *b2;
int length;

int bcmp(b1, b2, length)
char *b1, *b2;
int length;

void
bzero(b, length)
char *b;
int length;

int ffs(i)
int i;
```

DESCRIPTION

The functions **bcopy**, **bcmp**, and **bzero( )** operate on variable length strings of bytes. They do not check for null bytes as the routines in **string**(3) do.

bcopy( ) copies *length* bytes from string *b1* to the string *b2*. Overlapping strings are handled correctly.

bcmp( ) compares byte string *b1* against byte string *b2*, returning zero if they are identical, non-zero otherwise. Both strings are assumed to be *length* bytes long. **bcmp( )** of length zero bytes always returns zero.

bzero( ) places *length* 0 bytes in the string *b*.

ffs( ) finds the first bit set in the argument passed it and returns the index of that bit. Bits are numbered starting at 1 from the right. A return value of zero indicates that the value passed is zero.

NOTES

The **bcmp( )** and **bcopy( )** routines take parameters backwards from **strcmp( )** and **strcpy( )**.

SEE ALSO

string(3)

NAME

byteorder, htonl, htons, ntohl, ntohs – convert values between host and network byte order

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>

netlong = htonl(hostlong);
u_long netlong, hostlong;

netshort = htons(hostshort);
u_short netshort, hostshort;

hostlong = ntohl(netlong);
u_long hostlong, netlong;

hostshort = ntohs(netshort);
u_short hostshort, netshort;
```

DESCRIPTION

These routines convert 16 and 32 bit quantities between network byte order and host byte order. On Sun-2, Sun-3 and Sun-4 systems, these routines are defined as NULL macros in the include file **<netinet/in.h>**. On Sun386i systems, these routines are functional since its host byte order is different from network byte order.

These routines are most often used in conjunction with Internet addresses and ports as returned by **gethostent**(3N) and **getservent**(3N).

SEE ALSO

gethostent(3N), **getservent**(3N)

NAME

catgets, catgetmsg – get message from a message catalog

SYNOPSIS

```
#include <nl_types.h>

char *catgets(catd, set_num, msg_num, s)
nl_catd catd;
int set_num, msg_num;
char *s;

char *catgetmsg(catd, set_num, msg_num, buf, buflen)
nl_catd catd;
int set_num;
int msg_num;
int buflen;
```

DESCRIPTION

catgets( ) reads the message *msg_num*, in set *set_num*, from the message catalog identified by *catd*. *catd* is a catalog descriptor returned from an earlier call to **catopen**(3C). *s* points to a default message string which will be returned by **catgets( )** if the identified message catalog is not currently available. The message-text is contained in an internal buffer area and should be copied by the application if it is to be saved or re-used after further calls to **catgets( )**.

catgetmsg( ) attempts to read up to *buflen* −1 bytes of a message string into the area pointed to by *buf*. *buflen* is an integer value containing the size in bytes of *buf*. The return string is always terminated with a null byte.

RETURN VALUES

On success, **catgets( )** returns a pointer to an internal buffer area containing the null-terminated message string. **catgets( )** returns a pointer to *s* if it fails because the message catalog specified by *catd* is not currently available. Otherwise, **catgets( )** returns a pointer to an empty string if the message catalog is available but does not contain the specified message.

On success, **catgetmsg( )** returns a pointer to the message string in *buf*. If *catd* is invalid or if *set_num* or *msg_num* is not in the message catalog, **catgetmsg( )** returns a pointer to an empty string.

SEE ALSO

catopen(3C), **locale**(5)

NAME

catopen, catclose – open/close a message catalog

SYNOPSIS

#include <nl_types.h>

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

int catclose(catd)
nl_catd catd;

DESCRIPTION

catopen( ) opens a message catalog and returns a catalog descriptor. *name* specifies the name of the message catalog to be opened. If *name* contains a '/' then *name* specifies a pathname for the message catalog. Otherwise, the environment variable **NLSPATH** is used with *name* substituted for **%N** (see **locale**(5)). If **NLSPATH** does not exist in the environment, or if a message catalog cannot be opened in any of the paths specified by **NLSPATH**, the **/etc/locale/LC_MESSAGES/***locale* directory is searched for a message catalog with filename *name*, followed by the **/usr/share/lib/locale/LC_MESSAGES/***locale* directory. In both cases *locale* stands for the current setting of the **LC_MESSAGES** category of locale.

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

catclose( ) closes the message catalog identified by *catd*. It invalidates any following references to the message catalog defined by *catd*.

RETURN VALUES

catopen( ) returns a message catalog descriptor on success. On failure, it returns –1.

catclose( ) returns:

0	on success.
–1	on failure.

SEE ALSO

catgets(3C), **locale**(5)

NOTES

Using **catopen( )** and **catclose( )** in conjunction with **gettext( )** or **textdomain( )** (see **gettext**(3)) is undefined.

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**(2V) or **system**(3).

The resolution of the clock is 16.667 milliseconds.

SEE ALSO

wait(2V), **system**(3), **times**(3V)

BUGS

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

NAME

crypt, _crypt, setkey, encrypt – password and data encryption

SYNOPSIS

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

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

setkey(key)
char *key;

encrypt(block, edflag)
char *block;

DESCRIPTION

crypt() is the password encryption routine, based on the NBS Data Encryption Standard, with variations intended (among other things) to frustrate use of hardware implementations of the DES for key search.

The first argument to **crypt()** is normally a user's typed password. The second is a 2-character string chosen from the set [a-zA-Z0-9./]. Unless it starts with '##' or '#$', the *salt* string is used to perturb the DES algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password, in the same alphabet as the salt. The first two characters are the salt itself.

If the *salt* string starts with '##', **pwdauth**(3) is called. If *pwdauth* returns TRUE, the salt is returned from **crypt.** Otherwise, NULL is returned. If the *salt* string starts with '#$', **grpauth** (see **pwdauth**(3)) is called. If **grpauth** returns TRUE, the salt is returned from **crypt**. Otherwise, NULL is returned. If there is a valid reason not to have this authentication happen, calling *_crypt* avoids authentication.

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

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

SEE ALSO

login(1), **passwd**(1), **getpass**(3V), **pwdauth**(3), **passwd**(5)

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

ctermid – generate filename for terminal

SYNOPSIS

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

DESCRIPTION

ctermid( ) generates the pathname 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 **<stdio.h>** header file.

ctermid( ) returns a pointer to a null string if it fails, or if the pathname that would refer to the controlling terminal cannot be determined.

SEE ALSO

ttyname(3V)

NOTES

The difference between **ctermid**( ) and **ttyname**(3V) is that **ttyname**( ) must be passed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while **ctermid**( ) returns a string (**/dev/tty**) that will refer to the terminal if used as a file name. Thus **ttyname**( ) is useful only if the process already has at least one file open to a terminal. **ctermid**( ) is useful largely for making code portable to (non-UNIX) systems where the current terminal is referred to by a name other than **/dev/tty**.

NAME

ctime, asctime, dysize, gmtime, localtime, strftime, strptime, timegm, timelocal, tzset, tzsetwall – convert date and time

SYNOPSIS

```
#include <time.h>

char *ctime(clock)
time_t *clock;

char *asctime(tm)
struct tm *tm;

int dysize(y)
int y;

struct tm *gmtime(clock)
time_t *clock;

struct tm *localtime(clock)
time_t *clock;

int strftime(buf, bufsize, fmt, tm)
char *buf;
int bufsize;
char *fmt;
struct tm *tm;

char *strptime(buf, fmt, tm)
char *buf;
char *fmt;
struct tm *tm;

time_t timegm(tm)
struct tm *tm;

time_t timelocal(tm)
struct tm *tm;

void tzset( )

void tzsetwall( )
```

SYSTEM V SYNOPSIS

In addition to the routines above, the following variables are available:

```
extern long timezone;

extern int daylight;

extern char *tzname[2];
```

DESCRIPTION

ctime( ) converts a long integer, pointed to by *clock*, to a 26-character string of the form produced by **asctime( )**. It first breaks down *clock* to a **tm** structure by calling **localtime( )**, and then calls **asctime( )** to convert that **tm** structure to a string.

asctime( ) converts a time value contained in a **tm** structure to a 26-character string of the form:

Sun Sep 16 01:03:52 1973\n\0

Each field has a constant width. **asctime( )** returns a pointer to the string.

dysize( ) returns the number of days in the argument year, either 365 or 366. **localtime( )** and **gmtime( )** return pointers to structures containing the time, broken down into various components of that time represented in a particular time zone. **localtime( )** breaks down a time specified by the value pointed to by the *clock* argument, correcting for the time zone and any time zone adjustments (such as Daylight Savings Time). Before doing so, **localtime( )** calls **tzset( )** (if **tzset( )** has not been called in the current process). **gmtime( )** breaks down a time specified by the value pointed to by the *clock* argument into GMT, which is the time the system uses.

strftime( ) converts a time value contained in the **tm** structure pointed to by *tm* to a character string in a format specified by *fmt*. The character string is placed into the array pointed to by *buf*, which is assumed to contain room for at least *buflen* characters. If the result contains no more than *buflen* characters, **strftime( )** returns the number of characters produced (not including the terminating null character). Otherwise, it returns zero and the contents of the array are indeterminate. *fmt* is a character string that consists of field descriptors and text characters, reminiscent of **printf**(3V). Each field descriptor consists of a **%** character followd by another character that specifies the replacement for the field descriptor. All other characters are copied from *fmt* into the result. The following field descriptors are supported:

%%	same as %
%a	day of week, using locale's abbreviated weekday names
%A	day of week, using locale's full weekday names
%b **%h**	month, using locale's abbreviated month names
%B	month, using locale's full month names
%c	date and time as %x %X
%C	date and time, in locale's long-format date and time representation
%d	day of month (01-31)
%D	date as %m/%d/%y
%e	day of month (1-31; single digits are preceded by a blank)
%H	hour (00-23)
%I	hour (00-12)
%j	day number of year (001-366)
%k	hour (0-23; single digits are preceded by a blank)
%l	hour (1-12; single digits are preceded by a blank)
%m	month number (01-12)
%M	minute (00-59)
%n	same as \n
%p	locale's equivalent of AM or PM, whichever is appropriate
%r	time as %I:%M:%S %p
%R	time as %H:%M
%S	seconds (00-59)
%t	same as \t
%T	time as %H:%M:%S
%U	week number of year (01-52), Sunday is the first day of the week
%w	day of week; Sunday is day 0
%W	week number of year (01-52), Monday is the first day of the week
%x	date, using locale's date format
%X	time, using locale's time format

%y year within century (00-99)

%Y year, including century (fore example, 1988)

%Z time zone abbreviation

The difference between **%U** and **%W** lies in which day is counted as the first day of the week. Week number 01 is the first week with four or more January days in it.

strptime( ) converts the character string pointed to by *buf* to a time value, which is stored in the **tm** structure pointed to by *tm*, using the format specified by *fmt*. A pointer to the character following the last character in the string pointed to by *buf* is returned. *fmt* is a character string that consists of field descriptors and text characters, reminiscent of **scanf**(3v). Each field descriptor consists of a % character followd by another character that specifies the replacement for the field descriptor. All other characters are copied from *fmt* into the result. The following field descriptors are supported:

%% same as %

%a
%A day of week, using locale's weekday names; either the abbreviated or full name may be specified

%b
%B
%h month, using locale's month names; either the abbreviated or full name may be specified

%c date and time as %x %X

%C date and time, in locale's long-format date and time representation

%d
%e day of month (1-31; leading zeroes are permitted but not required)

%D date as %m/%d/%y

%H
%k hour (0-23; leading zeroes are permitted but not required)

%I
%l hour (0-12; leading zeroes are permitted but not required)

%j day number of year (001-366)

%m month number (1-12; leading zeroes are permitted but not required)

%M minute (0-59; leading zeroes are permitted but not required)

%p locale's equivalent of AM or PM

%r time as %I:%M:%S %p

%R time as %H:%M

%S seconds (0-59; leading zeroes are permitted but not required)

%T time as %H:%M:%S

%x date, using locale's date format

%X time, using locale's time format

%y year within century (0-99; leading zeroes are permitted but not required)

%Y year, including century (for example, 1988)

Case is ignored when matching items such as month or weekday names. The %M, %S, %y, and %Y fields are optional; if they would be matched by white space, the match is suppressed and the appropriate field of the **tm** structure pointed to by *tm* is left unchanged. If any of the format items **%d**, **%e**, **%H**, **%k**, **%I**, **%l**, **%m**, **%M**, **%S**, **%y**, or **%Y** are matched, but the string that matches them is followed by white space, all subsequent items in the format string are skipped up to white space or the end of the format. The net result is that, for example, the format **%m/%d/%y** can be matched by the string **12/31**; the **tm_mon** and **tm_mday** fields of the **tm** structure pointed to by *tm* will be set to 11 and 31, respectively, while the **tm_year** field will be unchanged.

timelocal( ) and **timegm( )** convert the time specified by the value pointed to by the *tm* argument to a time value that represents that time expressed as the number of seconds since Jan. 1, 1970, 00:00, Greenwich Mean Time. **timelocal( )** converts a **tm** structure that represents local time, correcting for the time zone and any time zone adjustments (such as Daylight Savings Time). Before doing so, **timelocal()** calls **tzset( )** (if **tzset( )** has not been called in the current process). **timegm( )** converts a **tm** structure that represents GMT.

tzset( ) uses the value of the environment variable TZ to set time conversion information used by **localtime( )**. If TZ is absent from the environment, the an 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; if TZ appears and begins with a slash, it is used as the absolute pathname of the *tzfile-format* (see **tzfile**(5)) file from which to read the time conversion information; if TZ appears and begins with a character other than a slash, it is 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.

Declarations of all the functions and externals, and the **tm** structure, are in the **<time.h>** header file. The structure (of type) **tm** structure includes the following fields:

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

tm_isdst is non-zero if Daylight Savings Time is in effect. **tm_zone** points to a string that is the name used for the local time zone at the time being converted. **tm_gmtoff** is the offset (in seconds) of the time represented from GMT, with positive values indicating East of Greenwich.

SYSTEM V DESCRIPTION

The external **long** variable **timezone** contains the difference, in seconds, between GMT and local standard time (in PST, **timezone** is 8*60*60). If this difference is not a constant, **timezone** will contain the value of the offset on January 1, 1970 at 00:00 GMT. Since this is not necessarily the same as the value at some particular time, the time in question should be converted to a **tm** structure using **localtime()** and the **tm_gmtoff** field of that structure should be used. The external variable **daylight** is non-zero if and only if Daylight Savings Time would be in effect within the current time zone at some time; it does not indicate whether Daylight Savings Time is currently in effect.

The external variable **tzname** is an array of two **char *** pointers. The first pointer points to a character string that is the name of the current time zone when Daylight Savings Time is not in effect; the second one, if Daylight Savings Time conversion should be applied, points to a character string that is the name of the current time zone when Daylight Savings Time is in effect. These strings are updated by **localtime( )** whenever a time is converted. If Daylight Savings Time is in effect at the time being converted, the second pointer is set to point to the name of the current time zone at that time, otherwise the first pointer is so set.

timezone, **daylight**, and **tzname** are retained for compatibility with existing programs.

FILES

/usr/share/lib/zoneinfo	standard time conversion information directory
/usr/share/lib/zoneinfo/localtime	local time zone file

SEE ALSO

gettimeofday(2), **getenv**(3V), **time**(3V), **environ**(5V), **tzfile**(5)

BUGS

The return values point to static data, whose contents are overwritten by each call. The **tm_zone** field of a returned **tm** structure 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, conv, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, iscntrl, isascii, isgraph, toupper, tolower, toascii – character classification and conversion macros and functions

SYNOPSIS

```
#include <ctype.h>

isalpha(c)

...
```

DESCRIPTION

Character Classification Macros

These macros classify character-coded integer values according to the rules of the coded character set defined by the character type information in the program's locale (category **LC_CTYPE**). On program startup the **LC_CTYPE** category of locale is equivalent to the **"C"** locale.

In the **"C"** locale, or in a locale where the character type information is not defined, characters are classified according to the rules of the US-ASCII 7-bit coded character set. The control characters are those below 040 (and the single byte 0177) (DEL). See **ascii**(7).

In all cases that argument is an **int**, the value of which must be representable as an **unsigned char** or must equal the value of the macro EOF. If the argument has any other value, the behavior is undefined.

Each is a predicate returning nonzero for true, zero for false. **isascii**( ) is defined on all integer values.

isalpha(*c*) *c* is a letter.

isupper(*c*) *c* is an upper case letter.

islower(*c*) *c* is a lower case letter.

isdigit(*c*) *c* is a digit [0-9].

isxdigit(*c*) *c* is a hexadecimal digit [0-9], [A-F], or [a-f].

isalnum(*c*) *c* is an alphanumeric character, that is, *c* is a letter or a digit.

isspace(*c*) *c* is a SPACE, TAB, RETURN, NEWLINE, FORMFEED, or vertical tab character.

ispunct(*c*) *c* is a punctuation character (neither control nor alphanumeric).

isprint(*c*) *c* is a printing character.

iscntrl(*c*) *c* is a delete character or ordinary control character.

isascii(*c*) *c* is an ASCII character, code less than 0200.

isgraph(*c*) *c* is a visible graphic character.

Character Conversion Macros

toascii(*c*)

Masks *c* with the correct value so that *c* is guaranteed to be an ASCII character in the range 0 through 0x7f. Will not perform mapping from a non-ASCII coded character set into ASCII.

Character Conversion Functions

These functions perform simple conversions on single characters. They replace the previous macro definitions which did not extend to support variant settings of the **LC_CTYPE** locale category.

toupper(*c*) Converts *c* to its upper-case equivalent. This function works correctly for all coded character sets and all characters within such sets selected by a valid setting of the **LC_CTYPE** locale category.

tolower(*c*) Converts *c* to its lower-case equivalent. This function works correctly for all coded character sets and all characters within such sets selected by a valid setting of the **LC_CTYPE** locale category.

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

SYSTEM V DESCRIPTION

Character Conversion Macros

The macros **_toupper**( ) and **_tolower**( ) are faster than the equivalent functions (**toupper**( ) and **tolower**( )) but only work properly on a restricted range of characters, and will not work on a **LC_CTYPE** category other than the default "C" (ASCII).

These macros perform simple conversions on single characters.

_toupper(*c*) converts *c* to its upper-case equivalent. Note: This *only* works where *c* is known to be a lower-case character to start with (presumably checked using **islower**( )).

_tolower(*c*) converts *c* to its lower-case equivalent. Note: This *only* works where *c* is known to be a upper-case character to start with (presumably checked using **isupper**( )).

SEE ALSO

setlocale(3V), **ascii**(7), **iso_8859_1**(7)

NAME

curses – System V 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

In SYSTEM V SYNOPSIS:

- compiling information

In DESCRIPTION and SYSTEM V DESCRIPTION:

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

In ROUTINES, descriptions of **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
- Low-level Curses Access
- Miscellaneous
- Use of **curscr**

In SYSTEM V ROUTINES, descriptions of **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:

- SYSTEM V ATTRIBUTES
- SYSTEM V FUNCTION KEYS

- LINE GRAPHICS

cc [*flags*] *files* **–lcurses –ltermcap** [*libraries*]

#include <curses.h> (automatically includes **<stdio.h>** and **<unctl.h>**.)

The parameters in the following list are not global variables. 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
```

SYSTEM V SYNOPSIS

/usr/5bin/cc [*flag* ...] *file* ... **–lcurses** [*library* ...]

#include <curses.h> (automatically includes **<stdio.h>**, **<termio.h>**, and **<unctrl.h>**).

DESCRIPTION

These routines give the user a method of updating screens with reasonable optimization. They keep an image of the current screen, and the user sets up an image of a new one. Then the **refresh( )** tells the routines to make the current screen look like the new one. In order to initialize the routines, the routine **initscr( )** must be called before any of the other routines that deal with windows and screens are used. The routine **endwin( )** should be called before exiting.

SYSTEM V 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 **tset** command in your **.profile** or **.login** file. For further details, see **tset**(1) and the **Tabs and Initialization** subsection of **terminfo**(5V).

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 that 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 that cause the characters to show up in modes such as underlined or in reverse video on terminals that support such display enhancements. Line drawing characters may be specified to be output. On input, **curses** is also able to translate arrow and function keys that transmit escape sequences into single values. The video attributes, line drawing characters, and input values use names, defined in **<curses.h>**, such as **A_REVERSE**, **ACS_HLINE**, and **KEY_LEFT**.

curses also defines the **WINDOW** * variable, **curscr**, which is used only for certain low-level operations like clearing and redrawing a garbaged screen. **curscr** can be used in only a few routines. If the window argument to **clearok**( ) is **curscr**, the next call to **wrefresh**( ) with any window will clear and repaint the screen 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 **curses**'s idea of how large a screen is.

If the environment variable **TERMINFO** is defined, any program using **curses** will check for a local terminal definition before checking in the standard place. For example, if the environment variable **TERM** is set to **sun**, then the compiled terminal definition is found in **/usr/share/lib/terminfo/s/sun**. The s is copied from the first letter of **sun** to avoid creation of huge directories.) However, if **TERMINFO** is set to **$HOME/myterms**, **curses** will first check **$HOME/myterms/s/sun**, and, if that fails, will then check **/usr/share/lib/terminfo/s/sun**. This is useful for developing experimental definitions or when write permission on **/usr/share/lib/terminfo** is not available.

The integer variables **LINES** and **COLS** are defined in **<curses.h>**, and will be filled in by **initscr**( ) with the size of the screen. For more information, see the subsection **Terminfo-Level Manipulations**. The constants **TRUE** and **FALSE** have the values **1** and **0**, respectively. The constants ERR and OK are returned by routines to indicate whether the routine successfully completed. These constants are also defined in **<curses.h>**.

ROUTINES

Many of the following routines have two or more versions. The routines prefixed with w require a *window* argument. The routines prefixed with **p** require a *pad* argument. Those without a prefix generally use **stdscr**.

The routines prefixed with **mv** require *y* and *x* coordinates to move to before performing the appropriate action. The **mv** routines imply a call to **move( )** before the call to the other routine. The window argument is always specified before the coordinates. *y* always refers to the row (of the window), and *x* always refers to the column. The upper left corner is always **(0,0)**, not **(1,1)**. The routines prefixed with **mvw** take both a *window* argument and *y* and *x* coordinates.

In each case, *win* is the window affected and *pad* is the pad affected. (**win** and *pad* are always of type **WINDOW *.**) Option-setting routines require a boolean flag *bf* with the value **TRUE** or **FALSE.** (*bf* is always of type **bool.**) The types **WINDOW, bool,** and **chtype** are defined in **<curses.h>** (see **SYNOPSIS** for a summary of what types all variables are).

All routines return either the integer ERR or the integer OK, unless otherwise noted. Routines that return pointers always return NULL on error.

Overall Screen Manipulation

WINDOW *initscr( )
The first routine called should almost always be **initscr( )**. The exceptions are **slk_init( ), filter( ),** and **ripoffline( )**. This will determine the terminal type and initialize all **curses** data structures. **initscr( )** also arranges that the first call to **refresh( )** will clear the screen. If errors occur, **initscr( )** will write an appropriate error message to standard error and exit; otherwise, a pointer to **stdscr** is returned. If the program wants an indication of error conditions, **newterm( )** should be used instead of **initscr( )**. **initscr( )** should only be called once per application.

endwin( )
A program should always call **endwin( )** before exiting or escaping frcm **curses** mode temporarily, to do a shell escape or **system**(3) call, for example. This routine will restore **termio**(4) 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( )**.

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: **refresh( )** is a macro.

WINDOW *newwin (*nlines, ncols, begin_y, begin_x*)
Create and return a pointer to a new window with the given number of lines (or rows), *nlines*, and columns, *ncols*. The upper left corner of the window is at line *begin_y*, column *begin_x*. If either *nlines* or *ncols* is 0, they will be set to the value of **lines**–*begin_y* and **cols**–*begin_x*. A new full-screen window is created by calling **newwin(0,0,0,0)**.

mvwin (*win, y, x*)
Move the window so that the upper left corner will be at position (*y*, *x*). If the move would cause the window to be off the screen, it is an error and the window is not moved.

WINDOW *subwin (*orig, nlines, ncols, begin_y, begin_x*)
Create and return a pointer to a new window with the given number of lines (or rows), *nlines*, and columns, *ncols*. The window is at position (*begin_y, begin_x*) on the screen. This position is relative to the screen, and not to the window *orig*. The window is made in the middle of the window *orig*, so that changes made to

one window will affect both windows. When using this routine, often it will be necessary to call **touchwin( )** or **touchline( )** on *orig* before calling **wrefresh**.

delwin (*win*)
Delete the named window, freeing up all memory associated with it. In the case of overlapping windows, subwindows should be deleted before the main window.

Output

These routines are used to "draw" text on windows.

addch (*ch*)

waddch (*win, ch*)

mvaddch (*y, x, ch*)

mvwaddch (*win, y, x, ch*)

The character *ch* is put into the window at the current cursor position of the window and the position of the window cursor is advanced. Its function is similar to that of **putchar( )** (see **putc**(3s)). At the right margin, an automatic newline is performed. At the bottom of the scrolling region, if **scrollok( )** is enabled, the scrolling region will be scrolled up one line.

If *ch* is a TAB, NEWLINE, or backspace, the cursor will be moved appropriately within the window. A NEWLINE also does a **clrtoeol( )** before moving. TAB characters are considered to be at every eighth column. If *ch* is another control character, it will be drawn in the CTRL-X notation. (Calling **winch( )** after adding a control character will not return the control character, but instead will return the representation of the control character.)

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

Note: *ch* is actually of type **chtype**, not a character.

Note: **addch( )**, **mvaddch( )**, and **mvwaddch( )** are macros.

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: **addstr( )**, **mvaddstr( )**, and **mvwaddstr( )** are macros.

box (*win, vertch, horch*)

A box is drawn around the edge of the window, *win*. *vertch* and *horch* are the characters the box is to be drawn with. If *vertch* and *horch* are **0**, then appropriate default characters, **ACS_VLINE** and **ACS_HLINE**, will be used.

Note: *vertch* and *horch* are actually of type **chtype**, not characters.

erase( )

werase (*win*)
These routines copy blanks to every position in the window.

Note: **erase( )** is a macro.

clear( )

wclear (*win*) These routines are like **erase( )** and **werase( )**, but they also call **clearok( )**, arranging that the screen will be cleared completely on the next call to **wrefresh( )** for that window, and repainted from scratch.

Note: **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: **clrtobot( )** is a macro.

clrtoeol( )

wclrtoeol (*win*) The current line to the right of the cursor, inclusive, is erased.

Note: **clrtoeol( )** is a macro.

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: **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: **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.

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: *ch* is actually of type **chtype**, not a character.

Note: **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: **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: **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.

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

wprintw (*win*, *fmt* [, arg . . .])

mvprintw (*y*, *x*, *fmt* [, *arg* . . .])

mvwprintw (*win*, *y*, *x*, *fmt* [, *arg* . . .])

These routines are analogous to **printf**(3V). The string that would be output by **printf**(3V) is instead output using **waddstr**( ) on the given window.

scroll (*win*) The window is scrolled up one line. This involves moving the lines in the window data structure. As an optimization, if the window is **stdscr** and the scrolling region is the entire window, the physical screen will be scrolled at the same time.

touchwin (*win*)

touchline (*win*, *start*, *count*)

Throw away all optimization information about which parts of the window have been touched, by pretending that the entire window has been drawn on. This is sometimes necessary when using overlapping windows, since a change to one window will affect the other window, but the records of which lines have been changed in the other window will not reflect the change. **touchline**( ) only pretends that *count* lines have been changed, beginning with line *start*.

Input

getch( )

wgetch (*win*)

mvgetch (*y*, *x*)

mvwgetch (*win*, *y*, *x*) A character is read from the terminal associated with the window. In NODELAY mode, if there is no input waiting, the value ERR is returned. In DELAY mode, the program will hang until the system passes text through to the program. Depending on the setting of **cbreak**( ), this will be after one character (CBREAK mode), or after the first newline (NOCBREAK mode). In HALF-DELAY mode, the program will hang until a character is typed or the specified timeout has been reached. Unless **noecho**( ) has been set, the character will also be echoed into the designated window. No **refresh**( ) will occur between the **move**( ) and the **getch**( ) done within the routines **mvgetch**( ) and **mvwgetch**( ).

When using **getch**( ), **wgetch**( ), **mvgetch**( ), or **mvwgetch**( ), do not set both NOCBREAK mode (**nocbreak**( )) and ECHO mode (**echo**( )) at the same time. Depending on the state of the terminal driver when each character is typed, the program may produce undesirable results.

If **keypad** (*win*, **TRUE**) has been called, and a function key is pressed, the token for that function key will be returned instead of the raw characters. See **keypad**( ) under **Input Options Setting**. Possible function keys are defined in **<curses.h>** with integers beginning with **0401**, whose names begin with **KEY_**. If a character is received that could be the beginning of a function key (such as escape), **curses** will set a timer. If the remainder of the sequence is not received within the designated time, the character will be passed through, otherwise the function key value will be returned. For this reason, on many terminals, there will be a delay after a user presses the escape key before the escape is returned to the program. Use by a programmer of the escape key for a single character routine is discouraged. Also see **notimeout**( ) below.

Note: **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: **getstr**( ), **mvgetstr**( ), and **mvwgetstr**( ) are macros.

inch( )

winch (*win*)

mvinch (*y*, *x*)

mvwinch (*win*, *y*, *x*)

The character, of type **chtype**, at the current position in the named window is returned. If any attributes are set for that position, their values will be OR'ed into the value returned. The predefined constants **A_CHARTEXT** and **A_ATTRIBUTES**, defined in **<curses.h>**, can be used with the C logical AND (&) operator to extract the character or attributes alone.

Note: **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**(3V), as do their arguments and return values. **wgetstr**( ) is called on the window, and the resulting line is used as input for the scan.

Output Options Setting

These routines set options within **curses** that deal with output. All options are initially FALSE, unless otherwise stated. It is not necessary to turn these options off before calling **endwin**( ).

clearok (*win*, *bf*)

If enabled (*bf* is **TRUE**), the next call to **wrefresh**( ) with this window will clear the screen completely and redraw the entire screen from scratch. This is useful when the contents of the screen are uncertain, or in some cases for a more pleasing visual effect.

idlok (*win*, *bf*)

If enabled (*bf* is **TRUE**), **curses** will consider using the hardware "insert/delete-line" feature of terminals so equipped. If disabled (*bf* is **FALSE**), **curses** will very seldom use this feature. The "insert/delete-character" feature is always considered. This option should be enabled only if your application needs "insert/delete-line", for example, for a screen editor. It is disabled by default because "insert/delete-line" tends to be visually annoying when used in applications where it is not really needed. If "insert/delete-line" cannot be used, **curses** will redraw the changed portions of all lines.

leaveok (*win*, *bf*)

Normally, the hardware cursor is left at the location of the window cursor being refreshed. This option allows the cursor to be left wherever the update happens to leave it. It is useful for applications where the cursor is not used, since it reduces the need for cursor motions. If possible, the cursor is made invisible when this option is enabled.

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: 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 RETURN and LINEFEED on output, and whether RETURN is translated into NEWLINE on input. Initially, the translations do occur. By disabling these translations using **nonl**( ), **curses** is able to make better use of the linefeed capability, resulting in faster cursor motion.

Input Options Setting

These routines set options within **curses** that deal with input. The options involve using **ioctl**(2) and therefore interact with **curses** routines. It is not necessary to turn these options off before calling **endwin**( ).

For more information on these options, refer to *Programming Utilities and Libraries*.

cbreak( )

nocbreak( )

These two routines put the terminal into and out of CBREAK mode, respectively. In CBREAK mode, characters typed by the user are immediately available to the program and erase/kill character processing is not performed. When in NOCBREAK mode, the tty driver will buffer characters typed until a NEWLINE or RETURN is typed. Interrupt and flow-control characters are unaffected by this mode (see **termio**(4)). Initially the terminal may or may not be in CBREAK mode, as it is inherited, therefore, a program should call **cbreak**( ) or **nocbreak**( ) explicitly. Most interactive programs using **curses** will set CBREAK mode.

Note: **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**( ).

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 terminal driver that are not set by **curses.**

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.

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.

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.

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( )**.

Miscellaneous

unctrl (*c*) This macro expands to a character string which is a printable representation of the character *c*. Control characters are displayed in the ^X notation. Printing characters are displayed as is.

unctrl( ) is a macro, defined in **<unctrl.h>**, which is automatically included by **<curses.h>**.

flusok(*win,boolf*) set flush-on-refresh flag for *win*

getcap(*name*) get terminal capability *name*

touchoverlap(*win1,win2*)
mark overlap of *win1* on *win2* as changed

Use of curscr

The special window **curscr** can be used in only a few routines. If the window argument to **clearok( )** is **curscr**, the next call to **wrefresh( )** with any window will cause the screen to be cleared and repainted from scratch. If the window argument to **wrefresh( )** is **curscr**, the screen is immediately cleared and repainted from scratch. This is how most programs would implement a "repaint-screen" routine. The source window argument to **overlay( )**, **overwrite( )**, and **copywin** may be **curscr**, in which case the current contents of the virtual terminal screen will be accessed.

Obsolete Calls

Various routines are provided to maintain compatibility in programs written for older versions of the curses library. These routines are all emulated as indicated below.

crmode( )	Replaced by **cbreak( )**.
gettmode( )	A no-op.
nocrmode( )	Replaced by **nocbreak( )**.

SYSTEM V ROUTINES

The above routines are available as described except for **flusok( )**, **getcap( )** and **touchoverlap( )** which are not available.

In addition, the following routines are available:

Overall Screen Manipulation

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**(3V) 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 that manipulates **SCREEN** pointers; all other routines affect only the current terminal.

Window and Pad Manipulation

wnoutrefresh (*win*)

doupdate( ) These two routines allow multiple updates to the physical terminal screen with more efficiency than **wrefresh( )** alone. How this is accomplished is described in the next paragraph.

curses keeps two data structures representing the terminal screen: a *physical* terminal screen, describing what is actually on the screen, and a *virtual* terminal screen, describing what the programmer wants to have on the screen. **wrefresh( )** works by first calling **wnoutrefresh( )**, which copies the named window to the virtual screen, and then by calling **doupdate( )**, which compares the virtual screen to the physical screen and does the actual update. If the programmer wishes to output several windows at once, a series of calls to **wrefresh( )** will result in alternating calls to **wnoutrefresh( )** and **doupdate( )**, causing several bursts of output to the screen. By first calling **wnoutrefresh( )** for each window, it is then possible to call **doupdate( )** once, resulting in only one burst of output, with probably fewer total characters transmitted and certainly less processor time used.

WINDOW *newpad (*nlines, ncols*)

Create and return a pointer to a new pad data structure with the given number of lines (or rows), *nlines*, and columns, *ncols*. A pad is a window that is not restricted by the screen size and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (for example, from scrolling or echoing of input) do not occur. It is not legal to call **wrefresh**( ) with a pad as an argument; the routines **prefresh**( ) or **pnoutrefresh**( ) should be called instead. Note: these routines require additional parameters to specify the part of the pad to be displayed and the location on the screen to be used for display.

WINDOW *subpad (*orig, nlines, ncols, begin_y, begin_x*)

Create and return a pointer to a subwindow within a pad with the given number of lines (or rows), *nlines*, and columns, *ncols*. Unlike **subwin**(), which uses screen coordinates, the window is at position (*begin_y, begin_x*) on the pad. The window is made in the middle of the window *orig*, so that changes made to one window will affect both windows. When using this routine, often it will be necessary to call **touchwin**( ) or **touchline**( ) on *orig* before calling **prefresh**( ).

prefresh (*pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol*)

pnoutrefresh (*pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol*)

These routines are analogous to

wrefresh( ) and **wnoutrefresh**( ) except that pads, instead of windows, are involved. The additional parameters are needed to indicate what part of the pad and screen are involved. *pminrow* and *pmincol* specify the upper left corner, in the pad, of the rectangle to be displayed. *sminrow*, *smincol*, *smaxrow*, and *smaxcol* specify the edges, on the screen, of the rectangle to be displayed in. The lower right corner in the pad of the rectangle to be displayed is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of *pminrow*, *pmincol*, *sminrow*, or *smincol* are treated as if they were zero.

Output

These routines are used to "draw" text on windows.

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: *ch* is actually of type **chtype**, not a character.

Note: **echochar**( ) is a macro.

attroff (*attrs*)

wattroff (*win, attrs*)

attron (*attrs*)

wattron (*win, attrs*)

attrset (*attrs*)

wattrset (*win, attrs*)

beep( )

flash( )
These routines are used to signal the terminal user. **beep**( ) will sound the audible alarm on the terminal, if possible, and if not, will flash the screen (visible bell), if that is possible. **flash**( ) will flash the screen, and if that is not possible, will sound the audible signal. If neither signal is possible, nothing will happen. Nearly all terminals have an audible signal (bell or beep) but only some can flash the screen.

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.

getbegyx (*win, y, x*)

getmaxyx (*win, y, x*)
Like **getyx**( ), these routines store the current beginning coordinates and size of the specified window.

Note: **getbegyx**( ) and **getmaxyx**( ) are macros.

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**( ).

vwprintw (*win, fmt, varglist*)
This routine corresponds to **vprintf**(3V). 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**(3V) and **varargs**(3) manual pages for a detailed description on how to use variable argument lists.

Input

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**( ).

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**(3V) and **varargs**(3) 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**( ).

setscrreg (*top, bot*)

wsetscrreg (*win, top, bot*)
These routines allow the user to set a software scrolling region in a window. *top* and *bot* are the line numbers of the top and bottom margin of the scrolling region. Line 0 is the top line of the window. If this option and **scrollok**( ) are enabled, an

attempt to move off the bottom margin line will cause all lines in the scrolling region to scroll up one line. Note: this has nothing to do with use of a physical scrolling region capability in the terminal, like that in the DEC VT100. Only the text of the window is scrolled; if **idlok()** is enabled and the terminal has either a scrolling region or "insert/delete-line" capability, they will probably be used by the output routines.

Note: **setscrreg()** and **wsetscrreg()** are macros.

Input Options Setting

These routines set options within **curses** that deal with input. The options involve using **ioctl(2)** and therefore interact with **curses** routines. It is not necessary to turn these options off before calling **endwin()**.

For more information on these options, refer to *Programming Utilities and Libraries.*

halfdelay (*tenths*)
Half-delay mode is similar to CBREAK mode in that characters typed by the user are immediately available to the program. However, after blocking for *tenths* tenths of seconds, ERR will be returned if nothing has been typed. *tenths* must be a number between 1 and 255. Use **nocbreak()** to leave half-delay mode.

intrflush (*win, bf*)
If this option is enabled, when an interrupt key is pressed on the keyboard (interrupt, break, quit) all output in the tty driver queue will be flushed, giving the effect of faster response to the interrupt, but causing **curses** to have the wrong idea of what is on the screen. Disabling the option prevents the flush. The default for the option is inherited from the tty driver settings. The window argument is ignored.

keypad (*win, bf*)
This option enables the keypad of the user's terminal. If enabled, the user can press a function key (such as an arrow key) and **wgetch()** will return a single value representing the function key, as in **KEY_LEFT**. If disabled, **curses** will not treat function keys specially and the program would have to interpret the escape sequences itself. If the keypad in the terminal can be turned on (made to transmit) and off (made to work locally), turning on this option will cause the terminal keypad to be turned on when **wgetch()** is called.

meta (*win, bf*)
If enabled, characters returned by **wgetch()** are transmitted with all 8 bits, instead of with the highest bit stripped. In order for **meta()** to work correctly, the **km** (**has_meta_key**) capability has to be specified in the terminal's **terminfo**(5V) entry.

nodelay (*win, bf*)
This option causes **wgetch()** to be a non-blocking call. If no input is ready, **wgetch()** will return ERR. If disabled, **wgetch()** will hang until a key is pressed.

notimeout (*win, bf*)
While interpreting an input escape sequence, **wgetch()** will set a timer while waiting for the next character. If **notimeout** (*win,* **TRUE**) is called, then **wgetch()** will not set a timer. The purpose of the timeout is to differentiate between sequences received from a function key and those typed by a user.

typeahead (*fildes*)
curses does "line-breakout optimization" by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update will be postponed until **refresh()** or **doupdate()** is called again. This allows faster response to commands typed in advance. Normally, the file descriptor for the input FILE pointer passed to **newterm()**, or **stdin** in the case that **initscr()** was used, will be used to do this typeahead checking. The **typeahead()** routine specifies that the file descriptor *fildes* is to be used to check for typeahead instead. If *fildes* is **–1**, then no typeahead checking will be done.

Note: *fildes* is a file descriptor, not a **<stdio.h>** FILE pointer.

Environment Queries

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()**.

Soft Labels

If desired, **curses** will manipulate the set of soft function-key labels that exist on many terminals. For those terminals that do not have soft labels, if you want to simulate them, **curses** will take over the bottom line of **stdscr**, reducing the size of **stdscr** and the variable **LINES**. **curses** standardizes on 8 labels of 8 characters each.

slk_init (*labfmt***)** In order to use soft labels, this routine must be called before **initscr()** or **newterm()** is called. If **initscr()** winds up using a line from **stdscr** to emulate the soft labels, then *labfmt* determines how the labels are arranged on the screen. Setting *labfmt* to **0** indicates that the labels are to be arranged in a 3-2-3 arrangement; **1** asks for a 4-4 arrangement.

slk_set (*labnum, label, labfmt***)**
labnum is the label number, from 1 to 8. *label* is the string to be put on the label, up to 8 characters in length. A null string or a NULL pointer will put up a blank label. *labfmt* is one of **0**, **1** or **2**, to indicate whether the label is to be left-justified, centered, or right-justified within the label.

slk_refresh()

slk_noutrefresh() These routines correspond to the routines **wrefresh()** and **wnoutrefresh()**. Most applications would use **slk_noutrefresh()** because a **wrefresh()** will most likely soon follow.

char *slk_label (*labnum***)**
The current label for label number *labnum*, with leading and trailing blanks stripped, is returned.

slk_clear() The soft labels are cleared from the screen.

slk_restore() The soft labels are restored to the screen after a **slk_clear()**.

slk_touch() All of the soft labels are forced to be output the next time a **slk_noutrefresh()** is performed.

Low-Level curses Access

The following routines give low-level access to various **curses** functionality. These routines typically would be used inside of library routines.

def_prog_mode()

def_shell_mode() Save the current terminal modes as the "program" (in **curses**) or "shell" (not in **curses**) state for use by the **reset_prog_mode()** and **reset_shell_mode()** routines. This is done automatically by **initscr()**.

reset_prog_mode()

reset_shell_mode() Restore the terminal to "program" (in **curses**) or "shell" (out of **curses**) state. These are done automatically by **endwin()** and **doupdate()** after an **endwin()**, so they normally would not be called.

getsyx (*y*, *x*)
The current coordinates of the virtual screen cursor are returned in *y* and *x*. Like **getyx**( ), the variables *y* and *x* do not take an & before them. If **leaveok**( ) is currently **TRUE**, then **–1**, **–1** will be returned. If lines may have been removed from the top of the screen using **ripoffline**( ) and the values are to be used beyond just passing them on to **setsyx**( ), the value *y*+**stdscr–>_yoffset** should be used for those other uses.

Note: **getsyx**( ) is a macro.

setsyx (*y*, *x*)
The virtual screen cursor is set to *y*, *x*. If *y* and *x* are both –1, then **leaveok**( ) will be set. The two routines **getsyx**( ) and **setsyx**( ) are designed to be used by a library routine that manipulates **curses** windows but does not want to mess up the current position of the program's cursor. The library routine would call **getsyx**( ) at the beginning, do its manipulation of its own windows, do a **wnoutrefresh**( ) on its windows, call **setsyx**( ), and then call **doupdate**( ).

ripoffline (*line*, *init*)
This routine provides access to the same facility that **slk_init**( ) uses to reduce the size of the screen. **ripoffline**( ) must be called before **initscr**( ) or **newterm**( ) is called. If *line* is positive, a line will be removed from the top of **stdscr**; if negative, a line will be removed from the bottom. When this is done inside **initscr**( ), the routine *init* is called with two arguments: a window pointer to the 1-line window that has been allocated and an integer with the number of columns in the window. Inside this initialization routine, the integer variables **LINES** and **COLS** (defined in **<curses.h>**) are not guaranteed to be accurate and **wrefresh**( ) or **doupdate**( ) must not be called. It is allowable to call **wnoutrefresh**( ) during the initialization routine.

ripoffline( ) can be called up to five times before calling **initscr**( ) or **newterm**( ).

scr_dump (*filename*)
The current contents of the virtual screen are written to the file *filename*.

scr_restore (*filename*)
The virtual screen is set to the contents of *filename*, which must have been written using **scr_dump**( ). The next call to **doupdate**( ) will restore the screen to what it looked like in the dump file.

scr_init (*filename*)
The contents of *filename* are read in and used to initialize the **curses** data structures about what the terminal currently has on its screen. If the data is determined to be valid, **curses** will base its next update of the screen on this information rather than clearing the screen and starting from scratch. **scr_init**( ) would be used after **initscr**( ) or a **system**(3) call to share the screen with another process that has done a **scr_dump**( ) after its **endwin**( ) call. The data will be declared invalid if the time-stamp of the tty is old or the **terminfo**(5V) 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 that 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**(5V) 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: **setupterm**( ) is automatically called by **initscr**( ) and **newterm**( ). This will define the set of terminal-dependent variables defined in the **terminfo**(5V) database. The **terminfo**(5V) variables *lines* and *columns* (see **terminfo**(5V)) are initialized by **setupterm**( ) as follows: if the environment variables **LINES** and **COLUMNS** exist, their values are used. If the above environment variables do not exist, and the window sizes in rows and columns as returned by the **TIOCGWINSZ ioctl** are non-zero, those sizes are used. Otherwise, the values for *lines* and *columns* specified in the **terminfo**(5V) 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**(5V) strings (including the output of **tparm**( ) should be printed with **tputs**() or **putp**(). Before exiting, **reset_shell_mode**( ) should be called to restore the tty modes. Programs that use cursor addressing should output **enter_ca_mode** upon startup and should output **exit_ca_mode** before exiting (see **terminfo**(5V)). 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: this is different from the **curses** routines (see **endwin**( )).

setupterm (*term, fildes, errret*)

Reads in the **terminfo**(5V) database, initializing the **terminfo**(5V) structures, but does not set up the output virtualization structures used by **curses**. The terminal type is in the character string *term*; if *term* is NULL, the environment variable TERM will be used. All output is to the file descriptor *fildes*. If *errret* is not NULL, then **setupterm**( ) will return OK or ERR and store a status value in the integer pointed to by *errret*. A status of **1** in *errret* is normal, **0** means that the terminal could not be found, and **–1** means that the **terminfo**(5V) database could not be found. If *errret* is NULL, **setupterm**( ) will print an error message upon finding an error and exit. Thus, the simplest call is '**setupterm ((char *)0, 1, (int *)0)**', which uses all the defaults.

The **terminfo**(5V) 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**(5V) boolean, numeric and string variables refer to. The pointer may be saved before calling **setupterm**( ) again. Further calls to **setupterm**( ) will allocate new space rather than reuse the space pointed to by *cur_term*.

set_curterm (*nterm*) *nterm* is of type TERMINAL * . **set_curterm**( ) sets the variable *cur_term* to *nterm*, and makes all of the **terminfo**(5V) 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**(5V) 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*, $p_1, p_2, \ldots, p_9$)

Instantiate the string *str* with parms $\mathrm{p_i}$. 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**(5V) string variable or the return value from **tparm**( ), **tgetstr**( ), **tigetstr**( ) or **tgoto**( ). *count* is the number of lines affected, or **1** if not applicable. **putchar**( ) is a **putc**(3s)-like routine to which the characters are passed, one at a time.

putp (*str*) A routine that calls **tputs**( ) (*str*, **1**, **putc**(3s).

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 **putc**(3s)-like routine **putc**(3s).

vidattr (*attrs*) Like **vidputs**( ), except that it outputs through **putc**(3s).

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.

Termcap Emulation

These routines are included as a conversion aid for programs that use the **termcap**(3X) library. Their parameters are the same and the routines are emulated using the **terminfo**(5V) 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

char *keyname (*c*) A character string corresponding to the key *c* is returned.

filter( ) This routine is one of the few that is to be called before **initscr**( ) or **newterm**( ) is called. It arranges things so that **curses** thinks that there is a 1-line screen. **curses** will not use any terminal capabilities that assume that they know what line on the screen the cursor is on.

Use of curscr

The special window **curscr** can be used in only a few routines. If the window argument to **clearok**( ) is **curscr**, the next call to **wrefresh**( ) with any window will cause the screen to be cleared and repainted from scratch. If the window argument to **wrefresh**( ) is **curscr**, the screen is immediately cleared and repainted from scratch. This is how most programs would implement a "repaint-screen" routine. The source window argument to **overlay**( ), **overwrite**( ), and **copywin** may be **curscr**, in which case the current contents of the virtual terminal screen will be accessed.

Obsolete Calls

Various routines are provided to maintain compatibility in programs written for older versions of the curses library. These routines are all emulated as indicated below.

crmode( ) Replaced by **cbreak**( ).

fixterm( ) Replaced by **reset_prog_mode**( ).

nocrmode( ) Replaced by **nocbreak**( ).

resetterm( )	Replaced by **reset_shell_mode( )**.
saveterm( )	Replaced by **def_prog_mode( )**.
setterm( )	Replaced by **setupterm( )**.

SYSTEM V 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)'

SYSTEM V FUNCTION KEYS

The following function keys, defined in **<curses.h>**, might be returned by **getch( )** if **keypad( )** has been enabled. Note: 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**(5V) 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 TAB characters
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 **waddce**. When defined for the terminal, the variable will have the **A_ALTCHARSET** bit turned on. Otherwise, the default character listed below will be stored in the variable. The names were chosen to be consistent with the DEC VT100 nomenclature.

Name	*Default*	*Glyph Description*
ACS_ULCORNER	+	upper left corner
ACS_LLCORNER	+	lower left corner
ACS_URCORNER	+	upper right corner
ACS_LRCORNER	+	lower right corner
ACS_RTEE	+	right tee (┤)
ACS_LTEE	+	left tee (├)
ACS_BTEE	+	bottom tee (⊥)
ACS_TTEE	+	top tee (⊤)
ACS_HLINE	−	horizontal line
ACS_VLINE	|	vertical line
ACS_PLUS	+	plus
ACS_S1	−	scan line 1
ACS_S9	_	scan line 9
ACS_DIAMOND	+	diamond
ACS_CKBOARD	:	checker board (stipple)
ACS_DEGREE	'	degree symbol
ACS_PLMINUS	#	plus/minus
ACS_BULLET	o	bullet
ACS_LARROW	<	arrow pointing left
ACS_RARROW	>	arrow pointing right
ACS_DARROW	v	arrow pointing down
ACS_UARROW	^	arrow pointing up
ACS_BOARD	#	board of squares
ACS_LANTERN	#	lantern symbol
ACS_BLOCK	#	solid square block

RETURN VALUES

Unless otherwise noted in the preceding routine descriptions, all routines return:

OK	on success.
ERR	on failure.

SYSTEM V RETURN VALUES

All macros return the value of their w version, except **setscrreg**( ), **wsetscrreg**( ), **getsyx**( ), **getyx**( ), **getbegy**( ), **getmaxyx**( ), which return no useful value.

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

FILES

.login

.profile

SYSTEM V FILES

/usr/share/lib/terminfo

SEE ALSO

cc(1V), **ld**(1), **ioctl**(2), **getenv**(3V), **plot**(3X), **printf**(3V), **putc**(3S), **scanf**(3V), **stdio**(3V), **system**(3), **varargs**(3), **vprintf**(3V), **termio**(4), **tty**(4), **term**(5V), **termcap**(5), **terminfo**(5V), **tic**(8V)

SYSTEM V WARNINGS

The plotting library **plot**(3X) and the curses library **curses**(3V) 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**(3V) code, and/or '**#undef move**' and '**#undef 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**(3V)) for output during that time can cause unpredictable results.

NAME

cuserid – get character login name of the user

SYNOPSIS

#include <stdio.h>

char *cuserid(s)
char *s;

DESCRIPTION

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

SEE ALSO

cc(1V), **ld**(1), **getlogin**(3V), **getpwent**(3V)

RETURN VALUES

cuserid() returns a pointer to the login name on success. On failure, **cuserid()** returns NULL, and if *s* is not NULL, places a null character (**'\0'**) at s[0].

NOTES

The internal static area to which **cuserid()** writes when *s* is NULL will be overwritten by a subsequent call to **getpwnam()** (see **getpwent**(3V)).

A compatibility problem has been identified with the **cuserid()** function. The traditional version of this library routine in SunOS Release 3.2 and later releases and all System V releases calls the **getlogin()** function, and if it fails uses the **getpwuid()** function to try to return a name associated with the real user ID associated with the calling process. POSIX.1 requires that the **cuserid()** function try to return a name associated with the effective user ID associated with the calling process. Although this usually yields the same results, use of set-uid programs may yield different results.

A binding interpretation has been issued by IEEE saying that the POSIX.1 functionality has to be provided for compliance with POSIX.1. However, balloting on the first update to POSIX.1, P1003.1a, has led to the removal of the **cuserid()** function from the standard. (This is the state in the second recirculation ballot of P1003.1a dated 11 December 1989.) The objections leading to this resolution had both users and implementors arguing for the historical version and for the version specified by POSIX.1. The only way to reach consensus appears to be to remove the function from the standard.

To further complicate the issue, System V Release 4.0 has kept the traditional version of **cuserid()**. XPG3 specifies the POSIX.1 version of **cuserid()**, but the test suite for conformance to XPG3 promises to accept either implementation. Both of these are anticipating the final approval of P1003.1a as a standard with the **cuserid()** function removed. Since we also expect the **cuserid()** function to be dropped from the standard when P1003.1a is approved, SunOS Release 4.1 provides the traditional **cuserid()** function in the C library. However, for users that need the version specified by POSIX.1, it is provided in a POSIX library available in the System V environment. This library can be accessed by specifying **–lposix** on the **cc**(1V) or **ld**(1) command line.

NAME

dbm, dbminit, dbmclose, fetch, store, delete, firstkey, nextkey – data base subroutines

SYNOPSIS

```
#include <dbm.h>

typedef struct {
        char *dptr;
        int dsize;
} datum;

dbminit(file)
char *file;

dbmclose()

datum fetch(key)
datum key;

store(key, content)
datum key, content;

delete(key)
datum key;

datum firstkey()

datum nextkey(key)
datum key;
```

DESCRIPTION

Note: the **dbm()** library has been superceded by **ndbm**(3), and is now implemented using **ndbm()**.

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

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

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

A database may be closed by calling **dbmclose**. You must close a database before opening a new one.

Once open, the data stored under a key is accessed by **fetch()** and data is placed under a key by **store**. A key (and its associated contents) is deleted by **delete**. A linear pass through all keys in a database may be made, in an (apparently) random order, by use of **firstkey()** and **nextkey**. **firstkey()** will return the first key in the database. With any key **nextkey()** will return the next key in the database. This code will traverse the data base:

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

SEE ALSO

ar(1V), **cat**(1V), **cp**(1), **tar**(1), **ndbm**(3)

DIAGNOSTICS

All functions that return an **int** indicate errors with negative values. A zero return indicates no error. Routines that return a **datum** indicate errors with a NULL (0) *dptr*.

BUGS

The **.pag** file will contain holes so that its apparent size is about four times its actual content. Older versions of the UNIX operating system may create real file blocks for these holes when touched. These files cannot be copied by normal means (**cp**(1), **cat**(1V), **tar**(1), **ar**(1V)) without filling in the holes.

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

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

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

The order of keys presented by **firstkey**( ) and **nextkey**( ) depends on a hashing function, not on anything interesting.

There are no interlocks and no reliable cache flushing; thus concurrent updating and reading is risky.

NAME

decimal_to_single, decimal_to_double, decimal_to_extended – convert decimal record to floating-point value

SYNOPSIS

```
#include <floatingpoint.h>

void decimal_to_single(px, pm, pd, ps)
single *px ;
decimal_mode *pm;
decimal_record *pd;
fp_exception_field_type *ps;

void decimal_to_double(px, pm, pd, ps)
double *px ;
decimal_mode *pm;
decimal_record *pd;
fp_exception_field_type *ps;

void decimal_to_extended(px, pm, pd, ps)
extended *px ;
decimal_mode *pm;
decimal_record *pd;
fp_exception_field_type *ps;
```

DESCRIPTION

The **decimal_to_floating**( ) functions convert the decimal record at **pd* into a floating-point value at **px*, observing the modes specified in **pm* and setting exceptions in **ps*. If there are no IEEE exceptions, **ps* will be zero.

pd->sign and *pd->fpclass* are always taken into account. *pd->exponent* and *pd->ds* are used when *pd->fpclass* is *fp_normal* or *fp_subnormal*. In these cases *pd->ds* must contain one or more ascii digits followed by a null character. **px* is set to a correctly rounded approximation to

(pd->sign)*(pd->ds)*10(pd->exponent)**

Thus if *pd->exponent* == –2 and *pd->ds* == "1234", **px* will get 12.34 rounded to storage precision. *pd->ds* cannot have more than **DECIMAL_STRING_LENGTH-1** significant digits because one character is used to terminate the string with a null character. If *pd->more != 0* on input then additional nonzero digits follow those in *pd->ds; fp_inexact* is set accordingly on output in **ps*.

**px* is correctly rounded according to the IEEE rounding modes in *pm->rd*. **ps* is set to contain *fp_inexact*, *fp_underflow*, or *fp_overflow* if any of these arise.

pd->ndigits, *pm->df*, and *pm->ndigits* are not used.

strtod(3), **scanf**(3V), fscanf( ), and sscanf( ) all use **decimal_to_double**( ).

SEE ALSO

scanf(3V), **strtod**(3)

NAME

des_crypt, ecb_crypt, cbc_crypt, des_setparity – fast DES encryption

SYNOPSIS

```
#include <des_crypt.h>

int ecb_crypt(key, data, datalen, mode)
char *key;
char *data;
unsigned datalen;
unsigned mode;

int cbc_crypt(key, data, datalen, mode, ivec)
char *key;
char *data;
unsigned datalen;
unsigned mode;
char *ivec;

void des_setparity(key)
char *key;
```

DESCRIPTION

ecb_crypt() and **cbc_crypt()** implement the NBS DES (Data Encryption Standard). These routines are faster and more general purpose than **crypt**(3). They also are able to utilize DES hardware if it is available. **ecb_crypt()** encrypts in ECB (Electronic Code Book) mode, which encrypts blocks of data independently. **cbc_crypt()** encrypts in CBC (Cipher Block Chaining) mode, which chains together successive blocks. CBC mode protects against insertions, deletions and substitutions of blocks. Also, regularities in the clear text will not appear in the cipher text.

Here is how to use these routines. The first parameter, *key*, is the 8-byte encryption key with parity. To set the key's parity, which for DES is in the low bit of each byte, use *des_setparity*. The second parameter, *data*, contains the data to be encrypted or decrypted. The third parameter, *datalen*, is the length in bytes of *data*, which must be a multiple of 8. The fourth parameter, *mode*, is formed by OR'ing together some things. For the encryption direction 'or' in either DES_ENCRYPT or DES_DECRYPT. For software versus hardware encryption, 'or' in either DES_HW or DES_SW. If DES_HW is specified, and there is no hardware, then the encryption is performed in software and the routine returns DESERR_NOHWDEVICE. For *cbc_crypt*, the parameter *ivec* is the 8-byte initialization vector for the chaining. It is updated to the next initialization vector upon return.

SEE ALSO

des(1), **crypt**(3)

DIAGNOSTICS

DESERR_NONE	No error.
DESERR_NOHWDEVICE	Encryption succeeded, but done in software instead of the requested hardware.
DESERR_HWERR	An error occurred in the hardware or driver.
DESERR_BADPARAM	Bad parameter to routine.

Given a result status *stat*, the macro DES_FAILED(stat) is false only for the first two statuses.

RESTRICTIONS

These routines are not available for export outside the U.S.

NAME

directory, opendir, readdir, telldir, seekdir, rewinddir, closedir – directory operations

SYNOPSIS

```
#include <dirent.h>

DIR *opendir(dirname)
char *dirname;

struct dirent *readdir(dirp)
DIR *dirp;

long telldir(dirp)
DIR *dirp;

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

void rewinddir(dirp)
DIR *dirp;

int closedir(dirp)
DIR *dirp;
```

SYSTEM V SYNOPSIS

For XPG2 conformance, use:

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

DESCRIPTION

opendir( ) opens the directory named by *dirname* and associates a *directory stream* with it. **opendir( )** returns a pointer to be used to identify the directory stream in subsequent operations. A NULL pointer is returned if *dirname* cannot be accessed or is not a directory, or if it cannot **malloc**(3V) enough memory to hold the whole thing.

readdir( ) returns a pointer to the next directory entry. It returns NULL upon reaching the end of the directory or detecting an invalid **seekdir( )** operation.

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. Values returned by **telldir( )** are good only for the lifetime of the DIR pointer from which they are derived. If the directory is closed and then reopened, the **telldir( )** value may be invalidated due to undetected directory compaction. It is safe to use a previous **telldir( )** value immediately after a call to **opendir( )** and before any calls to **readdir**.

rewinddir( ) resets the position of the named directory stream to the beginning of the directory. I also causes the directory stream to refer to the current state of the corresponding directory, as a call to **opendir( )** would have done.

closedir( ) closes the named directory stream and frees the structure associated with the DIR pointer.

RETURN VALUES

opendir( ) returns a pointer to an object of type **DIR** on success. On failure, it returns NULL and sets **errno** to indicate the error.

readdir( ) returns a pointer to an object of type **struct dirent** on success. On failure, it returns NULL and sets **errno** to indicate the error. When the end of the directory is encountered, **readdir**( ) returns NULL and leaves **errno** unchanged.

closedir( ) returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

telldir( ) returns the current location associated with the specified directory stream.

ERRORS

If any of the following conditions occur, **opendir**( ) sets **errno** to:

EACCES Search permission is denied for a component of *dirname*.

Read permission is denied for *dirname*.

ENAMETOOLONG The length of *dirname* exceeds {PATH_MAX}.

A pathname component is longer than {NAME_MAX} (see **sysconf**(2V)) while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).

ENOENT The named directory does not exist.

ENOTDIR A component of *dirname* is not a directory.

for each of the following conditions, when the condition is detected, **opendir**( ) sets **errno** to one of the following:

EMFILE Too many file descriptors are currently open for the process.

ENFILE Too many file descriptors are currently open in the system.

For each of the following conditions, when the condition is detected, **readdir**( ) sets **errno** to the following:

EBADF *dirp* does not refer to an open directory stream.

For each of the following conditions, when the condition is detected, **closedir**( ) sets **errno** to the following:

EBADF *dirp* does not refer to an open directory stream.

SYSTEM V ERRORS

In addition to the above, **opendir**( ) may set **errno** to the following:

ENOENT *dirname* points to an empty string.

EXAMPLES

Sample code which searchs a directory for entry "name" is:

```
dirp = opendir(".");
for (dp = readdir(dirp); dp != NULL; dp = readdir(dirp))
        if (!strcmp(dp->d_name, name)) {
                closedir (dirp);
                return FOUND;
        }
closedir (dirp);
return NOT_FOUND;
```

SEE ALSO

close(2V), **lseek**(2V), **open**(2V), **read**(2V), **getwd**(3), **malloc**(3V), **dir**(5)

NOTES

The **directory** library routines now use a new include file, **<dirent.h>**. This replaces the file, **<sys/dir.h>**, used in previous releases. Furthermore, with the use of this new file, the **readdir**( ) routine returns directory entries whose structure is named **struct dirent** rather than **struct direct** as before. The file **<sys/dir.h>** is retained in the current SunOS release for purposes of backwards source code compatibility; programs which use the **directory**( ) library and **<sys/dir.h>** will continue to compile and run without source code modifications. However, existing programs should convert to the use of the new include file, **<dirent.h>**, as **<sys/dir.h>** will be removed in a future major release.

The *X/Open Portability Guide, issue 2* (XPG2) requires **<sys/dirent.h>** rather than **<dirent.h>**. **/usr/xpg2include/sys/dirent.h** is functionally equivalent to **/usr/include/dirent.h**. In future SunOS releases, X/Open conformance will require **<dirent.h>**.

NAME

dlopen, dlsym, dlerror, dlclose – simple programmatic interface to the dynamic linker

SYNOPSIS

```
#include <dlfcn.h>

void *dlopen(path, mode)
char *path; int mode;

void *dlsym(handle, symbol)
void *handle; char *symbol;

char *dlerror()

int dlclose(handle);
void *handle;
```

DESCRIPTION

These functions provide a simple programmatic interface to the services of the dynamic link-editor. Operations are provided to add a new shared object to an program's address space, obtain the address bindings of symbols defined by such objects, and to remove such objects when their use is no longer required.

dlopen() provides access to the shared object in *path*, returning a descriptor that can be used for later references to the object in calls to **dlsym()** and **dlclose()**. If *path* was not in the address space prior to the call to **dlopen()**, then it will be placed in the address space, and if it defines a function with the name *_init* that function will be called by **dlopen()**. If, however, *path* has already been placed in the address space in a previous call to **dlopen()**, then it will not be added a second time, although a count of **dlopen()** operations on *path* will be maintained. *mode* is an integer containing flags describing options to be applied to the opening and loading process — it is reserved for future expansion and must always have the value 1. A null pointer supplied for *path* is interpreted as a reference to the "main" executable of the process. If **dlopen()** fails, it will return a null pointer.

dlsym() returns the address binding of the symbol described in the null-terminated character string *symbol* as it occurs in the shared object identified by *handle*. The symbols exported by objects added to the address space by **dlopen()** can be accessed *only* through calls to **dlsym()**, such symbols do not supersede any definition of those symbols already present in the address space when the object is loaded, nor are they available to satisfy "normal" dynamic linking references. **dlsym()** returns a null pointer if the symbol can not be found. A null pointer supplied as the value of *handle* is interpreted as a reference to the executable from which the call to **dlsym()** is being made — thus a shared object can reference its own symbols.

dlerror returns a null-terminated character string describing the last error that occurred during a **dlopen()**, **dlsym()**, or **dlclose()**. If no such error has occurred, then **dlerror()** will return a null pointer. At each call to **dlerror()**, the "last error" indication will be reset, thus in the case of two calls to **dlerror()**, and where the second call follows the first immediately, the second call will always return a null pointer.

dlclose() deletes a reference to the shared object referenced by *handle*. If the reference count drops to 0, then if the object referenced by *handle* defines a function *_fini*, that function will be called, the object removed from the address space, and *handle* destroyed. If **dlclose()** is successful, it will return a value of 0. A failing call to **dlclose()** will return a non-zero value.

The object-intrinsic functions *_init* and *_fini* are called with no arguments and treated as though their types were **void**.

These functions are obtained by specifying **–ldl** as an option to **ld**(1).

SEE ALSO

ld(1), **link**(5)

NAME

drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48 – generate uniformly distributed pseudo-random numbers

SYNOPSIS

double drand48( )

double erand48(xsubi)
unsigned short xsubi[3];

long lrand48( )

long nrand48(xsubi)
unsigned short xsubi[3];

long mrand48( )

long jrand48(xsubi)
unsigned short xsubi[3];

void srand48(seedval)
long seedval;

unsigned short *seed48(seed16v)
unsigned short seed16v[3];

void lcong48(param)
unsigned short param[7];

DESCRIPTION

This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

drand48( ) and **erand48( )** return non-negative double-precision floating-point values uniformly distributed over the interval [0.0, 1.0).

lrand48( ) and **nrand48( )** return non-negative long integers uniformly distributed over the interval $[0, 2^{31})$.

mrand48( ) and **jrand48( )** return signed long integers uniformly distributed over the interval $[-2^{31}, 2^{31})$.

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. **erand48( )**, **nrand48( )**, and **jrand48( )** do not require an initialization entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values, X_i, according to the linear congruential formula

$$X_{n+1} = (aX_n + c)_{\mathrm{mod}\ m} \qquad n \geq 0.$$

The parameter $m = 2^{48}$; hence 48-bit integer arithmetic is performed. Unless **lcong48( )** has been invoked, the multiplier value a and the addend value c are given by

$$a = \mathrm{5DEECE66D}_{16} = 273673163155_8$$
$$c = \mathrm{B}_{16} = 13_8.$$

The value returned by any of the functions **drand48( )**, **erand48( )**, **lrand48( )**, **nrand48( )**, **mrand48( )**, or **jrand48( )** is computed by first generating the next 48-bit X_i in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (leftmost) bits of X_i and transformed into the returned value.

drand48( ), **lrand48( )**, and **mrand48( )** store the last 48-bit X_i generated in an internal buffer; that is why they must be initialized prior to being invoked. The functions **erand48( )**, **nrand48( )**, and **jrand48( )** require the calling program to provide storage for the successive X_i values in the array specified as an argument when the functions are invoked. That is why these routines do not have to be initialized; the calling program merely has to place the desired initial value of X_i into the array and pass it as an argument. By using different arguments, functions **erand48()**, **nrand48()**, and **jrand48()** allow separate modules of a large program to generate several *independent* streams of pseudo-random numbers, that is, the sequence of numbers in each stream will *not* depend upon how many times the routines have been called to generate numbers for the other streams.

The initializer function **srand48()** sets the high-order 32 bits of X_i to the 32 bits contained in its argument. The low-order 16 bits of X_i are set to the arbitrary value $330E_{16}$.

The initializer function **seed48()** sets the value of X_i to the 48-bit value specified in the argument array. In addition, the previous value of X_i is copied into a 48-bit internal buffer, used only by **seed48()**, and a pointer to this buffer is the value returned by **seed48()**. This returned pointer, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time — use the pointer to get at and store the last X_i value, and then use this value to reinitialize via **seed48()** when the program is restarted.

The initialization function **lcong48()** allows the user to specify the initial X_i, the multiplier value a, and the addend value c. Argument array elements *param*[0-2] specify X_i, *param*[3-5] specify the multiplier a, and *param*[6] specifies the 16-bit addend c. After **lcong48()** has been called, a subsequent call to either **srand48()** or **seed48()** will restore the "standard" multiplier and addend values, a and c, specified on the previous page.

SEE ALSO

rand(3V)

NAME

econvert, fconvert, gconvert, seconvert, sfconvert, sgconvert, ecvt, fcvt, gcvt – output conversion

SYNOPSIS

```
#include <floatingpoint.h>

char *econvert(value, ndigit, decpt, sign, buf)
double value;
int ndigit, *decpt, *sign;
char *buf;

char *fconvert(value, ndigit, decpt, sign, buf)
double value;
int ndigit, *decpt, *sign;
char *buf;

char *gconvert(value, ndigit, trailing, buf)
double value;
int ndigit;
int trailing;
char *buf;

char *seconvert(value, ndigit, decpt, sign, buf)
single *value;
int ndigit, *decpt, *sign;
char *buf;

char *sfconvert(value, ndigit, decpt, sign, buf)
single *value;
int ndigit, *decpt, *sign;
char *buf;

char *sgconvert(value, ndigit, trailing, buf)
single *value;
int ndigit;
int trailing;
char *buf;

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

econvert( ) converts the *value* to a null-terminated string of *ndigit* ASCII digits in *buf* and returns a pointer to *buf*. *buf* should contain at least *ndigit+1* characters. The position of the radix character relative to the beginning of the string is stored indirectly through *decpt*. Thus *buf* == "314" and **decpt* == 1 corresponds to the numerical value 3.14, while *buf* == "314" and **decpt* == –1 corresponds to the numerical value .0314. If the sign of the result is negative, the word pointed to by *sign* is nonzero; otherwise it is zero. The least significant digit is rounded.

fconvert works much like **econvert**, except that the correct digit has been rounded as if for **sprintf(%w.nf)** output with *n=ndigit* digits to the right of the radix character. *ndigit* can be negative to indicate rounding to the left of the radix character. The return value is a pointer to *buf*. *buf* should contain at least *310+max(0,ndigit)* characters to accomodate any double-precision *value*.

gconvert( ) converts the *value* to a null-terminated ASCII string in *buf* and returns a pointer to *buf*. It produces *ndigit* significant digits in fixed-decimal format, like **sprintf(%w.nf)**, if possible, and otherwise in floating-decimal format, like **sprintf(%w.ne)**; in either case *buf* is ready for printing, with sign and exponent. The result corresponds to that obtained by

```
(void) sprintf(buf, "%w.ng", value);
```

If *trailing*= 0, trailing zeros and a trailing point are suppressed, as in **sprintf(%g)**. If *trailing*!= 0, trailing zeros and a trailing point are retained, as in **sprintf(%#g)**.

seconvert, **sfconvert**, and **sgconvert( )** are single-precision versions of these functions, and are more efficient than the corresponding double-precision versions. A pointer rather than the value itself is passed to avoid **C**'s usual conversion of single-precision arguments to double.

ecvt( ) and **fcvt( )** are obsolete versions of **econvert( )** and **fconvert( )** that create a string in a static data area, overwritten by each call, and return values that point to that static data. These functions are therefore not reentrant.

gcvt( ) is an obsolete version of **gconvert( )** that always suppresses trailing zeros and point.

IEEE Infinities and NaNs are treated similarly by these functions. ''NaN'' is returned for NaN, and ''Inf'' or ''Infinity'' for Infinity. The longer form is produced when *ndigit* >= 8.

The radix character is determined by the current setting of the program's locale (category LC_NUMERIC). In the **"C"** locale or if the locale is undefined, the readix character defaults to a period '**.**'.

SEE ALSO

printf(3V)

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 it is reset by the routines **brk**(2), **malloc**(3V), standard input/output (**stdio**(3V)), the profile (**–p**) option of cc(1V), and so on. Thus, the current value of the program break should be determined by **sbrk(0)** (see **brk**(2)).

SEE ALSO

cc(1V), **brk**(2), **malloc**(3V), **stdio**(3V)

NAME

ethers, ether_ntoa, ether_aton, ether_ntohost, ether_hostton, ether_line – Ethernet address mapping operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>

char *
ether_ntoa(e)
struct ether_addr *e;

struct ether_addr *ether_aton(s)
char *s;

ether_ntohost(hostname, e)
char *hostname;
struct ether_addr *e;

ether_hostton(hostname, e)
char *hostname;
struct ether_addr *e;

ether_line(l, e, hostname)
char *l;
struct ether_addr *e;
char *hostname;
```

DESCRIPTION

These routines are useful for mapping 48 bit Ethernet numbers to their ASCII representations or their corresponding host names, and vice versa.

The function **ether_ntoa( )** converts a 48 bit Ethernet number pointed to by *e* to its standard ACSII representation; it returns a pointer to the ASCII string. The representation is of the form: *x*:*x*:*x*:*x*:*x*:*x* where *x* is a hexadecimal number between 0 and ff. The function **ether_aton( )** converts an ASCII string in the standard representation back to a 48 bit Ethernet number; the function returns NULL if the string cannot be scanned successfully.

The function **ether_ntohost( )** maps an Ethernet number (pointed to by *e*) to its associated hostname. The string pointed to by **hostname** must be long enough to hold the hostname and a null character. The function returns zero upon success and non-zero upon failure. Inversely, the function **ether_hostton( )** maps a hostname string to its corresponding Ethernet number; the function modifies the Ethernet number pointed to by *e*. The function also returns zero upon success and non-zero upon failure.

The function **ether_line( )** scans a line (pointed to by *l*) and sets the hostname and the Ethernet number (pointed to by *e*). The string pointed to by **hostname** must be long enough to hold the hostname and a null character. The function returns zero upon success and non-zero upon failure. The format of the scanned line is described by **ethers(5)**.

FILES

/etc/ethers	(or the Network Information Service (NIS) maps **ethers.byaddr** and **ethers.byname**)

SEE ALSO

ethers(5)

NOTES

The Network Information Service (NIS) was formerly known as Sun Yellow Pages (YP). The functionality of the two remains the same; only the name has changed.

NAME

execl, execv, execle, 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 execlp(file, arg0 [ , arg1,...  , argn ] (char *)0)
char *file, *arg0, *arg1, ..., *argn;

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

extern char **environ;
```

DESCRIPTION

These routines provide various interfaces to the **execve( )** system call. Refer to **execve(2V)** for a description of their properties; only brief descriptions are provided here.

exec( ) in all its forms overlays the calling process with the named file, then transfers to the entry point of the core image of the file. There can be no return from a successful **exec( )**; the calling core image is lost.

The *filename* argument is a pointer to the name of the file to be executed. The pointers *arg* [0], *arg* [1] ... address null-terminated strings. Conventionally *arg* [0] is the name of the file.

Two interfaces are available. **execl( )** is useful when a known file with known arguments is being called; the arguments to **execl( )** are the character strings constituting the file and the arguments; the first argument is conventionally the same as the file name (or its last component). A **(char *)0** argument must end the argument list. The cast to type **char *** insures portability.

The **execv( )** version is useful when the number of arguments is unknown in advance; the arguments to **execv( )** are the name of the file to be executed and a vector of strings containing the arguments. The last argument string must be followed by a 0 pointer.

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

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

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. As indicated, *argc* is conventionally at least one and the first member of the array points to a string containing the name of the file.

argv is directly usable in another **execv( )** because *argv* [*argc*] is 0.

envp is a pointer to an array of strings that constitute the *environment* of the process. Each string consists of a name, an '=', and a null-terminated value. The array of pointers is terminated by a NULL pointer. The shell **sh**(1) passes an environment entry for each global shell variable defined when the program is called. See **environ**(5V) for some conventionally used names. The C run-time start-off routine places a copy of *envp* in the global cell *environ*, which is used by **execv( )** and **execl( )** to pass the environment to any subprograms executed by the current program.

execlp( ) and **execvp( )** are called with the same arguments as **execl( )** and **execv( )**, but duplicate the shell's actions in searching for an executable *file* in a list of directories. The directory list is obtained from the environment.

RETURN VALUES

These functions return to the calling process only on failure. They return −1 and set **errno** to indicate the error if *path* or *file* cannot be found, if it is not executable, if it does not start with a valid magic number (see **a.out**(5)), if maximum memory is exceeded, or if the arguments require too much space. Even for the super-user, at least one of the execute-permission bits must be set for a file to be executed.

ERRORS

If any of the following conditions occur, these functions will return and set **errno** to one of the following:

E2BIG The number of bytes used by the new process image's argument list and environment list is greater than {ARG_MAX} bytes (see **sysconf**(2V)).

EACCES Search permission is denied for a directory listed in the new process image file's path prefix.

The new process image file denies execution permission.

The new process image file is not a regular file.

ENAMETOOLONG The length of the *path* or *file*, or an element of the environment variable **PATH** prefixed to a file, exceeds {PATH_MAX}.

A pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect for that file (see **pathconf**(2V)).

ENOENT One or more components of the new process image file's pathname do not exist.

ENOTDIR A component of the new process image file's path prefix is not a directory.

if the following condition occurs, **execl**( ), **execv**( ), and **execle**( ) set **errno** to:

ENOEXEC The new process image file has the appropriate access permission, but is not in the proper format.

If the following condition is detected, the exec functions set **errno** to:

ENOMEM The new process image requires more memory than there is swap space available.

On Sun-3 systems, the new process image requires more than 2^{31} bytes.

SYSTEM V ERRORS

In addition to the above, if the following condition occurs, the exec functions set **errno** to:

ENOENT *path* or *file* points to a null pathname.

FILES

/usr/bin/sh shell, invoked if command *file* found by **execlp**( ) or **execvp**( )

SEE ALSO

csh(1), **sh**(1), **execve**(2V), **fork**(2V), **pathconf**(2V), **sysconf**(2V), **a.out**(5), **environ**(5V)

Programming Utilities and Libraries

NAME

exit – terminate a process after performing cleanup

SYNOPSIS

void
exit(status)
int status;

DESCRIPTION

exit( ) terminates a process by calling **exit**(2V) after calling any termination handlers named by calls to **on_exit**. Normally, this is just the Standard I/O library function **_cleanup**. **exit**( ) never returns.

SEE ALSO

exit(2V), **intro**(3), **on_exit**(3)

NAME

exportent, getexportent, setexportent, addexportent, remexportent, endexportent, getexportopt – get exported file system information

SYNOPSIS

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

FILE *setexportent( )

struct exportent *getexportent(filep)
FILE *filep;

int addexportent(filep, dirname, options)
FILE *filep;
char *dirname;
char *options;

int remexportent(filep, dirname)
FILE *filep;
char *dirname;

char *getexportopt(xent, opt)
struct exportent *xent;
char *opt;

void endexportent(filep)
FILE *filep;
```

DESCRIPTION

These routines access the exported filesystem information in **/etc/xtab**.

setexportent( ) opens the export information file and returns a file pointer to use with **getexportent**, **addexportent**, **remexportent**, and **endexportent**. **getexportent( )** reads the next line from *filep* and returns a pointer to an object with the following structure containing the broken-out fields of a line in the file, **/etc/xtab** The fields have meanings described in **exports**(5).

```
#define ACCESS_OPT  "access"  /* machines that can mount fs */
#define ROOT_OPT    "root"    /* machines with root access of fs */
#define RO_OPT      "ro"      /* export read-only */
#define ANON_OPT    "anon"    /* uid for anonymous requests */
#define SECURE_OPT  "secure"  /* require secure NFS for access */
#define WINDOW_OPT  "window"  /* expiration window for credential */

struct exportent {
        char *xent_dirname;     /* directory (or file) to export */
        char *xent_options;     /* options, as above */
};
```

addexportent( ) adds the **exportent( )** to the end of the open file *filep*. It returns 0 if successful and –1 on failure. **remexportent( )** removes the indicated entry from the list. It also returns 0 on success and –1 on failure. **getexportopt( )** scans the *xent_options* field of the **exportent( )** structure for a substring that matches *opt*. It returns the string value of *opt*, or NULL if the option is not found.

endexportent( ) closes the file.

FILES

/etc/exports
/etc/xtab

SEE ALSO

exports(5), **exportfs**(8)

DIAGNOSTICS

NULL pointer (0) returned on EOF or error.

BUGS

The returned **exportent**( ) structure points to static information that is overwritten in each call.

NAME

fclose, fflush – close or flush a stream

SYNOPSIS

```
#include <stdio.h>

fclose(stream)
FILE *stream;

fflush(stream)
FILE *stream;
```

DESCRIPTION

fclose( ) writes out any buffered data for the named stream, and closes the named stream. Buffers allocated by the standard input/output system are freed.

fclose( ) is performed automatically for all open files upon calling **exit**(3).

fflush( ) writes any unwritten data for an output stream or an update stream in which the most recent operation was not input to be delivered to the host environment to the file; otherwise it is ignored. The named stream remains open.

SYSTEM V DESCRIPTION

When **fflush( )** is called on a stream opened for reading, any unread data buffered in the stream is invalidated. When **fflush( )** is called on a stream opened for reading, if the file is not already at EOF, and the file is one capable of seeking, the file offset of the underlying open file description is adjusted so the next operation on the open file description deals with the byte after the last byte read from or written to the stream being flushed.

RETURN VALUES

fclose( ) and **fflush( )** return:

0	on success.
EOF	if any error (such as trying to write to a file that has not been opened for writing) was detected.

SEE ALSO

close(2V), **exit**(3), **fopen**(3V), **setbuf**(3V)

NAME

ferror, feof, clearerr, fileno – stream status inquiries

SYNOPSIS

```
#include <stdio.h>

ferror(stream)
FILE *stream;

feof(stream)
FILE *stream;

clearerr(stream)
FILE *stream;

fileno(stream)
FILE *stream;
```

DESCRIPTION

ferror( ) returns non-zero when an error has occurred reading from or writing to the named stream, otherwise zero. Unless cleared by **clearerr( )**, the error indication lasts until the stream is closed.

feof( ) returns non-zero when EOF has previously been detected reading the named input stream, otherwise zero. Unless cleared by **clearerr( )**, the EOF indication lasts until the stream is closed.

clearerr( ) resets the error indication and EOF indication to zero on the named stream.

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

SYSTEM V DESCRIPTION

feof( ) returns non-zero when EOF has previously been detected reading the named input stream, otherwise zero. Unless cleared by **clearerr( )**, the EOF indication lasts until the stream is closed, however, operations which attempt to read from the stream will ignore the current state of the EOF indication and attempt to read from the file descriptor associated with the stream.

SEE ALSO

open(2V), **fopen**(3V)

NOTES

These functions are defined in the C library and are also defined as macros in **<stdio.h>**.

NAME

single_to_decimal, double_to_decimal, extended_to_decimal – convert floating-point value to decimal record

SYNOPSIS

```
#include <floatingpoint.h>

void single_to_decimal(px, pm, pd, ps)
single *px ;
decimal_mode *pm;
decimal_record *pd;
fp_exception_field_type *ps;

void double_to_decimal(px, pm, pd, ps)
double *px ;
decimal_mode *pm;
decimal_record *pd;
fp_exception_field_type *ps;

void extended_to_decimal(px, pm, pd, ps)
extended *px ;
decimal_mode *pm;
decimal_record *pd;
fp_exception_field_type *ps;
```

DESCRIPTION

The **floating_to_decimal( )** functions convert the floating-point value at *px into a decimal record at *pd, observing the modes specified in *pm and setting exceptions in *ps. If there are no IEEE exceptions, *ps will be zero.

If *px is zero, infinity, or NaN, then only pd->sign and pd->fpclass are set. Otherwise pd->exponent and pd->ds are also set so that

(pd->sign)*(pd->ds)*10(pd->exponent)**

is a correctly rounded approximation to *px. pd->ds has at least one and no more than **DECIMAL_STRING_LENGTH–1** significant digits because one character is used to terminate the string with a null character.

pd->ds is correctly rounded according to the IEEE rounding modes in pm->rd. *ps has fp_inexact set if the result was inexact, and has fp_overflow set if the string result does not fit in pd->ds because of the limitation **DECIMAL_STRING_LENGTH.**

If pm->df == floating_form, then pd->ds always contains pm->ndigits significant digits. Thus if *px == 12.34 and pm->ndigits == 8, then pd->ds will contain 12340000 and pd->exponent will contain –6.

If pm->df == fixed_form and pm->ndigits >= 0, then pd->ds always contains pm->ndigits after the point and as many digits as necessary before the point. Since the latter is not known in advance, the total number of digits required is returned in pd->ndigits; if that number >= **DECIMAL_STRING_LENGTH**, then ds is undefined. pd->exponent always gets –pm->ndigits. Thus if *px == 12.34 and pm->ndigits == 1, then pd->ds gets 123, pd->exponent gets –1, and pd->ndigits gets 3.

If pm->df == fixed_form and pm->ndigits < 0, then pm->ds always contains –pm->ndigits trailing zeros; in other words, rounding occurs –pm->ndigits to the left of the decimal point, but the digits rounded away are retained as zeros. The total number of digits required is in pd->ndigits. pd->exponent always gets 0. Thus if *px == 12.34 and pm->ndigits == –1, then pd->ds gets 10, pd->exponent gets 0, and pd->ndigits gets 2.

pd->more is not used.

econvert( ), **fconvert( )** and **gconvert( )** (see **econvert**(3)), and **printf( )** and **sprintf( )** (see **printf**(3V)) all use **double_to_decimal( )**.

SEE ALSO

econvert(3), **printf**(3V)

NAME

floatingpoint – IEEE floating point definitions

SYNOPSIS

#include <sys/ieeefp.h>
#include <floatingpoint.h>

DESCRIPTION

This file defines constants, types, variables, and functions used to implement standard floating point according to ANSI/IEEE Std 754-1985. The variables and functions are implemented in **libc.a**. The included file **<sys/ieeefp.h>** defines certain types of interest to the kernel.

IEEE Rounding Modes:

fp_direction_type The type of the IEEE rounding direction mode. Note: the order of enumeration varies according to hardware.

fp_direction The IEEE rounding direction mode currently in force. This is a global variable that is intended to reflect the hardware state, so it should only be written indirectly through a function like **ieee_flags ("set","direction",...)** that also sets the hardware state.

fp_precision_type The type of the IEEE rounding precision mode, which only applies on systems that support extended precision such as Sun-3 systems with 68881's.

fp_precision The IEEE rounding precision mode currently in force. This is a global variable that is intended to reflect the hardware state on systems with extended precision, so it should only be written indirectly through a function like **ieee_flags("set","precision",...)**.

SIGFPE handling:

sigfpe_code_type The type of a **SIGFPE** code.

sigfpe_handler_type The type of a user-definable **SIGFPE** exception handler called to handle a particular **SIGFPE** code.

SIGFPE_DEFAULT A macro indicating the default **SIGFPE** exception handling, namely to perform the exception handling specified by calls to **ieee_handler**(3M), if any, and otherwise to dump core using **abort**(3).

SIGFPE_IGNORE A macro indicating an alternate **SIGFPE** exception handling, namely to ignore and continue execution.

SIGFPE_ABORT A macro indicating an alternate **SIGFPE** exception handling, namely to abort with a core dump.

IEEE Exception Handling:

N_IEEE_EXCEPTION The number of distinct IEEE floating-point exceptions.

fp_exception_type The type of the **N_IEEE_EXCEPTION** exceptions. Each exception is given a bit number.

fp_exception_field_type

The type intended to hold at least **N_IEEE_EXCEPTION** bits corresponding to the IEEE exceptions numbered by **fp_exception_type**. Thus **fp_inexact** corresponds to the least significant bit and **fp_invalid** to the fifth least significant bit. Note: some operations may set more than one exception.

fp_accrued_exceptions

The IEEE exceptions between the time this global variable was last cleared, and the last time a function like **ieee_flags("get","exception",...)** was called to update the variable by obtaining the hardware state.

ieee_handlers An array of user-specifiable signal handlers for use by the standard **SIGFPE** handler for IEEE arithmetic-related **SIGFPE** codes. Since IEEE trapping modes correspond to hardware modes, elements of this array should only be modified with a function like **ieee_handler**(3M) that performs the appropriate hardware mode update. If no **sigfpe_handler** has been declared for a particular IEEE-related **SIGFPE** code, then the related **ieee_handlers** will be invoked.

IEEE Formats and Classification:

single ;*extended* Definitions of IEEE formats.

fp_class_type An enumeration of the various classes of IEEE values and symbols.

IEEE Base Conversion:

The functions described under **floating_to_decimal**(3) and **decimal_to_floating**(3) not only satisfy the IEEE Standard, but also the stricter requirements of correct rounding for all arguments.

DECIMAL_STRING_LENGTH

The length of a **decimal_string**.

decimal_string The digit buffer in a **decimal_record**.

decimal_record The canonical form for representing an unpacked decimal floating-point number.

decimal_form The type used to specify fixed or floating binary to decimal conversion.

decimal_mode A struct that contains specifications for conversion between binary and decimal.

decimal_string_form An enumeration of possible valid character strings representing floating-point numbers, infinities, or NaNs.

SEE ALSO

abort(3), **decimal_to_floating**(3), **econvert**(3), **floating_to_decimal**(3), **ieee_flags**(3M), **ieee_handler**(3M), **sigfpe**(3), **string_to_decimal**(3), **strtod**(3)

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(fd, type)
int fd;
char *type;
```

DESCRIPTION

fopen() opens the file named by *filename* and associates a stream with it. If the open succeeds, **fopen()** returns a pointer to be used to identify the stream in subsequent operations.

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 EOF

freopen() opens the file named by *filename* and associates the stream pointed to by *stream* with it. The *type* argument is used just as in **fopen**. The original stream is closed, regardless of whether the open ultimately succeeds. If the open succeeds, **freopen()** returns the original value of *stream*.

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

fdopen() associates a stream with the file descriptor *fd*. File descriptors are obtained from calls like **open**(2V), **dup**(2V), **creat**(2V), or **pipe**(2V), which open files but do not return streams. Streams are necessary input for many of the Section 3S library routines. The *type* of the stream must agree with the access permissions 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**(3S) or **rewind()**, and input may not be directly followed by output without an intervening **fseek()**, **rewind()**, or an input operation which encounters EOF.

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.

SYSTEM V DESCRIPTION

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

RETURN VALUES

On success, **fopen()**, **freopen()**, and **fdopen()** return a pointer to **FILE** which identifies the opened stream. On failure, they return NULL.

SEE ALSO

open(2V), **pipe**(2V), **fclose**(3V), **fseek**(3S)

BUGS

In order to support the same number of open files that the system does, **fopen()** must allocate additional memory for data structures using **calloc()** after 64 files have been opened. This confuses some programs which use their own memory allocators.

NAME

fread, fwrite – buffered binary input/output

SYNOPSIS

```
#include <stdio.h>

int fread (ptr, size, nitems, stream)
char *ptr;
int size;
int nitems;
FILE *stream;

int fwrite (ptr, size, nitems, stream)
char *ptr;
int size;
int nitems;
FILE *stream;
```

DESCRIPTION

fread( ) reads, into a block pointed to by *ptr*, *nitems* items of data from the named input stream *stream*, where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. It returns the number of items actually read. **fread( )** stops reading if an end-of-file or error condition is encountered while reading from *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 the file referred to by *stream* .

fwrite( ) writes at most *nitems* items of data from the block pointed to by *ptr* to the named output stream *stream*. It returns the number of items actually written. **fwrite( )** stops writing when it has written *nitems* items of data or if an error condition is encountered on *stream*. **fwrite( )** does not change the contents of the block pointed to by *ptr*.

If *size* or *nitems* is non-positive, no characters are read or written and 0 is returned by both **fread( )** and **fwrite( )**.

SEE ALSO

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

DIAGNOSTICS

fread( ) and **fwrite( )** return 0 upon end of file or error.

NAME

fseek, ftell, rewind – reposition a stream

SYNOPSIS

```
#include <stdio.h>

fseek(stream, offset, ptrname)
FILE *stream;
long offset;

long ftell(stream)
FILE *stream;

rewind(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, the current position, or the end of the file, according as *ptrname* has the value 0, 1, or 2.

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

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

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

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

SEE ALSO

lseek(2V), **fopen**(3V), **popen**(3S), **ungetc**(3S)

DIAGNOSTICS

fseek( ) returns –1 for improper seeks, otherwise zero. An improper seek can be, for example, an **fseek( )** done on a file associated with a non-seekable device, such as a tty or a pipe; in particular, **fseek( )** may not be used on a terminal, or on a file opened using **popen**(3S).

WARNING

Although on the UNIX system an offset returned by **ftell( )** is measured in bytes, and it is permissible to seek to positions relative to that offset, portability to a (non-UNIX) system 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

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 will be possible for unrelated processes to unintentionally interfere with each other's operation. Therefore, it is strongly suggested that the most significant byte of a key in some sense refer to a project so that keys do not conflict across a given system.

ftok() returns a key based on *path* and ID that is usable in subsequent **msgget**, **semget**, and **shmget**() system calls. *path* must be the path name of an existing file that is accessible to the process. ID is a character which uniquely identifies a project. Note: **ftok**() will return the same key for linked files when called with the same ID and that it will return different keys when called with the same file name but different IDs.

SEE ALSO

intro(2), **msgget**(2), **semget**(2), **shmget**(2)

DIAGNOSTICS

ftok() returns **(key_t) –1** if *path* does not exist or if it is not accessible to the process.

WARNING

If the file whose *path* is passed to **ftok**() is removed when keys still refer to the file, future calls to **ftok**() with the same *path* and ID will return an error. If the same file is recreated, then **ftok**() is likely to return a different key than it did the original time it was called.

NAME

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**(2V)) containing information about the object, and an integer. Possible values of the integer, defined in the **<ftw.h>** header file, are FTW_F for a file, FTW_D for a directory, FTW_DNR for a directory that cannot be read, and FTW_NS for an object for which **stat()** could not successfully be executed. If the integer is FTW_DNR, descendants of that directory will not be processed. If the integer is FTW_NS, the **stat()** structure will contain garbage. An example of an object that would cause FTW_NS to be passed to *fn* would be a file in a directory with read but without execute (search) permission.

ftw() visits a directory before visiting any of its descendants.

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

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

SEE ALSO

stat(2V), **malloc**(3V)

BUGS

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

It could be made to run faster and use less storage on deep structures at the cost of considerable complexity.

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

NAME

getacinfo, getacdir, getacflg, getacmin, setac, endac – get audit control file information

SYNOPSIS

```
int getacdir(dir, len)
char *dir;
int len;

int getacmin(min_val)
int *min_val;

int getacflg(auditstring, len)
char *auditstring;
int len;

void setac( )

void endac( )
```

DESCRIPTION

When first called, **getacdir( )** provides information about the first audit directory in the **audit_control** file; thereafter, it returns the next directory in the file. Successive calls list all the directories listed in **audit_control(5)** The parameter *len* specifies the length of the buffer *dir* . On return, *dir* points to the directory entry.

getacmin( ) reads the minimum value from the **audit_control** file and returns the value in **min_val**. The minimum value specifies how full the file system to which the audit files are being written can get before the script **audit_warn** is invoked.

getacflg( ) reads the system audit value from the **audit_control** file and returns the value in *auditstring*. The parameter *len* specifies the length of the buffer *auditstring*.

Calling *setac* rewinds the **audit_control** file to allow repeated searches.

Calling *endac* closes the **audit_control** file when processing is complete.

RETURN VALUES

getacdir( ), **getacflg( )** and **getacmin( )** return:

0 on success.

–2 on failure and set **errno** to indicate the error.

getacmin( ) and **getacflg( )** return:

1 on EOF.

getacdir( ) returns:

–1 on EOF.

2 if the directory search had to start from the beginning because one of the other functions was called between calls to **getacdir( )**.

These functions return:

–3 if the directory entry format in the **audit_control** file is incorrect.

getacdir( ) and **getacflg( )** return:

–3 if the input buffer is too short to accommodate the record.

SEE ALSO

audit_control(5)

NAME

getauditflagsbin, getauditflagschar – convert audit flag specifications

SYNOPSIS

```
#include <sys/label.h>
#include <sys/audit.h>
#include <sys/auevents.h>

int getauditflagsbin(auditstring, masks)
char *auditstring;
audit_state_t *masks;

int getauditflagschar(auditstring, masks, verbose)
char *auditstring;
audit_state_t *masks;
int verbose;
```

DESCRIPTION

getauditflagsbin() converts the character representation of audit values pointed to by *auditstring* into **audit_state_t** fields pointed to by *masks*. These fields indicate which events are to be audited when they succeed and which are to be audited when they fail. The character string syntax is described in **audit_control(5)**.

getauditflagschar() converts the **audit_state_t** fields pointed to by *masks* into a string pointed to by *auditstring*. If *verbose* is zero, the short (2-character) flag names are used. If *verbose* is non-zero, the long flag names are used. *auditstring* should be large enough to contain the ASCII representation of the events.

auditstring contains a series of event names, each one identifying a single audit class, separated by commas. The **audit_state_t** fields pointed to by *masks* correspond to binary values defined in *audit.h*.

DIAGNOSTICS

–1 is returned on error and 0 on success.

SEE ALSO

audit.log(5), audit_control(5)

BUGS

This is not a very extensible interface.

NAME

getc, getchar, fgetc, getw – get character or integer from stream

SYNOPSIS

```
#include <stdio.h>

int getc(stream)
FILE *stream;

int getchar( )

int fgetc(stream)
FILE *stream;

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

DESCRIPTION

getc( ) returns the next character (that is, 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 C **int** (*word*) 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.

RETURN VALUES

On success, **getc( )**, **getchar( )** and **fgetc( )** return the next character from the named input stream as an integer. On failure, or on EOF, they return EOF. The EOF condition is remembered, even on a terminal, and all subsequent operations which attempt to read from the stream will return EOF until the condition is cleared with **clearerr( )** (see **ferror**(3V)).

getw( ) returns the next C **int** from the named input stream on success. On failure, or on EOF, it returns EOF, but since EOF is a valid integer, use **ferror**(3V) to detect **getw( )** errors.

SYSTEM V RETURN VALUES

On failure, or on EOF, these functions return EOF. The EOF condition is remembered, even on a terminal, however, operations which attempt to read from the stream will ignore the current state of the EOF indication and attempt to read from the file descriptor associated with the stream.

SEE ALSO

ferror(3V), **fopen**(3V), **fread**(3S), **gets**(3S), **putc**(3S), **scanf**(3V), **ungetc**(3S)

WARNINGS

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.

BUGS

Because it is implemented as a macro, **getc( )** treats a stream argument with side effects incorrectly. 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 readable using **getw( )** on a different processor.

NAME

getcwd – get pathname of current working directory

SYNOPSIS

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

DESCRIPTION

getcwd( ) returns a pointer to the current directory pathname. The value of *size* must be at least two greater than the length of the pathname to be returned.

If *buf* is a NULL pointer, **getcwd( )** will obtain *size* bytes of space using **malloc**(3V). 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.

RETURN VALUES

getcwd( ) returns a pointer to the current directory pathname on success. If *size* is not large enough, or if an error occurs in a lower-level function, **getcwd( )** returns NULL and sets **errno** to indicate the error.

ERRORS

EINVAL	*size* is less than or equal to zero.
ERANGE	*size* is greater than zero, but is smaller than the length of the pathname plus 1.

If the following condition is detected, **getcwd( )** sets **errno** to:

EACCES	Read or search permission is denied for a component of the pathname.

EXAMPLES

```
char *cwd, *getcwd( );
.
.
.
if ((cwd = getcwd((char *)NULL, 64)) == NULL) {
        perror ("pwd");
        exit (1);
}
printf("%s\n", cwd);
```

SEE ALSO

pwd(1), **getwd**(3), **malloc**(3V), **popen**(3S)

BUGS

Since this function uses **popen( )** to create a pipe to the **pwd** command, it is slower than **getwd( )** and gives poorer error diagnostics. **getcwd( )** is provided only for compatibility with other UNIX operating systems.

NAME

getenv – return value for environment name

SYNOPSIS

```
#include <stdlib.h>

char *getenv(name)
char *name;
```

DESCRIPTION

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

RETURN VALUES

On success, **getenv( )** returns a pointer to a string containing the value for the specified *name*. If the specified *name* cannot be found, it returns NULL.

SEE ALSO

environ(5V), **execve**(2V), **putenv**(3)

NAME

getfauditflags – generates the process audit state

SYNOPSIS

```
#include <sys/types.h>
#include <sys/audit.h>
#include <sys/label.h>

void getfauditflags(usremasks, usrdmasks, lastmasks)
audit_state_t *usremasks;
audit_state_t *usrdmasks;
audit_state_t *lastmasks;
```

DESCRIPTION

getfauditflags generates the process audit state from the user audit value as input to **getfauditflags** and the system audit value as specified in the **audit_control** file. **getfauditflags** obtains the system audit value by calling **getacflg**. The user audit value, pointed to by *usremasks* and *usrdmasks* is passed into **getfauditflags.**

usremasks points to **audit_state_t** fields which contains two values. The first value defines which events are *always* to be audited when they succeed. The second value defines which events are always to be audited when they fail.

usrdmasks also points to **audit_state_t** fields which contains two values. The first value defines which events are *never* to be audited when they succeed. The second value defines which events are never to be audited when they fail.

The structures pointed to by *usremasks* and *usrdmasks* may be obtained from the **passwd.adjunct** file by calling **getpwaent( )** which returns a pointer to a strucure containing all **passwd.adjunct** fields for a user.

lastmasks points to **audit_state_t** as well. The first value defines which events are to be audited when they succeed and the second value defines which events are to be audited when they fail.

Both *usremasks* and *usrdmasks* override the values in the system audit values.

DIAGNOSTICS

-1 is returned on error and 0 on success.

SEE ALSO

getauditflags(3), **getacinfo**(3), **audit.log**(5), **audit_control**(5)

NAME

getfsent, getfsspec, getfsfile, getfstype, setfsent, endfsent – get file system descriptor file entry

SYNOPSIS

```
#include <fstab.h>

struct fstab *getfsent( )

struct fstab *getfsspec(spec)
char *spec;

struct fstab *getfsfile(file)
char *file;

struct fstab *getfstype(type)
char *type;

int setfsent( )

int endfsent( )
```

DESCRIPTION

These routines are included for compatibility with 4.2 BSD; they have been superseded by the **getmntent**(3) library routines.

getfsent, *getfsspec*, *getfstype*, and *getfsfile* each return a pointer to an object with the following structure containing the broken-out fields of a line in the file system description file, **<fstab.h>**.

```
struct fstab {
        char    *fs_spec;
        char    *fs_file;
        char    *fs_type;
        int     fs_freq;
        int     fs_passno;
};
```

The fields have meanings described in **fstab**(5).

getfsent( ) reads the next line of the file, opening the file if necessary.

getfsent( ) opens and rewinds the file.

endfsent closes the file.

getfsspec and *getfsfile* sequentially search from the beginning of the file until a matching special file name or file system file name is found, or until EOF is encountered. *getfstype* does likewise, matching on the file system type field.

FILES

/etc/fstab

SEE ALSO

fstab(5)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

The return value points to static information which is overwritten in each call.

NAME

getgraent, getgranam, setgraent, endgraent, fgetgraent – get group adjunct file entry

SYNOPSIS

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

struct group_adjunct *getgraent( )

struct group_adjunct *getgranam(name)
char *name;

struct group_adjunct *fgetgraent(f)
FILE *f;

void setgraent( )

void endgraent( )
```

DESCRIPTION

getgraent( ) and **getgranam( )** each return pointers to an object with the following structure containing the broken-out fields of a line in the group adjunct file. Each line contains a **group_adjunct** structure, defined in the **<grpadj.h>** header file.

```
struct  group_adjunct {
        char   *gra_name;        /* the name of the group */
        char   *gra_passwd;      /* the encrypted group password */
};
```

When first called, **getgraent( )** returns a pointer to a **group_adjunct** structure corresponding to the first line in the file. Thereafter, it returns a pointer to the next **group_adjunct** structure in the file. So successive calls may be used to traverse the entire file.

For locating a particular group, **getgranam( )** searches through the file until it finds group *filename*, then returns a pointer to that structure.

A call to **getgraent( )** rewinds the group adjunct file to allow repeated searches. A call to **endgraent( )** closes the group adjunct file when processing is complete.

Because read access is required on **/etc/security/group.adjunct**, **getgraent( )** and **getgranam( )** will fail unless the calling process has effective UID of root.

FILES

/etc/security/group.adjunct
/var/yp/*domainname***/group.adjunct**

SEE ALSO

getlogin(3V), **getgrent(3V)**, **getpwaent(3)**, **getpwent(3V)**, **ypserv(8)**

DIAGNOSTICS

A NULL pointer is returned on end-of-file or error.

BUGS

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

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 fields of a line in the group file. Each line contains a "group" structure, defined in **<grp.h>**.

```
struct  group {
        char    *gr_name;       /* name of the group */
        char    *gr_passwd;     /* encrypted password of the group */
        gid_t   gr_gid;         /* numerical group ID */
        char    **gr_mem;       /* null-terminated array of pointers to the
                                        individual 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 must refer to an open file in the same format as the group file **/etc/group**.

RETURN VALUES

getgrent( ), **getgrgid( )**, and **getgrnam( )** return a pointer to **struct group** on success. On EOF or error, they return NULL.

FILES

/etc/group

SEE ALSO

getlogin(3V), **getpwent**(3V), **group**(5), **ypserv**(8)

BUGS

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

Unlike the corresponding routines for passwords (see **getpwent**(3v)), which always search the entire file, these routines start searching from the current file location.

WARNING

The above routines use the standard I/O library, which increases the size of programs not otherwise using standard I/O more than might be expected.

NAME

gethostent, gethostbyaddr, gethostbyname, sethostent, endhostent – get network host entry

SYNOPSIS

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

struct hostent *gethostent( )

struct hostent *gethostbyname(name)
char *name;

struct hostent *gethostbyaddr(addr, len, type)
char *addr;
int len, type;

sethostent(stayopen)
int stayopen
endhostent( )
```

DESCRIPTION

gethostent, **gethostbyname**, and **gethostbyaddr( )** each return a pointer to an object with the following structure containing the broken-out fields of a line in the network host data base, **/etc/hosts**. In the case of **gethostbyaddr( )**, *addr* is a pointer to the binary format address of length *len* (not a character string).

```
struct  hostent {
        char    *h_name;        /* official name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* address type */
        int     h_length;       /* length of address */
        char    **h_addr_list;  /* list of addresses from name server */
};
```

The members of this structure are:

h_name Official name of the host.

h_aliases A zero terminated array of alternate names for the host.

h_addrtype The type of address being returned; currently always **AF_INET**.

h_length The length, in bytes, of the address.

h_addr_list A pointer to a list of network addresses for the named host. Host addresses are returned in network byte order.

gethostent( ) reads the next line of the file, opening the file if necessary.

sethostent( ) opens and rewinds the file. If the *stayopen* flag is non-zero, the host data base will not be closed after each call to **gethostent( )** (either directly, or indirectly through one of the other "gethost" calls).

endhostent( ) closes the file.

gethostbyname( ) and **gethostbyaddr( )** sequentially search from the beginning of the file until a matching host name or host address is found, or until end-of-file is encountered. Host addresses are supplied in network order.

FILES

/etc/hosts

SEE ALSO

hosts(5), **ypserv**(8)

DIAGNOSTICS

A NULL pointer is returned on end-of-file or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet address format is currently understood.

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, or if there is no entry in **/etc/utmp** for the process's 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(getuid())**.

FILES

/etc/utmp

SEE ALSO

cuserid(3v), **getpwent**(3v), **utmp**(5V)

RETURN VALUES

getlogin() returns a pointer to the login name on success. If the name is not found, it returns NULL.

BUGS

The return values point to static data whose content is overwritten by each call.

getlogin() does not work for processes running under a **pty** (for example, emacs shell buffers, or shell tools) unless the program ''fakes'' the login name in the **/etc/utmp** file.

NAME

getmntent, setmntent, addmntent, endmntent, hasmntopt – get file system descriptor file entry

SYNOPSIS

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

FILE *setmntent(filep, type)
char *filep;
char *type;

struct mntent *getmntent(filep)
FILE *filep;

int addmntent(filep, mnt)
FILE *filep;
struct mntent *mnt;

char *hasmntopt(mnt, opt)
struct mntent *mnt;
char *opt;

int endmntent(filep)
FILE *filep;
```

DESCRIPTION

These routines replace the **getfsent( )** routines for accessing the file system description file **/etc/fstab**. They are also used to access the mounted file system description file **/etc/mtab.**

setmntent( ) opens a file system description file and returns a file pointer which can then be used with **getmntent**, **addmntent**, or **endmntent**. The *type* argument is the same as in **fopen**(3V). **getmntent( )** reads the next line from *filep* and returns a pointer to an object with the following structure containing the broken-out fields of a line in the file system description file, **<mntent.h>**. On failure, **getmntent( )** returns the NULL pointer. The fields have meanings described in **fstab**(5).

```
struct mntent{
        char   *mnt_fsname;  /* name of mounted file system */
        char   *mnt_dir;     /* file system path prefix */
        char   *mnt_type;    /* MNTTYPE_* */
        char   *mnt_opts;    /* MNTOPT* */
        int    mnt_freq;     /* dump frequency, in days */
        int    mnt_passno;   /* pass number on parallel fsck */
};
```

addmntent( ) adds the **mntent** structure *mnt* to the end of the open file *filep*. **addmntent( )** returns 0 on success, 1 on failure. Note: *filep* has to be opened for writing if this is to work. **hasmntopt( )** scans the **mnt_opts** field of the **mntent** structure *mnt* for a substring that matches *opt*. It returns the address of the substring if a match is found, 0 otherwise. **endmntent( )** closes the file. It always returns 1, so should be treated as type void.

FILES

/etc/fstab
/etc/mtab

SEE ALSO

fopen(3V), **getfsent**(3), **fstab**(5)

DIAGNOSTICS

NULL pointer (0) returned on EOF or error.

BUGS

The returned **mntent** structure points to static information that is overwritten in each call.

NAME

getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent – get network entry

SYNOPSIS

```
#include <netdb.h>

struct netent *getnetent( )

struct netent *getnetbyname(name)
char *name;

struct netent *getnetbyaddr(net, type)
long net;
int type;

setnetent (stayopen)
int stayopen;

endnetent( )
```

DESCRIPTION

getnetent, **getnetbyname**, and **getnetbyaddr( )** each return a pointer to an object with the following structure containing the broken-out fields of a line in the network data base, **/etc/networks.**

```
struct  netent {
        char    *n_name;        /* official name of net */
        char    **n_aliases;    /* alias list */
        int     n_addrtype;     /* net number type */
        long    n_net;          /* net number */
};
```

The members of this structure are:

n_name	The official name of the network.
n_aliases	A zero terminated list of alternate names for the network.
n_addrtype	The type of the network number returned; currently only AF_INET.
n_net	The network number. Network numbers are returned in machine byte order.

getnetent( ) reads the next line of the file, opening the file if necessary.

setnetent( ) opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to **setnetent( )** (either directly, or indirectly through one of the other "getnet" calls).

endnetent( ) closes the file.

getnetbyname( ) and **getnetbyaddr( )** sequentially search from the beginning of the file until a matching net name or net address and type is found, or until end-of-file is encountered. Network numbers are supplied in host order.

FILES

/etc/networks

SEE ALSO

networks(5), **ypserv**(8)

DIAGNOSTICS

A NULL pointer is returned on end-of-file or error.

BUGS

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

Only Internet network numbers are currently understood.

NAME

getnetgrent, setnetgrent, endnetgrent, innetgr – get network group entry

SYNOPSIS

getnetgrent(machinep, userp, domainp)
char **machinep, **userp, **domainp;

setnetgrent(netgroup)
char *netgroup

endnetgrent( )

innetgr(netgroup, machine, user, domain)
char *netgroup, *machine, *user, *domain;

DESCRIPTION

getnetgrent( ) returns the next member of a network group. After the call, *machinep* will contain a pointer to a string containing the name of the machine part of the network group member, and similarly for *userp* and *domainp*. If any of *machinep*, *userp* or *domainp* is returned as a NULL pointer, it signifies a wild card. **getnetgrent( )** will use **malloc(3V)** to allocate space for the name. This space is released when a **endnetgrent( )** call is made. **getnetgrent( )** returns 1 if it succeeded in obtaining another member of the network group, 0 if it has reached the end of the group.

getnetgrent( ) establishes the network group from which **getnetgrent( )** will obtain members, and also restarts calls to **getnetgrent( )** from the beginning of the list. If the previous **setnetgrent( )** call was to a different network group, a **endnetgrent( )** call is implied. **endnetgrent( )** frees the space allocated during the **getnetgrent( )** calls. **innetgr** returns 1 or 0, depending on whether *netgroup* contains the machine, user, domain triple as a member. Any of the three strings *machine*, *user*, or *domain* can be NULL, in which case it signifies a wild card.

FILES

/etc/netgroup

WARNINGS

The Network Information Service (NIS) must be running when using **getnetgrent( )**, since it only inspects the NIS netgroup map, never the local files.

NOTES

The Network Information Service (NIS) was formerly known as Sun Yellow Pages (YP). The functionality of the two remains the same; only the name has changed.

NAME

getopt, optarg, optind – get option letter from argument vector

SYNOPSIS

```
int getopt(argc, argv, optstring)
int argc;
char **argv;
char *optstring;

extern char *optarg;
extern int optind, opterr;
```

DESCRIPTION

getopt( ) returns the next option letter in *argv* that matches a letter in *optstring*. *optstring* must contain the option letters the command using **getopt( )** will recognize; if a letter is followed by a colon, the option is expected to have an argument, or group of arguments, which must be separated from it by white space.

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

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

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

DIAGNOSTICS

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

EXAMPLE

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options **a** and **b**, and the option **o**, which requires an option argument:

```
main(argc, argv)
int argc;
char **argv;
{
        int c;
        extern char *optarg;
        extern int optind;
        .
        .
        .
        while ((c = getopt(argc, argv, "abo:")) != –1)
                switch (c) {
                case 'a':
                        if (bflg)
                                errflg++;
                        else
                                aflg++;
                        break;
                case 'b':
                        if (aflg)
                                errflg++;
                        else
                                bproc ();
                        break;
```

```
            case 'o':
                ofile = optarg;
                break;
            case '?':
                errflg++;
            }
        if (errflg) {
            (void)fprintf(stderr, "usage: . . . ");
            exit (2);
        }
        for (; optind < argc; optind++) {
            if (access(argv[optind], 4)) {
        .
        .
        .
    }
```

SEE ALSO

getopts(1)

WARNING

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**, or if that cannot be opened, from the standard input, after prompting with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most 8 characters. An interrupt will terminate input and send an interrupt signal to the calling program before returning.

SYSTEM V DESCRIPTION

If **/dev/tty** cannot be opened, **getpass( )** returns a NULL pointer. It does not read the standard input.

FILES

/dev/tty

SEE ALSO

crypt(3)

NOTES

The above routine uses **<stdio.h>**, which increases the size of programs not otherwise using standard I/O, more than might be expected.

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent – get protocol entry

SYNOPSIS

```
#include <netdb.h>

struct protoent *getprotoent( )

struct protoent *getprotobyname(name)
char *name;

struct protoent *getprotobynumber(proto)
int proto;

setprotoent(stayopen)
int stayopen;

endprotoent( )
```

DESCRIPTION

getprotoent, **getprotobyname**, and **getprotobynumber**( ) each return a pointer to an object with the following structure containing the broken-out fields of a line in the network protocol data base, **/etc/protocols**.

```
        struct   protoent {
                 char    *p_name;        /* official name of protocol */
                 char    **p_aliases;    /* alias list */
                 int     p_proto;        /* protocol number */
        };
```

The members of this structure are:

p_name	The official name of the protocol.
p_aliases	A zero terminated list of alternate names for the protocol.
p_proto	The protocol number.

getprotoent( ) reads the next line of the file, opening the file if necessary.

setprotoent( ) opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to **getprotoent**( ) (either directly, or indirectly through one of the other "getproto" calls).

endprotoent( ) closes the file.

getprotobyname( ) and **getprotobynumber**( ) sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until end-of-file is encountered.

FILES

/etc/protocols

SEE ALSO

protocols(5), **ypserv**(8)

DIAGNOSTICS

A NULL pointer is returned on end-of-file or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet protocols are currently understood.

NAME

getpw – get name from uid

SYNOPSIS

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

DESCRIPTION

getpw( ) is obsoleted by **getpwent**(3V).

getpw( ) searches the password file for the (numerical) *uid*, and fills in *buf* with the corresponding line; it returns non-zero if *uid* could not be found. The line is null-terminated.

FILES

/etc/passwd

SEE ALSO

getpwent(3V), **passwd**(5)

DIAGNOSTICS

Non-zero return on error.

NAME

getpwaent, getpwanam, setpwaent, endpwaent, fgetpwaent – get password adjunct file entry

SYNOPSIS

```
#include <sys/types.h>
#include <sys/label.h>
#include <sys/audit.h>
#include <pwdadj.h>

struct passwd_adjunct *getpwaent( )

struct passwd_adjunct *getpwanam(name)
char *name;

struct passwd_adjunct *fgetpwaent(f)
FILE *f;

void setpwaent( )

void endpwaent( )
```

DESCRIPTION

Both **getpwaent( )** and **getpwanam( )** return a pointer to an object with the following structure containing the broken-out fields of a line in the password adjunct file. Each line in the file contains a **passwd_adjunct** structure, declared in the **<pwdadj.h>** header file:

```
struct  passwd_adjunct {
        char          *pwa_name;
        char          *pwa_passwd;
        blabel_t       pwa_minimum;
        blabel_t       pwa_maximum;
        blabel_t       pwa_def;
        audit_state_t  pwa_au_always;
        audit_state_t  pwa_au_never;
        int           pwa_version;
};
```

When first called, **getpwaent( )** returns a pointer to a **passwd_adjunct** structure describing data from the first line in the file. Thereafter, it returns a pointer to a **passwd_adjunct** structure describing data from the next line in the file. So successive calls can be used to search the entire file.

getpwanam( ) searches from the beginning of the file until it finds a login name matching *name*, then returns a pointer to the particular structure in which it was found.

Calling **setpwaent( )** rewinds the password adjunct file to allow repeated searches. Calling **endpwaent( )** closes the password adjunct file when processing is complete.

Because read access is required on **/etc/security/passwd.adjunct, getpwaent( )** and **getpwanam( )** will fail unless the calling process has effective UID of root.

FILES

/etc/security/passwd.adjunct
/var/yp/*domainname***/passwd.adjunct.byname**

DIAGNOSTICS

A NULL pointer is returned on end-of-file or error.

SEE ALSO

getpwent(3V), **getgrent**(3V), **passwd.adjunct**(5), **ypserv**(8)

BUGS

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

NAME

getpwent, getpwuid, getpwnam, setpwent, endpwent, setpwfile, fgetpwent – get password file entry

SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwent()

struct passwd *getpwuid(uid)
uid_t uid;

struct passwd *getpwnam(name)
char *name;

void setpwent()

void endpwent()

int setpwfile(name)
char *name;

struct passwd *fgetpwent(f)
FILE *f;
```

DESCRIPTION

getpwent(), **getpwuid()** and **getpwnam()** each return a pointer to an object with the following structure containing the fields of a line in the password 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;
        uid_t   pw_uid;
        gid_t   pw_gid;
        int     pw_quota;
        char    *pw_comment;
        char    *pw_gecos;
        char    *pw_dir;
        char    *pw_shell;
};
struct passwd *getpwent(), *getpwuid(), *getpwnam();
```

The fields **pw_quota** and **pw_comment** are unused; the others have meanings described in **passwd**(5). When first called, **getpwent()** returns a pointer to the first passwd structure in the file; thereafter, it returns a pointer to the next passwd structure in the file; so successive calls can be used to search the entire file. **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.

setpwfile() changes the default password file to *name* thus allowing alternate password files to be used. Note: it does *not* close the previous file. If this is desired, **endpwent()** should be called prior to it. **setpwfile()** will fail if it is called before a call to one of **getpwent()**, **getpwuid()**, **setpwent()**, or **getpwnam()** , or if it is called before a call to one of these functions and after a call to **endpwent()**.

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

SYSTEM V DESCRIPTION

struct passwd is declared in **pwd.h** as:

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

The field **pw_age** is used to hold a value for "password aging" on some systems; "password aging" is not supported on Sun systems.

RETURN VALUES

getpwent( ), **getpwuid( )**, and **getpwnam( )** return a pointer to **struct passwd** on success. On EOF or error, or if the requested entry is not found, they return NULL.

setpwfile( ) returns:

1	on success.
0	on failure.

FILES

/etc/passwd
/var/yp/*domainname***/passwd.byname**
/var/yp/*domainname***/passwd.byuid**

SEE ALSO

getgrent(3V), **issecure**(3), **getlogin**(3V), **passwd**(5), **ypserv**(8)

NOTES

The above routines use the standard I/O library, which increases the size of programs not otherwise using standard I/O more than might be expected.

setpwfile( ) and **fgetpwent( )** are obsolete and should not be used, because when the system is running in secure mode (see **issecure**(3)), the password file only contains part of the information needed for a user database entry.

BUGS

All information is contained in a static area which is overwritten by subsequent calls to these functions, so it must be copied if it is to be saved.

NAME

getrpcent, getrpcbyname, getrpcbynumber, endrpcent, setrpcent – get RPC entry

SYNOPSIS

```
#include <netdb.h>

struct rpcent *getrpcent( )

struct rpcent *getrpcbyname(name)
char *name;

struct rpcent *getrpcbynumber(number)
int number;

setrpcent (stayopen)
int stayopen

endrpcent ( )
```

DESCRIPTION

getrpcent, **getrpcbyname**, and **getrpcbynumber( )** each return a pointer to an object with the following structure containing the broken-out fields of a line in the rpc program number data base, **/etc/rpc**.

```
struct   rpcent {
         char     *r_name;        /* name of server for this rpc program */
         char     **r_aliases;    /* alias list */
         long     r_number;       /* rpc program number */
};
```

The members of this structure are:

r_name	The name of the server for this rpc program.
r_aliases	A zero terminated list of alternate names for the rpc program.
r_number	The rpc program number for this service.

getrpcent( ) reads the next line of the file, opening the file if necessary.

setrpcent( ) opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to **getrpcent( )** (either directly, or indirectly through one of the other "getrpc" calls).

endrpcent closes the file.

getrpcbyname( ) and **getrpcbynumber( )** sequentially search from the beginning of the file until a matching rpc program name or program number is found, or until end-of-file is encountered.

FILES

/etc/rpc

SEE ALSO

rpc(5), **rpcinfo**(8C), **ypserv**(8)

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

BUGS

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

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;
FILE *stream;
```

DESCRIPTION

gets( ) reads characters from the standard input stream, **stdin**, into the array pointed to by *s*, until a NEWLINE character is read or an EOF condition is encountered. The NEWLINE character is discarded and the string is terminated with a null character. **gets( )** returns its argument.

fgets( ) reads characters from the stream into the array pointed to by *s*, until $n-1$ characters are read, a NEWLINE character is read and transferred to *s*, or an EOF condition is encountered. The string is then terminated with a null character. **fgets( )** returns its first argument.

SEE ALSO

puts(3S), **getc**(3V), **scanf**(3V), **fread**(3S), **ferror**(3V)

BUGS

If the input to **gets** ( ) or **fgets** ( ) contains a null character, the null terminates the input, and all subsequent data will be lost.

DIAGNOSTICS

If EOF is encountered and no characters have been read, no characters are transferred to *s* and a NULL pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a NULL pointer is returned. Otherwise *s* is returned.

NAME

getservent, getservbyport, getservbyname, setservent, endservent – get service entry

SYNOPSIS

```
#include <netdb.h>

struct servent *getservent( )

struct servent *getservbyname(name, proto)
char *name, *proto;

struct servent *getservbyport(port, proto)
int port; char *proto;

setservent(stayopen)
int stayopen;

endservent( )
```

DESCRIPTION

getservent, *getservbyname*, and *getservbyport* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network services data base, **/etc/services**.

```
struct  servent {
        char    *s_name;        /* official name of service */
        char    **s_aliases;    /* alias list */
        int     s_port;         /* port service resides at */
        char    *s_proto;       /* protocol to use */
};
```

The members of this structure are:

s_name	The official name of the service.
s_aliases	A zero terminated list of alternate names for the service.
s_port	The port number at which the service resides. Port numbers are returned in network short byte order.
s_proto	The name of the protocol to use when contacting the service.

getservent( ) reads the next line of the file, opening the file if necessary.

getservent( ) opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to **getservent( )** (either directly, or indirectly through one of the other "getserv" calls).

endservent( ) closes the file.

getservbyname( ) and **getservbyport( )** sequentially search from the beginning of the file until a matching protocol name or port number is found, or until end-of-file is encountered. If a protocol name is also supplied (non-NULL), searches must also match the protocol.

FILES

/etc/services

SEE ALSO

getprotoent(3N), **services**(5), **ypserv**(8)

DIAGNOSTICS

A NULL pointer is returned on end-of-file or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved. Expecting port numbers to fit in a 32 bit quantity is probably naive.

NAME

getsubopt – parse sub options from a string.

SYNOPSIS

```
int getsubopt(optionp, tokens, valuep)
char     **optionp;
char     *tokens[];
char     **valuep;
```

DESCRIPTION

getsubopt( ) is a function to parse suboptions in a flag argument that was initially parsed by **getopt(3)**. These suboptions are separated by commas and may consist of either a single token, or a token-value pair separated by an equal sign. Since commas delimit suboptions in the option string they are not allowed to be part of the suboption or the value of a suboption. An example command that uses this syntax is **mount(8)**, which allows you to specify mount parameters with the *-o* switch as follows :

```
        pepper % mount  -o  rw,hard,bg,wsize=1024  speed:/usr   /usr
```

In this example there are four suboptions: 'rw', 'hard', 'bg', and 'wsize', the last of which has an associated value of 1024.

getsubopt( ) takes the address of a pointer to the option string, a vector of possible tokens, and the address of a value string pointer. It returns the index of the token that matched the suboption in the input string or -1 if there was no match. If the option string at **optionp* contains only one subobtion, **getsubopt( )** updates **optionp* to point to the NUL at the end of the string, otherwise it isolates the suboption by replacing the comma seperator with a NUL, and updates **optionp* to point to the start of the next suboption. If the suboption has an associated value, **getsubopt( )** updates **valuep* to point to the value's first character. Otherwise it sets **valuep* to NULL.

The token vector is organized as a series of pointers to null-terminated strings. The end of the token vector is identified by a NULL pointer.

When **getsubopt( )** returns, if **valuep* is not NULL, then the suboption processed included a value. The calling program may use this information to determine if the presence or lack of a value for this subobtion is an error.

Additionally, when **getsubopt( )** fails to match the suboption with the tokens in the *tokens* array, the calling program should decide if this is an error, or if the unrecognized option should be passed on to another program.

DIAGNOSTICS

getsubopt( ) returns -1 when the token it is scanning is not in the token vector. The variable addressed by *valuep* contains a pointer to the first character of the *token* that was not recognized rather than a pointer to a value for that token.

The variable addressed by *optionp* points to the next option to be parsed, or a NUL character if there are no more options.

EXAMPLE

The following code fragment shows how you might process options to the **mount(8)** command using **getsubopt(3)**.

```
char *myopts[] = {
#define READONLY      0
                      "ro",
#define READWRITE     1
                      "rw",
#define WRITESIZE     2
                      "wsize",
#define READSIZE      3
                      "rsize",
                      NULL };
```

```
main(argc, argv)
        int  argc;
        char **argv;
{
        int sc, c, errflag;
        char *options, *value;
        extern char *optarg;
        extern int optind;
        .
        .
        .
        while((c = getopt(argc, argv, "abf:o:")) != -1) {
                switch (c) {
                case ’a’: /* process a option */
                        break;
                case ’b’: /* process b option */
                        break;
                case ’f’:
                        ofile = optarg;
                        break;
                case ’?’:
                        errflag++;
                        break;
                case ’o’:
                        options = optarg;
                        while (*options != ’\0’) {
                                switch(getsubopt(&options,myopts,&value) {
                                case READONLY : /* process ro option */
                                        break;
                                case READWRITE : /* process rw option */
                                        break;
                                  case WRITESIZE : /* process wsize option */
                                        if (value == NULL) {
                                                error_no_arg();
                                                errflag++;
                                        } else
                                                write_size = atoi(value);
                                        break;
                                case READSIZE : /* process rsize option */
                                        if (value == NULL) {
                                                error_no_arg();
                                                errflag++;
                                        } else
                                                read_size = atoi(value);
                                        break;
                                default :
                                        /* process unknown token */
                                        error_bad_token(value);
                                        errflag++;
                                        break;
                                }
                        }
                        break;
                        }
                }
                if (errflag) {
                        /* print Usage instructions etc. */
                }
                for (; optind<argc; optind++) {
                        /* process remaining arguments */
                }
                .
                .
                .
        }
```

SEE ALSO

getopt(3)

NOTES

During parsing, commas in the option input string are changed to nulls.

White space in tokens or token-value pairs must be protected from the shell by quotes.

NAME

gettext, textdomain – retrieve a message string, get and set text domain

SYNOPSIS

```
char *gettext(msgtag)
char *msgtag;

char *textdomain(domainname)
char *domainname;
```

DESCRIPTION

gettext( ) returns a pointer to a null-terminated string (target string). *msgtag* is a string used at run-time to select the target string from the current domain of the active pool of messages. The length and contents of strings returned by **gettext( )** are undetermined until called at run-time. The string returned by **gettext( )** cannot be modified by the caller, but may be overwritten by a subsequent call to **gettext( )**. The LC_MESSAGES locale category setting determines the locale of strings that **gettext( )** returns.

The calling process can dynamically change the choice of locale for strings returned by **gettext( )** by invoking the **setlocale**(3V) function with the correct category and the required locale. If **setlocale( )** is not called or is called with an invalid value, **gettext( )** defaults to the "C" locale. The default name for the current domain is the empty string.

gettext( ) first attempts to resolve the target string from the active domain and locale of the message pool. The current locale and domain are determined by the combination of both the LC_MESSAGES category of locale and the current domain setting.

If the target string cannot be found by using the current locale and domain then *msgtag* and current domain are applied to the implementation-defined default locale (this default locale could contain any language). If the default locale does not also contain the target string then the *msgtag* and current domain will be applied to the "C" locale of the message pool. If the target string still cannot be found then **gettext( )** will return *msgtag*.

Any of the following conditions will result in a message not being found in the string archive:

- Non-existent archive selected after **setlocale( )** or **textdomain( )** was called.
- Non-existent archive in the "C" environment if **setlocale( )** was not called.
- Non-existent or deleted entry in the archive.

textdomain( ) sets the current domain to *domainname*. Subsequent calls to **gettext( )** refer to this domain. If *domainname* is NULL, **textdomain( )** returns the name of the current domain without changing it.

The setting of domain made by the last successful **textdomain( )** call remains valid across any number of subsequent calls to **setlocale( )**.

RETURN VALUES

gettext( ) returns a pointer to the null-terminated target string on success. On failure, **gettext( )** returns *msgtag*.

textdomain( ) returns a pointer to the name of the current domain. If the domain has not been set prior to this call, **textdomain( )** returns a pointer to an empty string. **textdomain( )** returns NULL if:

- *domainname* contains an invalid character.
- *domainname* is longer than LINE_MAX bytes in length.
- If, at the time of the call to **textdomain( )**, the combination of current locale and *domainname* creates a domain that does not exist at run-time. Note: in this case **textdomain( )** may have been called prior to a successful **setlocale**(3V) call, but **textdomain( )** will always check against current locale setting.

EXAMPLES

The following produces 'Hit Return\n' in a locale that is invalid or is valid and contains the same target string as the key:

```
printf( gettext( "Hit Return\n" );
```

On a system whose default language is French, and whose process has the **LC_MESSAGES** category validly set, the following might print: 'Bonjour':

```
setlocale( LC_MESSAGES, "" );
textdomain( "Morning" );
printf( gettext( "Welcome" );
```

If the **LC_MESSAGES** category was invalidly set and the default (**LC_DEFAULT**) is set to English, the last example above might print 'Good morning'. If the default is not set or is also invalid, the example would print 'Welcome'.

SEE ALSO

setlocale(3V), **installtxt(8)**

NAME

getttyent, getttynam, setttyent, endttyent – get ttytab file entry

SYNOPSIS

```
#include <ttyent.h>

struct ttyent *getttyent( )

struct ttyent *getttynam(name)
char *name;

setttyent( )

endttyent( )
```

DESCRIPTION

getttyent( ) and **getttynam( )** each return a pointer to an object with the following structure containing the broken-out fields of a line from the tty description file.

```
struct  ttyent {
        char    *ty_name;       /* terminal device name */
        char    *ty_getty;      /* command to execute, usually getty */
        char    *ty_type;       /* terminal type for termcap (3X) */
        int     ty_status;      /* status flags (see below for defines) */
        char    *ty_window;     /* command to start up window manager */
        char    *ty_comment;    /* usually the location of the terminal */
};
#define TTY_ON          0x1     /* enable logins (startup getty) */
#define TTY_SECURE      0x2     /* allow root to login */
```

ty_name is the name of the character-special file in the directory **/dev**. For various reasons, it must reside in the directory **/dev**.

ty_getty is the command (usually getty(8)) which is invoked by **init** to initialize tty line characteristics. In fact, any arbitrary command can be used; a typical use is to initiate a terminal emulator in a window system.

ty_type is the name of the default terminal type connected to this tty line. This is typically a name from the **termcap**(5) data base. The environment variable TERM is initialized with this name by **getty**(8) or **login**(1).

ty_status is a mask of bit fields which indicate various actions to be allowed on this tty line. The following is a description of each flag.

TTY_ON
: Enables logins (that is, **init**(8) will start the specified "getty" command on this entry).

TTY_SECURE
: Allows root to login on this terminal. Note: TTY_ON must be included for this to be useful.

ty_window is the command to execute for a window system associated with the line. The window system will be started before the command specified in the **ty_getty** entry is executed. If none is specified, this will be NULL.

ty_comment is the trailing comment field, if any; a leading delimiter and white space will be removed.

getttyent( ) reads the next line from the **ttytab** file, opening the file if necessary; **setttyent( )** rewinds the file; **endttyent( )** closes it.

getttynam( ) searches from the beginning of the file until a matching *name* is found (or until EOF is encountered).

FILES

/etc/ttytab

SEE ALSO

login(1), **ttyslot**(3V), **gettytab**(5), **ttytab**(5), **termcap**(5), **getty**(8), **init**(8)

DIAGNOSTICS

NULL pointer (0) returned on EOF or error.

BUGS

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

NAME

getusershell, setusershell, endusershell – get legal user shells

SYNOPSIS

char *getusershell( )

setusershell( )

endusershell( )

DESCRIPTION

getusershell( ) returns a pointer to a legal user shell as defined by the system manager in the file **/etc/shells**. If **/etc/shells** does not exist, the four locations of the two standard system shells **/bin/sh**, **/bin/csh**, **/usr/bin/sh** and **/usr/bin/csh** are returned.

getusershell() reads the next line (opening the file if necessary); **setusershell()** rewinds the file; **endusershell( )** closes it.

FILES

/etc/shells
/bin/sh
/bin/csh
/usr/bin/sh
/usr/bin/csh

DIAGNOSTICS

The routine **getusershell( )** returns a NULL pointer (0) on EOF or error.

BUGS

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

NAME

getwd – get current working directory pathname

SYNOPSIS

#include <sys/param.h>

char *getwd(pathname)
char pathname[MAXPATHLEN];

DESCRIPTION

getwd( ) copies the absolute pathname of the current working directory to *pathname* and returns a pointer to the result.

DIAGNOSTICS

getwd( ) returns zero and places a message in *pathname* if an error occurs.

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.

EXAMPLE

The following example will read in strings followed by two numbers and store them in a hash table, discarding duplicates. It will then read in strings and find the matching entry in the hash table and print it 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(3), **lsearch**(3), **malloc**(3V), **string**(3), **tsearch**(3)

DIAGNOSTICS

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

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

WARNING

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

BUGS

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

NAME

inet inet_addr, inet_network, inet_makeaddr, inet_lnaof, inet_netof, inet_ntoa – Internet address manipulation

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long
inet_addr(cp)
char *cp;

inet_network(cp)
char *cp;

struct in_addr
inet_makeaddr(net, lna)
int net, lna;

inet_lnaof(in)
struct in_addr in;

inet_netof(in)
struct in_addr in;

char *
inet_ntoa(in)
struct in_addr in;
```

DESCRIPTION

The routines **inet_addr()** and **inet_network()** each interpret character strings representing numbers expressed in the Internet standard '.' notation, returning numbers suitable for use as Internet addresses and Internet network numbers, respectively. The routine **inet_makeaddr()** takes an Internet network number and a local network address and constructs an Internet address from it. The routines **inet_netof()** and **inet_lnaof()** break apart Internet host addresses, returning the network number and local network address part, respectively.

The routine **inet_ntoa()** returns a pointer to a string in the base 256 notation "d.d.d.d" described below.

All Internet address are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES

Values specified using the '.' notation take one of the following forms:

a.b.c.d
a.b.c
a.b
a

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address. Note: when an Internet address is viewed as a 32-bit integer quantity on Sun386i systems, the bytes referred to above appear as **d.c.b.a**. That is, Sun386i bytes are ordered from right to left.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right most two bytes of the network address. This makes the three part address format convenient for specifying Class B network addresses as "128.net.host".

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses as "net.host".

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as ''parts'' in a '.' notation may be decimal, octal, or hexadecimal, as specified in the C language (that is, a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

SEE ALSO

gethostent(3N), **getnetent**(3N), **hosts**(5), **networks**(5),

DIAGNOSTICS

The value −1 is returned by **inet_addr( )** and **inet_network( )** for malformed requests.

BUGS

The problem of host byte ordering versus network byte ordering is confusing. A simple way to specify Class C network addresses in a manner similar to that for Class B and Class A is needed.

The return value from **inet_ntoa( )** points to static information which is overwritten in each call.

NAME

initgroups – initialize supplementary group IDs

SYNOPSIS

```
initgroups(name, basegid)
char *name;
int basegid;
```

DESCRIPTION

initgroups() reads through the group file and sets up, using the **setgroups** call (see **getgroups**(2V)), the supplementary group IDs for the user specified in *name*. The **basegid** is automatically included in the supplementary group IDs. Typically this value is given as the group number from the password file.

FILES

/etc/group

SEE ALSO

getgroups(2V), **getgrent**(3V)

DIAGNOSTICS

initgroups() returns –1 if it was not invoked by the super-user.

BUGS

initgroups() uses the routines based on **getgrent**(3V). If the invoking program uses any of these routines, the group structure will be overwritten in the call to **initgroups**.

NAME

insque, remque – insert/remove element from a queue

SYNOPSIS

```
struct qelem {
        struct   qelem *q_forw;
        struct   qelem *q_back;
        char     q_data[ ];
};

insque(elem, pred)
struct qelem *elem, *pred;

remque(elem)
struct qelem *elem;
```

DESCRIPTION

insque() and **remque()** manipulate queues built from doubly linked lists. Each element in the queue must be in the form of ‘‘struct qelem’’. **insque()** inserts *elem* in a queue immediately after *pred*; **remque()** removes an entry *elem* from a queue.

NAME

issecure – indicates whether system is running secure

SYNOPSIS

int issecure( )

DESCRIPTION

This function tells whether the system has been configured to run in secure mode. It returns 0 if the system is not running secure, and non-zero if the system is running secure.

NAME

kvm_getu, kvm_getcmd – get the u-area or invocation arguments for a process

SYNOPSIS

```
#include <kvm.h>
#include <sys/param.h>
#include <sys/user.h>
#include <sys/proc.h>

struct user *kvm_getu(kd, proc)
kvm_t *kd;
struct proc *proc;

int kvm_getcmd(kd, proc, u, arg, env)
kvm_t *kd;
struct proc *proc;
struct user *u;
char ***arg;
char ***env;
```

DESCRIPTION

kvm_getu() reads the u-area of the process specified by *proc* to an area of static storage associated with *kd* and returns a pointer to it. Subsequent calls to **kvm_getu()** will overwrite this static area.

kd is a pointer to a kernel identifier returned by **kvm_open**(3K). *proc* is a pointer to a copy (in the current process' address space) of a *proc* structure (obtained, for instance, by a prior **kvm_nextproc**(3K) call).

kvm_getcmd() constructs a list of string pointers that represent the command arguments and environment that were used to initiate the process specified by *proc*.

kd is a pointer to a kernel identifier returned by **kvm_open**(3K). *u* is a pointer to a copy (in the current process' address space) of a *user* structure (obtained, for instance, by a prior **kvm_getu()** call). If *arg* is not NULL, then the command line arguments are formed into a null-terminated array of string pointers. The address of the first such pointer is returned in *arg*. If *env* is not NULL, then the environment is formed into a null-terminated array of string pointers. The address of the first of these is returned in *env*.

The pointers returned in *arg* and *env* refer to data allocated by **malloc**(3V) and should be freed (by a call to **free** (see **malloc**(3V)) when no longer needed. Both the string pointers and the strings themselves are deallocated when freed.

Since the environment and command line arguments may have been modified by the user process, there is no guarantee that it will be possible to reconstruct the original command at all. Thus, **kvm_getcmd()** will make the best attempt possible, returning −1 if the user process data is unrecognizable.

RETURN VALUES

On success, **kvm_getu()** returns a pointer to a copy of the u-area of the process specified by *proc*. On failure, it returns NULL.

kvm_getcmd() returns:

0	on success.
−1	on failure.

SEE ALSO

execve(2V), **kvm_nextproc**(3K), **kvm_open**(3K), **kvm_read**(3K), **malloc**(3V)

NOTES

If **kvm_getcmd()** returns −1, the caller still has the option of using the command line fragment that is stored in the u-area.

NAME

kvm_getproc, kvm_nextproc, kvm_setproc – read system process structures

SYNOPSIS

```
#include <kvm.h>
#include <sys/param.h>
#include <sys/time.h>
#include <sys/proc.h>

struct proc *kvm_getproc(kd, pid)
kvm_t *kd;
int pid;

struct proc *kvm_nextproc(kd)
kvm_t *kd;

int kvm_setproc(kd)
kvm_t *kd;
```

DESCRIPTION

kvm_nextproc() may be used to sequentially read all of the system process structures from the kernel identified by *kd* (see **kvm_open**(3K)). Each call to **kvm_nextproc()** returns a pointer to the static memory area that contains a copy of the next valid process table entry. There is no guarantee that the data will remain valid across calls to **kvm_nextproc()**, **kvm_setproc()**, or **kvm_getproc()**. Therefore, if the process structure must be saved, it should be copied to non-volatile storage.

For performance reasons, many implementations will cache a set of system process structures. Since the system state is liable to change between calls to **kvm_nextproc()**, and since the cache may contain obsolete information, there is no guarantee that *every* process structure returned refers to an active process, nor is it certain that *all* processes will be reported.

kvm_setproc() rewinds the process list, enabling **kvm_nextproc()** to rescan from the beginning of the system process table. **kvm_setproc()** will always flush the process structure cache, allowing an application to re-scan the process table of a running system.

kvm_getproc() locates the **proc** structure of the process specified by *pid* and returns a pointer to it. **kvm_getproc()** does not interact with the process table pointer manipulated by **kvm_nextproc**, however, the restrictions regarding the validity of the data still apply.

RETURN VALUES

On success, **kvm_nextproc()** returns a pointer to a copy of the next valid process table entry. On failure, it returns NULL.

On success, **kvm_getproc()** returns a pointer to the **proc** structure of the process specified by *pid*. On failure, it returns NULL.

kvm_setproc() returns:

0	on success.
–1	on failure.

SEE ALSO

kvm_getu(3K), **kvm_open**(3K), **kvm_read**(3K)

NAME

kvm_nlist – get entries from kernel symbol table

SYNOPSIS

```
#include <kvm.h>
#include <nlist.h>

int kvm_nlist(kd, nl)
kvm_t *kd;
struct nlist *nl;
```

DESCRIPTION

kvm_nlist( ) examines the symbol table from the kernel image identified by *kd* (see **kvm_open**(3K)) and selectively extracts a list of values and puts them in the array of **nlist**( ) structures pointed to by *nl*. The name list pointed to by **nl**( ) consists of an array of structures containing names, types and values. The *n_name* field of each such structure is taken to be a pointer to a character string representing a symbol name. The list is terminated by an entry with a NULL pointer (or a pointer to a null string) in the *n_name* field. For each entry in *nl*, if the named symbol is present in the kernel symbol table, its value and type are placed in the *n_value* and *n_type* fields. If a symbol cannot be located, the corresponding *n_type* field of **nl**( ) is set to zero.

RETURN VALUES

On success, **kvm_nlist**( ) returns the number of symbols that were not located in the symbol table. On failure, it returns –1 and sets all of the *n_type* fields in members of the array pointed to by **nl** to zero.

SEE ALSO

kvm_open(3K), **kvm_read**(3K), **nlist**(3V), **a.out**(5)

NAME

kvm_open, kvm_close – specify a kernel to examine

SYNOPSIS

```
#include <kvm.h>
#include <fcntl.h>

kvm_t *kvm_open(namelist, corefile, swapfile, flag, errstr)
char *namelist, *corefile, *swapfile;
int flag;
char *errstr;

int kvm_close(kd)
kvm_t *kd;
```

DESCRIPTION

kvm_open() initializes a set of file descriptors to be used in subsequent calls to kernel VM routines. It returns a pointer to a kernel identifier that must be used as the *kd* argument in subsequent kernel VM function calls.

The *namelist* argument specifies an unstripped executable file whose symbol table will be used to locate various offsets in *corefile*. If *namelist* is NULL, the symbol table of the currently running kernel is used to determine offsets in the core image. In this case, it is up to the implementation to select an appropriate way to resolve symbolic references (for instance, using **/vmunix** as a default *namelist* file).

corefile specifies a file that contains an image of physical memory, for instance, a kernel crash dump file (see **savecore**(8)) or the special device **/dev/mem**. If *corefile* is NULL, the currently running kernel is accessed (using **/dev/mem** and **/dev/kmem**).

swapfile specifies a file that represents the swap device. If both *corefile* and *swapfile* are NULL, the swap device of the "currently running kernel" is accessed. Otherwise, if *swapfile* is NULL, **kvm_open()** may succeed but subsequent **kvm_getu**(3K) function calls may fail if the desired information is swapped out.

flag is used to specify read or write access for *corefile* and may have one of the following values:

O_RDONLY	open for reading
O_RDWR	open for reading and writing

errstr is used to control error reporting. If it is a NULL pointer, no error messages will be printed. If it is non-NULL, it is assumed to be the address of a string that will be used to prefix error messages generated by **kvm_open**. Errors are printed to **stderr**. A useful value to supply for *errstr* would be **argv**[0]. This has the effect of printing the process name in front of any error messages.

kvm_close() closes all file descriptors that were associated with *kd*. These files are also closed on **exit**(2v) and **execve**(2V). **kvm_close()** also resets the **proc** pointer associated with **kvm_nextproc**(3K) and flushes any cached kernel data.

RETURN VALUES

kmv_open() returns a non-NULL value suitable for use with subsequent kernel VM function calls. On failure, it returns NULL and no files are opened.

kvm_close() returns:

0	on success.
−1	on failure.

FILES

/vmunix
/dev/kmem
/dev/mem
/dev/drum

SEE ALSO

execve(2V), **exit**(2v), **kvm_getu**(3K), **kvm_nextproc**(3K), **kvm_nlist**(3K), **kvm_read**(3K), **savecore**(8)

NAME

kvm_read, kvm_write – copy data to or from a kernel image or running system

SYNOPSIS

```
#include <kvm.h>

int kvm_read(kd, addr, buf, nbytes)
kvm_t *kd;
unsigned long addr;
char *buf;
unsigned nbytes;

int kvm_write(kd, addr, buf, nbytes)
kvm_t *kd;
unsigned long addr;
char *buf;
unsigned nbytes;
```

DESCRIPTION

kvm_read( ) transfers data from the kernel image specified by *kd* (see **kvm_open**(3K)) to the address space of the process. *nbytes* bytes of data are copied from the kernel virtual address given by *addr* to the buffer pointed to by *buf*.

kvm_write( ) is like **kvm_read**( ), except that the direction of data transfer is reversed. In order to use this function, the **kvm_open**(3K) call that returned *kd* must have specified write access. If a user virtual address is given, it is resolved in the address space of the process specified in the most recent **kvm_getu**(3K) call.

RETURN VALUES

On success, **kvm_read**( ) and **kvm_write**( ) return the number of bytes actually transferred. On failure, they return –1.

SEE ALSO

kvm_getu(3K), **kvm_nlist**(3K), **kvm_open**(3K)

NAME

l3tol, ltol3 – convert between 3-byte integers and long integers

SYNOPSIS

```
#include <stdlib.h>
void l3tol (lp, cp, n)
long *lp;
const char *cp;
int n;

void ltol3 (cp, lp, n)
char *cp;
const 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 filesystem maintenance where the block numbers are three bytes long.

SEE ALSO

fs(5)

WARNINGS

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

NAME

ldahread – read the archive header of a member of a COFF archive file

SYNOPSIS

```
#include <stdio.h>
#include <ar.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldahread (ldptr, arhead)
LDFILE *ldptr;
ARCHDR *arhead;
```

AVAILABILITY

Available only on Sun 386i systems running a SunOS 4.0*x* release or earlier. Not a SunOS 4.1 release feature.

DESCRIPTION

If **TYPE**(*ldptr*) is the archive file magic number, **ldahread** reads the archive header of the COFF file currently associated with *ldptr* into the area of memory beginning at *arhead*.

ldahread returns **SUCCESS** or **FAILURE**. **ldahread** will fail if **TYPE**(*ldptr*) does not represent an archive file, or if it cannot read the archive header.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), **ldfcn**(3), **ldopen**(3X), **intro**(5)

NAME

ldclose, ldaclose – close a COFF file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldclose (ldptr)
LDFILE *ldptr;

int ldaclose (ldptr)
LDFILE *ldptr;
```

AVAILABILITY

Available only on Sun 386i systems running a SunOS 4.0*x* release or earlier. Not a SunOS 4.1 release feature.

DESCRIPTION

ldopen(3X) and **ldclose**() are designed to provide uniform access to both simple COFF object files and COFF object files that are members of archive files. Thus an archive of COFF files can be processed as if it were a series of simple COFF files.

If TYPE(*ldptr*) does not represent an archive file, **ldclose**() will close the file and free the memory allocated to the **LDFILE** structure associated with *ldptr*. If TYPE(*ldptr*) is the magic number of an archive file, and if there are any more files in the archive, **ldclose**() will reinitialize OFFSET(*ldptr*) to the file address of the next archive member and return **FAILURE**. The **LDFILE** structure is prepared for a subsequent **ldopen**(3X). In all other cases, **ldclose**() returns SUCCESS.

ldaclose() closes the file and frees the memory allocated to the **LDFILE** structure associated with *ldptr* regardless of the value of TYPE*(ldptr)*. **ldaclose**() always returns SUCCESS. The function is often used in conjunction with *ldaopen*.

The program must be loaded with the object file access routine library **libld.a**.

intro(5) describes *INCDIR* and *LIBDIR*.

SEE ALSO

fclose(3V), **ldfcn**(3), **ldopen**(3X), **intro**(5)

NAME

ldfcn – common object file access routines

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

AVAILABILITY

Available only on Sun 386i systems running a SunOS 4.0*x* release or earlier. Not a SunOS 4.1 release feature.

DESCRIPTION

These routines are for reading COFF object files and archives containing COFF object files. Although the calling program must know the detailed structure of the parts of the object file that it processes, the routines effectively insulate the calling program from knowledge of the overall structure of the object file.

The interface between the calling program and the object file access routines is based on the defined type **LDFILE**, defined as **struct ldfile**, declared in the header file **ldfcn.h**. The primary purpose of this structure is to provide uniform access to both simple object files and to object files that are members of an archive file.

The function **ldopen**(3X) allocates and initializes the **LDFILE** structure and returns a pointer to the structure to the calling program. The fields of the **LDFILE** structure may be accessed individually through macros defined in **ldfcn.h** and contain the following information:

LDFILE *ldptr;

TYPE(ldptr) The file magic number used to distinguish between archive members and simple object files.

IOPTR(ldptr) The file pointer returned by *fopen* and used by the standard input/output functions.

OFFSET(ldptr) The file address of the beginning of the object file; the offset is non-zero if the object file is a member of an archive file.

HEADER(ldptr) The file header structure of the object file.

The object file access functions themselves may be divided into four categories:

(1) Functions that open or close an object file

ldopen(3X) and **ldaopen**( ) (see **ldopen**(3X))
open a common object file
ldclose(3X) and **ldaclose**( ) (see **ldclose**(3X))
close a common object file

(2) Functions that read header or symbol table information

ldahread(3X)
read the archive header of a member of an archive file
ldfhread(3X)
read the file header of a common object file
ldshread(3X) and **ldnshread**( ) (see **ldshread**(3X))
read a section header of a common object file
ldtbread(3X)
read a symbol table entry of a common object file
ldgetname(3X)
retrieve a symbol name from a symbol table entry or from the string table

(3) Functions that position an object file at (seek to) the start of the section, relocation, or line number information for a particular section.

ldohseek(3X)
seek to the optional file header of a common object file

ldsseek(3X) and **ldnsseek**( ) (see **ldsseek**(3X))
seek to a section of a common object file

ldrseek(3X) and **ldnrseek**( ) (see **ldrseek**(3X))
seek to the relocation information for a section of a common object file

ldlseek(3X) and **ldnlseek**( ) (see **ldlseek**(3X))
seek to the line number information for a section of a common object file

ldtbseek(3X)
seek to the symbol table of a common object file

(4) The unction **ldtbindex**(3X), which returns the index of a particular common object file symbol table entry.

These functions are described in detail on their respective manual pages.

All the functions except **ldopen**(3X), **ldgetname**(3X), **ldtbindex**(3X) return either **SUCCESS** or **FAILURE**, both constants defined in **ldfcn.h**. **ldopen**(3X) and **ldaopen**( ) (see **ldopen**(3X)) both return pointers to an **LDFILE** structure.

Additional access to an object file is provided through a set of macros defined in **ldfcn.h**. These macros parallel the standard input/output file reading and manipulating functions, translating a reference of the **LDFILE** structure into a reference to its file descriptor field.

The following macros are provided:

```
GETC(ldptr)
FGETC(ldptr)
GETW(ldptr)
UNGETC(c, ldptr)
FGETS(s, n, ldptr)
FREAD((char *) ptr, sizeof (*ptr), nitems, ldptr)
FSEEK(ldptr, offset, ptrname)
FTELL(ldptr)
REWIND(ldptr)
FEOF(ldptr)
FERROR(ldptr)
FILENO(ldptr)
SETBUF(ldptr, buf)
STROFFSET(ldptr)
```

The **STROFFSET** macro calculates the address of the string table. See the manual entries for the corresponding standard input/output library functions for details on the use of the rest of the macros.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

fseek(3S), **ldahread**(3X), **ldclose**(3X), **ldgetname**(3X), **ldfhread**(3X), **ldlread**(3X), **ldlseek**(3X), **ldohseek**(3X), **ldopen**(3X), **ldrseek**(3X), **ldlseek**(3X), **ldshread**(3X), **ldtbindex**(3X), **ldtbread**(3X), **ldtbseek**(3X), **stdio**(3V), **intro**(5)

WARNING

The macro **FSEEK** defined in the header file **ldfcn.h** translates into a call to the standard input/output function **fseek**(3S). **FSEEK** should not be used to seek from the end of an archive file since the end of an archive file may not be the same as the end of one of its object file members.

NAME

ldfhread – read the file header of a COFF file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldfhread (ldptr, filehead)
LDFILE *ldptr;
FILHDR *filehead;
```

AVAILABILITY

Available only on Sun 386i systems running a SunOS 4.0*x* release or earlier. Not a SunOS 4.1 release feature.

DESCRIPTION

ldfhread( ) reads the file header of the COFF file currently associated with *ldptr* into the area of memory beginning at *filehead*.

ldfhread( ) returns **SUCCESS** or **FAILURE**. **ldfhread( )** will fail if it cannot read the file header.

In most cases the use of **ldfhread( )** can be avoided by using the macro **HEADER(***ldptr***)** defined in **ldfcn.h** (see **ldfcn**(3)). The information in any field, *fieldname*, of the file header may be accessed using **HEADER(ldptr).fieldname.**

The program must be loaded with the object file access routine library **libld.a.**

SEE ALSO

ldclose(3X), **ldfcn**(3), **ldopen**(3X)

NAME

ldgetname – retrieve symbol name for COFF file symbol table entry

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

char *ldgetname (ldptr, symbol)
LDFILE *ldptr;
SYMENT *symbol;
```

AVAILABILITY

Available only on Sun 386i systems running a SunOS 4.0*x* release or earlier. Not a SunOS 4.1 release feature.

DESCRIPTION

ldgetname() returns a pointer to the name associated with **symbol** as a string. The string is contained in a static buffer local to **ldgetname()** that is overwritten by each call to **ldgetname()**, and therefore must be copied by the caller if the name is to be saved.

ldgetname() can be used to retrieve names from object files without any backward compatibility problems. **ldgetname()** will return NULL (defined in **stdio.h**) for an object file if the name cannot be retrieved. This situation can occur:

- if the "string table" cannot be found,
- if not enough memory can be allocated for the string table,
- if the string table appears not to be a string table (for example, if an auxiliary entry is handed to **ldgetname()** that looks like a reference to a name in a nonexistent string table), or
- if the name's offset into the string table is past the end of the string table.

Typically, **ldgetname()** will be called immediately after a successful call to **ldtbread()** to retrieve the name associated with the symbol table entry filled by **ldtbread()**.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), **ldfcn**(3), **ldopen**(3X), **ldtbread**(3X), **ldtbseek**(3X)

NAME

ldlread, ldlinit, ldlitem – manipulate line number entries of a COFF file function

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <linenum.h>
#include <ldfcn.h>

int ldlread(ldptr, fcnindx, linenum, linent)
LDFILE *ldptr;
long fcnindx;
unsigned short linenum;
LINENO *linent;

int ldlinit(ldptr, fcnindx)
LDFILE *ldptr;
long fcnindx;

int ldlitem(ldptr, linenum, linent)
LDFILE *ldptr;
unsigned short linenum;
LINENO *linent;
```

AVAILABILITY

Available only on Sun 386i systems running a SunOS 4.0*x* release or earlier. Not a SunOS 4.1 release feature.

DESCRIPTION

ldlread() searches the line number entries of the COFF file currently associated with *ldptr*. **ldlread()** begins its search with the line number entry for the beginning of a function and confines its search to the line numbers associated with a single function. The function is identified by *fcnindx*, the index of its entry in the object file symbol table. **ldlread()** reads the entry with the smallest line number equal to or greater than *linenum* into the memory beginning at *linent*.

ldlinit() and **ldlitem()** together perform exactly the same function as **ldlread()**. After an initial call to **ldlread()** or **ldlinit()**, **ldlitem()** may be used to retrieve a series of line number entries associated with a single function. **ldlinit()** simply locates the line number entries for the function identified by *fcnindx*. **ldlitem()** finds and reads the entry with the smallest line number equal to or greater than *linenum* into the memory beginning at **linent()**.

ldlread(), **ldlinit()**, and **ldlitem()** each return either **SUCCESS** or **FAILURE**. **ldlread()** will fail if there are no line number entries in the object file, if *fcnindx* does not index a function entry in the symbol table, or if it finds no line number equal to or greater than *linenum*. **ldlinit()** will fail if there are no line number entries in the object file or if *fcnindx* does not index a function entry in the symbol table. **ldlitem()** will fail if it finds no line number equal to or greater than *linenum*.

The programs must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), **ldfcn**(3), **ldopen**(3X), **ldtbindex**(3X)

NAME

ldlseek, ldnlseek – seek to line number entries of a section of a COFF file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldlseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnlseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

AVAILABILITY

Available only on Sun 386i systems running a SunOS 4.0*x* release or earlier. Not a SunOS 4.1 release feature.

DESCRIPTION

ldlseek( ) seeks to the line number entries of the section specified by *sectindx* of the COFF file currently associated with *ldptr*.

ldnlseek( ) seeks to the line number entries of the section specified by *sectname*.

ldlseek( ) and **ldnlseek**( ) return **SUCCESS** or **FAILURE**. **ldlseek**( ) will fail if *sectindx* is greater than the number of sections in the object file; **ldnlseek**( ) will fail if there is no section name corresponding with **sectname*. Either function will fail if the specified section has no line number entries or if it cannot seek to the specified line number entries.

Note that the first section has an index of **one**.

The program must be loaded with the object file access routine library **libld.a.**

SEE ALSO

ldclose(3X), **ldfcn**(3), **ldopen**(3X), **ldshread**(3X)

NAME

ldohseek – seek to the optional file header of a COFF file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldohseek (ldptr)
LDFILE *ldptr;
```

AVAILABILITY

Available only on Sun 386i systems running a SunOS 4.0*x* release or earlier. Not a SunOS 4.1 release feature.

DESCRIPTION

ldohseek( ) seeks to the optional file header of the COFF file currently associated with *ldptr*.

ldohsee( ) returns SUCCESS or FAILURE. **ldohseek**( ) will fail if the object file has no optional header or if it cannot seek to the optional header.

The program must be loaded with the object file access routine library **libld**.a.

SEE ALSO

ldclose(3X), **ldfcn**(3), **ldopen**(3X), **ldfhread**(3X)

NAME

ldopen, ldaopen – open a COFF file for reading

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

LDFILE *ldopen (filename, ldptr)
char *filename;
LDFILE *ldptr;

LDFILE *ldaopen (filename, oldptr)
char *filename;
LDFILE *oldptr;
```

AVAILABILITY

Available only on Sun 386i systems running a SunOS 4.0*.x* release or earlier. Not a SunOS 4.1 release feature.

DESCRIPTION

ldopen() and **ldclose(3X)** are designed to provide uniform access to both simple object files and object files that are members of archive files. Thus an archive of COFF files can be processed as if it were a series of simple COFF files.

If *ldptr* has the value NULL, then **ldopen()** will open *filename* and allocate and initialize the **LDFILE** structure, and return a pointer to the structure to the calling program.

If *ldptr* is valid and if **TYPE(***ldptr***)** is the archive magic number, **ldopen()** will reinitialize the **LDFILE** structure for the next archive member of *filename*.

ldopen() and **ldclose(3X)** are designed to work in concert. *ldclose* will return **FAILURE** only when **TYPE(***ldptr***)** is the archive magic number and there is another file in the archive to be processed. Only then should **ldopen()** be called with the current value of *ldptr*. In all other cases, in particular whenever a new *filename* is opened, **ldopen()** should be called with a NULL *ldptr* argument.

The following is a prototype for the use of **ldopen()** and **ldclose(3X)**.

```
/* for each filename to be processed */

ldptr = NULL;
do
{
        if ( (ldptr = ldopen(filename, ldptr)) != NULL )
        {
                /* check magic number */
                /* process the file */
        }
} while (ldclose(ldptr) == FAILURE );
```

If the value of *oldptr* is not NULL, **ldaopen()** will open *filename* anew and allocate and initialize a new **LDFILE** structure, copying the **TYPE**, **OFFSET**, and **HEADER** fields from *oldptr*. **ldaopen()** returns a pointer to the new **LDFILE** structure. This new pointer is independent of the old pointer, *oldptr*. The two pointers may be used concurrently to read separate parts of the object file. For example, one pointer may be used to step sequentially through the relocation information, while the other is used to read indexed symbol table entries.

Both **ldopen**( ) and **ldaopen**( ) open *filename* for reading. Both functions return NULL if *filename* cannot be opened, or if memory for the **LDFILE** structure cannot be allocated. A successful open does not insure that the given file is a COFF file or an archived object file.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

fopen(3V), **ldclose**(3X), **ldfcn**(3)

NAME

ldrseek, ldnrseek – seek to relocation entries of a section of a COFF file

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldrseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnrseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;

AVAILABILITY

Available only on Sun 386i systems running a SunOS 4.0*.x* release or earlier. Not a SunOS 4.1 release feature.

DESCRIPTION

ldrseek() seeks to the relocation entries of the section specified by *sectindx* of the COFF file currently associated with *ldptr*.

ldnrseek() seeks to the relocation entries of the section specified by *sectname*.

ldrseek() and **ldnrseek()** return **SUCCESS** or **FAILURE.** **ldrseek()** will fail if *sectindx* is greater than the number of sections in the object file; **ldnrseek()** will fail if there is no section name corresponding with *sectname*. Either function will fail if the specified section has no relocation entries or if it cannot seek to the specified relocation entries.

Note: the first section has an index of **one**.

The program must be loaded with the object file access routine library **libld.a.**

SEE ALSO

ldclose(3X), **ldfcn**(3), **ldopen**(3X), **ldshread**(3X)

NAME

ldshread, ldnshread – read an indexed/named section header of a COFF file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <scnhdr.h>
#include <ldfcn.h>

int ldshread (ldptr, sectindx, secthead)
LDFILE *ldptr;
unsigned short sectindx;
SCNHDR *secthead;

int ldnshread (ldptr, sectname, secthead)
LDFILE *ldptr;
char *sectname;
SCNHDR *secthead;
```

AVAILABILITY

Available only on Sun 386i systems running a SunOS 4.0*x* release or earlier. Not a SunOS 4.1 release feature.

DESCRIPTION

ldshread() reads the section header specified by *sectindx* of the COFF file currently associated with *ldptr* into the area of memory beginning at *secthead*.

ldnshread() reads the section header specified by *sectname* into the area of memory beginning at *secthead*.

ldshread() and **ldnshread()** return **SUCCESS** or **FAILURE. ldshread()** will fail if *sectindx* is greater than the number of sections in the object file; **ldnshread()** will fail if there is no section name corresponding with *sectname*. Either function will fail if it cannot read the specified section header.

Note: the first section header has an index of *one*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), **ldfcn**(3), **ldopen**(3X)

NAME

ldsseek, ldnsseek – seek to an indexed/named section of a COFF file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldsseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnsseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

AVAILABILITY

Available only on Sun 386i systems running a SunOS 4.0.*x* release or earlier. Not a SunOS 4.1 release feature.

DESCRIPTION

ldsseek() seeks to the section specified by *sectindx* of the COFF file currently associated with *ldptr*.

ldnsseek() seeks to the section specified by *sectname*.

ldsseek() and **ldnsseek()** return **SUCCESS** or **FAILURE. ldsseek()** will fail if *sectindx* is greater than the number of sections in the object file; **ldnsseek()** will fail if there is no section name corresponding with *sectname*. Either function will fail if there is no section data for the specified section or if it cannot seek to the specified section.

Note: the first section has an index of *one*.

The program must be loaded with the object file access routine library **libld.a.**

SEE ALSO

ldclose(3X), ldfcn(3), ldopen(3X), ldshread(3X)

NAME

ldtbindex – compute the index of a symbol table entry of a COFF file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

long ldtbindex (ldptr)
LDFILE *ldptr;
```

AVAILABILITY

Available only on Sun 386i systems running a SunOS 4.0*.x* release or earlier. Not a SunOS 4.1 release feature.

DESCRIPTION

ldtbindex() returns the (**long**) index of the symbol table entry at the current position of the COFF file associated with *ldptr*.

The index returned by **ldtbindex**() may be used in subsequent calls to **ldtbread**(3X). However, since **ldtbindex** () returns the index of the symbol table entry that begins at the current position of the object file, if **ldtbindex**() is called immediately after a particular symbol table entry has been read, it will return the index of the next entry.

ldtbindex() will fail if there are no symbols in the object file, or if the object file is not positioned at the beginning of a symbol table entry.

Note that the first symbol in the symbol table has an index of *zero*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), **ldfcn**(3), **ldopen**(3X), **ldtbread**(3X), **ldtbseek**(3X)

NAME

ldtbread – read an indexed symbol table entry of a COFF file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldtbread (ldptr, symindex, symbol)
LDFILE *ldptr;
long symindex;
SYMENT *symbol;
```

AVAILABILITY

Available only on Sun 386i systems running a SunOS 4.0*.x* release or earlier. Not a SunOS 4.1 release feature.

DESCRIPTION

ldtbread() reads the symbol table entry specified by *symindex* of the COFF file currently associated with *ldptr* into the area of memory beginning at **symbol**.

ldtbread() returns **SUCCESS** or **FAILURE**. **ldtbread**() will fail if *symindex* is greater than or equal to the number of symbols in the object file, or if it cannot read the specified symbol table entry.

Note: the first symbol in the symbol table has an index of *zero*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), **ldfcn**(3), **ldopen**(3X), **ldtbseek**(3X), **ldgetname**(3X)

NAME

ldtbseek – seek to the symbol table of a COFF file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldtbseek (ldptr)
LDFILE *ldptr;
```

AVAILABILITY

Available only on Sun 386i systems running a SunOS 4.0*x* release or earlier. Not a SunOS 4.1 release feature.

DESCRIPTION

ldtbseek( ) seeks to the symbol table of the COFF file currently associated with *ldptr*.

ldtbseek( ) returns SUCCESS or **FAILURE**. **ldtbseek**( ) will fail if the symbol table has been stripped from the object file, or if it cannot seek to the symbol table.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), **ldfcn**(3), **ldopen**(3X), **ldtbread**(3X)

NAME

localdtconv – get date and time formatting conventions

SYNOPSIS

#include <locale.h>

struct dtconv *localdtconv( )

DESCRIPTION

localdtconv( ) returns a pointer to a structure of type **struct dtconv** containing values appropriate for the formatting of dates and times according to the rules of the current locale.

The members include the following:

char *abbrev_month_names[12]
The abbreviated names of the months; for example, the abbreviated name for January is **abbrev_month_names[0]** and the abbreviated name for December is **abbrev_month_names[11]**.

char *month_names[12]
The full names of the months; for example, the full name for January is **month_names[0]** and the full name for December is **month_names[11]**.

char *abbrev_weekday_names[7]
The abbreviated names of the weekdays; for example, the abbreviated name for Sunday is **abbrev_weekday_names[0]** and the abbreviated name for Saturday is **abbrev_weekday_names[6]**.

char *weekday_names[7]
The full names of the weekdays; for example, the full name for Sunday is **weekday_names[0]** and the full name for Saturday is **weekday_names[6]**.

char *time_format
The standard format for times, using the format specifiers supported by **strftime**( ) and **strptime**( ) (see **ctime**(3V)).

char *sdate_format
The standard short format for dates, using the format specifiers supported by **ctime** (3V).

char *dtime_format
The standard short format for dates and times together, using the format specifiers supported by **ctime**(3V).

char *am_string
The string representing AM.

char *pm_string
The string representing PM.

char *ldate_format
The standard long format for dates, using the format specifiers supported by **ctime**(3V).

The values for the members in the C locale are:

abbrev_month_names	**Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec**
month_names	**January, February, March, April, May, June, July, August, September, October, November, December**
abbrev_weekday_names	**Sun, Mon, Tue, Wed, Thu, Fri, Sat**
weekday_names	**Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday**
time_format	**%H:%M:%S**

sdate_format	**%m/%d/%y**
dtime_format	**%a %b %e %T %Z %Y**
am_string	**AM**
pm_string	**PM**
ldate_format	**%A, %B %e, %Y**

FILES

/usr/share/lib/locale/LC_TIME
standard locale information directory for category **LC_TIME**

SEE ALSO

ctime(3V), **setlocale**(3V)

NAME

localeconv – get numeric and monetary formatting conventions

SYNOPSIS

```
#include <limits.h>
#include <locale.h>

struct lconv *localeconv()
```

DESCRIPTION

localeconv() returns a pointer to a structure of type **struct lconv** containing values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

The members of the structure with type (**char** *) are strings; if a string has the value **""**, the value is not available in the current locale or has zero length. The members with type **char** are nonnegative numbers; if any of them have the value **CHAR_MAX** the value is not available in the current locale. The **lconv** structure is defined in **<locale.h>** as follows:

```
struct lconv {
        char    *decimal_point;       /* decimal point character */
        char    *thousands_sep;       /* thousands separator character */
        char    *grouping;            /* grouping of digits */
        char    *int_curr_symbol;     /* international currency symbol */
        char    *currency_symbol;     /* local currency symbol */
        char    *mon_decimal_point;   /* monetary decimal point character */
        char    *mon_thousands_sep;   /* monetary thousands separator */
        char    *mon_grouping;        /* monetary grouping of digits */
        char    *positive_sign;       /* monetary credit symbol */
        char    *negative_sign;       /* monetary debit symbol */
        char    int_frac_digits;      /* intl monetary number of fractional digits */
        char    frac_digits;          /* monetary number of fractional digits */
        char    p_cs_precedes;        /* true if currency symbol precedes credit */
        char    p_sep_by_space;       /* true if space separates c.s. from credit */
        char    n_cs_precedes;        /* true if currency symbol precedes debit */
        char    n_sep_by_space;       /* true if space separates c.s. from debit */
        char    p_sign_posn;          /* position of sign for credit */
        char    n_sign_posn;          /* position of sign for debit */
};
```

The fields of this structure represent:

decimal_point
The decimal-point character used to format non-monetary quantities.

thousands_sep
The character used to separate groups of digits to the left of the decimal-point character in formatted non-monetary quantities.

grouping
A string whose elements indicate the size of each group of digits in formatted non-monetary quantities.

int_curr_symbol
The international currency symbol applicable to the current locale, left-justified within a four-character SPACE-padded field. The character sequences are those specified in: *ISO 4217 Codes for the Representation of Currency and Funds*.

currency_symbol
The local currency symbol applicable to the current locale.

mon_decimal_point
The decimal-point used to format monetary quantities.

mon_thousands_sep
The character used to separate groups of digits to the left of the decimal-point character in formatted monetary quantities.

mon_grouping
A string whose elements indicate the size of each group of digits in formatted monetary quantities.

positive_sign
The string used to indicate a nonnegative-valued formatted monetary quantity.

negative_sign
The string used to indicate a negative-valued formatted monetary quantity.

int_frac_digits
The number of fractional digits (those after the decimal-point) to be displayed in an internationally formatted monetary quantity.

frac_digits
The number of fractional digits (those to the right of the decimal-point) to be displayed in a formatted monetary quantity.

p_cs_precedes
1 if the **currency_symbol** precedes the value for a nonnegative formatted monetary quantity; 0 if the **currency_symbol** succeeds the value for a nonnegative formatted monetary quantity.

p_sep_by_space
1 if the **currency_symbol** is separated by a SPACE from the value for a nonnegative formatted monetary quantity; 0 if the **currency_symbol** is not separated by a SPACE from the value for a nonnegative formatted monetary quantity.

n_cs_precedes
1 if the **currency_symbol** precedes the value for a negative formatted monetary quantity; 0 if the **currency_symbol** succeeds the value for a negative formatted monetary quantity.

n_sep_by_space
1 if the **currency_symbol** is separated by a SPACE from the value for a negative formatted monetary quantity; 0 if the **currency_symbol** is not separated by a SPACE from the value for a negative formatted monetary quantity.

p_sign_posn
A value indicating the positioning of the **positive_sign** for a nonnegative formatted monetary quantity.

n_sign_posn
A value indicating the positioning of the **negative_sign** for a negative formatted monetary quantity.

The elements of **grouping** and **mon_grouping** are interpreted as follows:

CHAR_MAX No further grouping is to be performed.

0 The previous element is to be repeatedly used for the remainder of the digits.

other The value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits to the left of the current group.

The values of **p_sign_posn** and **n_sign_posn** are interpreted as follows:

0 Parentheses surround the quantity and **currency_symbol.**

1 The sign string precedes the quantity and **currency_symbol.**

2 The sign string succeeds the quantity and **currency_symbol**.

3 The sign string immediately precedes the **currency_symbol**.

4 The sign string immediately succeds the **currency_symbol**.

The values for the members in the C locale are:

field	*value*
decimal_point	"."
thousands_sep	""
grouping	""
int_curr_symbol	""
currency_symbol	""
mon_decimal_point	""
mon_thousands_sep	""
mon_grouping	""
positive_sign	""
negative_sign	""
int_frac_digits	CHAR_MAX
frac_digits	CHAR_MAX
p_cs_precedes	CHAR_MAX
p_sep_by_space	CHAR_MAX
n_cs_precedes	CHAR_MAX
n_sep_by_space	CHAR_MAX
p_sign_posn	CHAR_MAX
n_sign_posn	CHAR_MAX

RETURN VALUES

localeconv( ) returns a pointer to **struct lconv** (see NOTES).

FILES

/usr/share/lib/locale/LC_MONETARY
standard locale information directory for category **LC_MONETARY**

/usr/share/lib/locale/LC_NUMERIC
standard locale information directory for category **LC_NUMERIC**

SEE ALSO

printf(3V), **scanf**(3V), **setlocale**(3V)

NOTES

localeconv( ) does not modify the **struct lconv** to which it returns a pointer, but subsequent calls to **setlocale**(3V) with categories **LC_ALL**, **LC_MONETARY**, or **LC_NUMERIC** may overwrite the contents of the structure.

NAME

lockf – record locking on files

SYNOPSIS

```
#include <unistd.h>

int lockf(fd, cmd, size)
int fd, cmd;
long size;
```

DESCRIPTION

lockf( ) places, removes, and tests for exclusive locks on sections of files. These locks are either advisory or mandatory depending on the mode bits of the file. The lock is mandatory if the set-GID bit (**S_ISGID**) is set and the group execute bit (**S_IXGRP**) is clear (see **stat**(2V) for information about mode bits). Otherwise, the lock is advisory.

If a process holds a mandatory exclusive lock on a segment of a file, both read and write operations block until the lock is removed (see **WARNINGS**).

An advisory lock does not affect read and write access to the locked segment. Advisory locks may be used by cooperating processes checking for locks using **F_GETLCK** and voluntarily observing the indicated read and write restrictions.

A locking call on an already locked file section fails, returning an error value or putting the call to sleep until that file section is unlocked. All the locks on a process are removed when that process terminates. See **fcntl**(2V) for more information about record locking.

fd is an open file descriptor. It must have **O_WRONLY** or **O_RDWR** permission for a successful locking call.

cmd is a control value which specifies the action to be taken. The accepted values for *cmd* 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 (non-blocking) */
#define F_TEST     3    /* Test section for other process' locks */
```

F_TEST returns –1 and sets **errno** to **EACCES** if a lock by another process already exists on the specified section. Otherwise, it returns 0. **F_LOCK** and **F_TLOCK** lock available file sections. **F_ULOCK** removes locks from file sections.

All other values of *cmd* are reserved for future applications and, until implemented, return an error.

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 *size* bytes if *size* is positive, and extends backward *size* bytes (the preceding bytes up to but not including the current offset) if *size* is negative. If *size* is zero, the section from the current offset through the largest file offset is locked (that is, from the current offset through the present or any future EOF). An area need not be allocated to the file to be locked, such a lock may exist after the EOF.

Sections locked with **F_LOCK** or **F_TLOCK** may contain all or part of an already locked section. They may also be partially or completely contained by an already locked section. Where these overlapping or adjacent locked sections occur, they are combined into a single section. If the table of active locks is full, a lock request requiring an additional table entry fails and an error value is returned.

F_LOCK and **F_TLOCK** differ only in their response to requests for unavailable resources. If a section is already locked, **F_LOCK** directs the calling process to sleep until the resource is available, **F_TLOCK** directs the function to return –1 and set **errno** to **EACCES** (see **ERRORS**).

When a **F_ULOCK** request releases part of a section with overlapping locks, the remaining section or sections retain the lock. If **F_ULOCK** removes the center of a locked section, the two separate locked sections remain, but an additional element is required in the table of active locks. If this table is full, **errno** is set to ENOLCK and the requested section is not released.

The danger of a deadlock exists when a process controlling a locked resource is put to sleep by requesting an unavailable resource. To avoid this danger, **lockf()** and **fcntl()** scan for this conflict before putting a locked resource to sleep. If a deadlock would result, an error value is returned.

The sleep process can be interrupted with any signal. **alarm**(3V) may be used to provide a timeout facility where needed.

RETURN VALUES

lockf( ) returns:

0	on success.
−1	on failure and sets **errno** to indicate the error.

ERRORS

EACCES	*cmd* is **F_TLOCK** or **F_TEST** and the section is already locked by another process. Note: In future, **lockf**( ) may generate EAGAIN under these conditions, so applications testing for EACCES should also test for EAGAIN.
EBADF	*fd* is not a valid open descriptor. *cmd* is **F_LOCK** or **F_TLOCK** and the process does not have write permission on the file.
EDEADLK	*cmd* is **F_LOCK** and a deadlock would occur.
EINTR	*cmd* is **F_LOCK** and a signal interrupted the process while it was waiting to complete the lock.
ENOLCK	*cmd* is **F_LOCK**, **F_TLOCK**, or **F_ULOCK** and there are no more file lock entries available.

SEE ALSO

chmod(2V), **fcntl**(2V), **flock**(2), **fork**(2V), **alarm**(3V), **lockd**(8C)

WARNINGS

Mandatory record locks are dangerous. If a runaway or otherwise out-of-control process should hold a mandatory lock on a file critical to the system and fail to release that lock, the entire system could hang or crash. For this reason, mandatory record locks may be removed in a future SunOS release.n Use advisory record locking whenever possible.

NOTES

A child process does not inherit locks from its parent on **fork**(2V).

BUGS

lockf() locks do not interact in any way with locks granted by **flock**(), but are compatible with locks granted by **fcntl**( ).

NAME

lsearch, lfind – linear search and update

SYNOPSIS

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

char *lsearch (key, base, nelp, width, compar)
char *key;
char *base;
unsigned int *nelp;
unsigned int width;
int (*compar)( );

char *lfind (key, base, nelp, width, compar)
char *key;
char *base;
unsigned int *nelp;
unsigned int width;
int (*compar)( );
```

DESCRIPTION

lsearch( ) is a linear search routine generalized from Knuth (6.1) Algorithm S. It returns a pointer into a table indicating where a datum may be found. If the datum does not occur, it is added at the end of the table. *key* points to the datum to be sought in the table. *base* points to the first element in the table. *nelp* points to an integer containing the current number of elements in the table. The integer is incremented if the datum is added to the table. *compar* is the name of the comparison function which the user must supply (**strcmp( )**, for example). It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal and non-zero otherwise.

lfind( ) is the same as **lsearch( )** except that if the datum is not found, it is not added to the table. Instead, a NULL pointer is returned.

NOTES

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

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

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

EXAMPLE

This fragment will read in ≤ TABSIZE strings of length ≤ ELSIZE and store them in a table, eliminating duplicates.

```
#include <stdio.h>
#include <search.h>
#define
TABSIZE 50
#define
ELSIZE 120
        char line[ELSIZE], tab[TABSIZE][ELSIZE], *lsearch( );
        unsigned nel = 0;
        int strcmp( );
        . . .
        while (fgets(line,
        ELSIZE, stdin) != NULL &&
            nel < TABSIZE)
                  (void) lsearch(line, (char *)tab, &nel, ELSIZE, strcmp);
         . . .
```

SEE ALSO

bsearch(3), **hsearch**(3), **tsearch**(3)

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

madvise – provide advice to VM system

SYNOPSIS

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

int madvise(addr, len, advice)
caddr_t addr;
size_t len;
int advice;
```

DESCRIPTION

madvise( ) advises the kernel that a region of user mapped memory in the range [*addr*, *addr* + *len*) will be accessed following a type of pattern. The kernel uses this information to optimize the procedure for manipulating and maintaining the resources associated with the specified mapping range.

Values for *advice* are defined in **<sys/mman.h>** as:

```
#define MADV_NORMAL      0x0        /* No further special treatment */
#define MADV_RANDOM      0x1        /* Expect random page references */
#define MADV_SEQUENTIAL             0x2/* Expect sequential page references */
#define MADV_WILLNEED    0x3        /* Will need these pages */
#define MADV_DONTNEED    0x4        /* Don't need these pages */
```

MADV_NORMAL

The default system characteristic where accessing memory within the address range causes the system to read data from the mapped file. The kernel reads all data from files into pages which are retained for a period of time as a "cache". System pages can be a scarce resource, so the kernel steals pages from other mappings when needed. This is a likely occurrence but only adversely affects system performance if a large amount of memory is accessed.

MADV_RANDOM

Tells the kernel to read in a minimum amount of data from a mapped file when doing any single particular access. Normally when an address of a mapped file is accessed, the system tries to read in as much data from the file as reasonable, in anticipation of other accesses within a certain locality.

MADV_SEQUENTIAL

Tells the system that addresses in this range are likely to only be accessed once, so the system will free the resources used to map the address range as quickly as possible. This is used in the **cat**(1V) and **cp**(1) utilities.

MADV_WILLNEED

Tells the system that a certain address range is definitely needed, so the kernel will read the specified range into memory immediately. This might be beneficial to programs who want to minimize the time it takes to access memory the first time since the kernel would need to read in from the file.

MADV_DONTNEED

Tells the kernel that the specified address range is no longer needed, so the system immediately frees the resources associated with the address range.

madvise( ) should be used by programs that have specific knowledge of their access patterns over a memory object (for example, a mapped file) and wish to increase system performance.

RETURN VALUES

madvise( ) returns:

0 on success.

–1 on failure and sets **errno** to indicate the error.

ERRORS

EINVAL *addr* is not a multiple of the page size as returned by **getpagesize**(2).

The length of the specified address range is less than or equal to 0.

advice was invalid.

EIO An I/O error occurred while reading from or writing to the file system.

ENOMEM Addresses in the range [*addr*, *addr* + *len*) are outside the valid range for the address space of a process, or specify one or more pages that are not mapped.

SEE ALSO

mctl(2), **mmap**(2)

NAME

malloc, free, realloc, calloc, cfree, memalign, valloc, mallocmap, mallopt, mallinfo, malloc_debug, malloc_verify, alloca – memory allocator

SYNOPSIS

```
#include <malloc.h>

char *malloc(size)
unsigned size;

int free(ptr)
char *ptr;

char *realloc(ptr, size)
char *ptr;
unsigned size;

char *calloc(nelem, elsize)
unsigned nelem, elsize;

int cfree(ptr)
char *ptr;

char *memalign(alignment, size)
unsigned alignment;
unsigned size;

char *valloc(size)
unsigned size;

void mallocmap( )

int mallopt(cmd, value)
int cmd, value;

struct mallinfo mallinfo( )

#include <alloca.h>

char *alloca(size)
int size;
```

SYSTEM V SYNOPSIS

```
#include <malloc.h>

void *malloc(size)
size_t size;

void free(ptr)
void *ptr;

void *realloc(ptr, size)
void *ptr;
size_t size;

void *calloc(nelem, elsize)
size_t nelem;
size_t elsize;

void *memalign(alignment, size)
size_t alignment;
size_t size;

void *valloc(size)
size_t size;
```

The XPG2 versions of the functions listed in this section are declared as they are in **SYNOPSIS** above, except **free()**, which is declared as:

void free(ptr)
char *ptr;

DESCRIPTION

These routines provide a general-purpose memory allocation package. They maintain a table of free blocks for efficient allocation and coalescing of free storage. When there is no suitable space already free, the allocation routines call **sbrk()** (see **brk(2)**) to get more memory from the system.

Each of the allocation routines returns a pointer to space suitably aligned for storage of any type of object. Each returns a NULL pointer if the request cannot be completed (see **DIAGNOSTICS**).

malloc() returns a pointer to a block of at least *size* bytes, which is appropriately aligned.

free() releases a previously allocated block. Its argument is a pointer to a block previously allocated by **malloc()**, **calloc()**, **realloc()**, **malloc()**, or **memalign()**.

realloc() changes the size of the block referenced by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If unable to honor a reallocation request, **realloc()** leaves its first argument unaltered. For backwards compatibility, **realloc()** accepts a pointer to a block freed since the most recent call to **malloc()**, **calloc()**, **realloc()**, **valloc()**, or **memalign()**. Note: using **realloc()** with a block freed *before* the most recent call to **malloc()**, **calloc()**, **realloc()**, **valloc()**, or **memalign()** is an error.

calloc() uses **malloc()** to allocate space for an array of *nelem* elements of size *elsize*, initializes the space to zeros, and returns a pointer to the initialized block. The block can be freed with **free()** or **cfree()**.

memalign() allocates *size* bytes on a specified alignment boundary, and returns a pointer to the allocated block. The value of the returned address is guaranteed to be an even multiple of *alignment*. Note: the value of *alignment* must be a power of two, and must be greater than or equal to the size of a word.

valloc(*size*) is equivalent to **memalign(getpagesize(), *size*)**.

mallocmap() prints a map of the heap to the standard output. **mallocmap()** prints each block's address, size (in bytes) and status (free or busy). A block must have a size that is no larger than the current extent of the heap.

mallopt() allows quick allocation of small blocks of memory. **mallopt()** tells subsequent calls to **malloc()** to allocate *holding blocks* containing small blocks. Under this small block algorithm, a request to **malloc()** for a small block of memory returns a pointer to one of the pre-allocated small blocks. Different holding blocks are created as needed for different sizes of small blocks.

cmd may be one of the following values, defined in **<malloc.h>**:

M_MXFAST Set the maximum size of blocks to be allocated using the small block algorithm (*maxfast*) to *value*. The algorithm allocates all blocks smaller than *maxfast* in large groups and then doles them out very quickly. Initially, *maxfast* is 0 and the small block algorithm is disabled.

M_NLBLKS Set the number of small blocks in a holding block (*numlblks*) to *value*. The holding blocks each contain *numlblks* blocks. *numlblks* must be greater than 1. The default value for *numlblks* is 100.

M_GRAIN Set the granularity for small block requests (*grain*) to *value*. The sizes of all blocks smaller than *maxfast* are rounded up to the nearest multiple of *grain*. *grain* must be greater than 0. The default value of *grain* is the smallest number of bytes which will allow alignment of any data type. When *grain* is set, *value* is rounded up to a multiple of this default.

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.

mallopt() may be called repeatedly, but may not be called after the first small block is allocated.

mallinfo() can be used during program development to determine the best settings for the parameters set by **mallopt()**. Do not call **mallinfo()** until after a call to **malloc()**. **mallinfo()** provides information describing space usage. It returns a **mallinfo** structure, defined in **<malloc.h>** as:

```
struct mallinfo {
        int arena;          /* total space in arena */
        int ordblks;        /* number of ordinary blocks */
        int smblks;         /* number of small blocks */
        int hblks;          /* number of holding blocks */
        int hblkhd;         /* space in holding block headers */
        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;       /* cost of enabling keep option */

        int mxfast;         /* max size of small blocks */
        int nlblks;         /* number of small blocks in a holding block */
        int grain;          /* small block rounding factor */
        int uordbytes;      /* space (including overhead) allocated in ord. blks */
        int allocated;      /* number of ordinary blocks allocated */
        int treeoverhead;           /* bytes used in maintaining the free tree */
};
```

alloca() allocates *size* bytes of space in the stack frame of the caller, and returns a pointer to the allocated block. This temporary space is automatically freed when the caller returns. Note that if the allocated block is beyond the current stack limit, the resulting behavior is undefined.

malloc(), **realloc()**, **memalign()** and **valloc()** return a non-NULL pointer if *size* is 0, and **calloc()** returns a non-NULL pointer if *nelem* or *elsize* is 0, but these pointers should *not* be dereferenced.

Note: Always cast the value returned by **malloc()**, **realloc()**, **calloc()**, **memalign()**, **valloc()** or **alloca()**.

SYSTEM V DESCRIPTION

The XPG2 versions of **malloc()**, **realloc()**, **memalign()** and **valloc()** return NULL if *size* is 0. The XPG2 version of **calloc()** returns NULL if *nelem* or *elsize* is 0.

RETURN VALUES

On success, **malloc()**, **calloc()**, **realloc()**, **memalign()**, **valloc()** and **alloca()** return a pointer to space suitably aligned for storage of any type of object. On failure, they return NULL.

free() and **cfree()** return:

1 on success.

0 on failure and set **errno** to indicate the error.

mallopt() returns 0 on success. If **mallopt()** is called after the allocation of a small block, or if *cmd* or *value* is invalid, it returns a non-zero value.

mallinfo() returns a **struct mallinfo**.

SYSTEM V RETURN VALUES

If *size* is 0, the XPG2 versions of **malloc()**, **realloc()**, **memalign()** and **valloc()** return NULL.

If *nelem* or *elsize* is 0, the XPG2 version of **calloc()** returns NULL.

free() does not return a value.

ERRORS

malloc(), **calloc()**, **realloc()**, **valloc()**, **memalign()**, **cfree()**, and **free()** will each fail if one or more of the following are true:

EINVAL — An invalid argument was specified.

The value of *ptr* passed to **free()**, **cfree()**, or **realloc()** was not a pointer to a block previously allocated by **malloc()**, **calloc()**, **realloc()**, **valloc()**, or **memalign()**.

The allocation heap is found to have been corrupted. More detailed information may be obtained by enabling range checks using **malloc_debug()**.

ENOMEM — *size* bytes of memory could not be allocated.

FILES

/usr/lib/debug/malloc.o	diagnostic versions of **malloc()** routines.
/usr/lib/debug/mallocmap.o	routines to print a map of the heap.

SEE ALSO

csh(1), **ld**(1), **brk**(2), **getrlimit**(2), **sigvec**(2), **sigstack**(2)

Stephenson, C.J., *Fast Fits*, in *Proceedings of the ACM 9th Symposium on Operating Systems*, SIGOPS *Operating Systems Review*, vol. 17, no. 5, October 1983.
Core Wars, in *Scientific American*, May 1984.

DIAGNOSTICS

More detailed diagnostics can be made available to programs using **malloc()**, **calloc()**, **realloc()**, **valloc()**, **memalign()**, **cfree()**, and **free()**, by including a special relocatable object file at link time (see FILES). This file also provides routines for control of error handling and diagnosis, as defined below. Note: these routines are *not* defined in the standard library.

```
int malloc_debug(level)
int level;

int malloc_verify()
```

malloc_debug() sets the level of error diagnosis and reporting during subsequent calls to **malloc()**, **calloc()**, **realloc()**, **valloc()**, **memalign()**, **cfree()**, and **free()**. The value of *level* is interpreted as follows:

Level 0 — **malloc()**, **calloc()**, **realloc()**, **valloc()**, **memalign()**, **cfree()**, and **free()** behave the same as in the standard library.

Level 1 — The routines abort with a message to the standard error if errors are detected in arguments or in the heap. If a bad block is encountered, its address and size are included in the message.

Level 2 — Same as level 1, except that the entire heap is examined on every call to the above routines.

malloc_debug() returns the previous error diagnostic level. The default level is 1.

malloc_verify() attempts to determine if the heap has been corrupted. It scans all blocks in the heap (both free and allocated) looking for strange addresses or absurd sizes, and also checks for inconsistencies in the free space table. **malloc_verify()** returns 1 if all checks pass without error, and otherwise returns 0. The checks can take a significant amount of time, so it should not be used indiscriminately.

WARNINGS

alloca() is machine-, compiler-, and most of all, system-dependent. Its use is strongly discouraged. See **getrlimit**(2), **sigvec**(2), **sigstack**(2), **csh**(1), and **ld**(1).

NOTES

Because **malloc()**, **realloc()**, **memalign()** and **valloc()** return a non-NULL pointer if *size* is 0, and **calloc()** returns a non-NULL pointer if *nelem* or *elsize* is 0, a zero size need not be treated as a special case if it should be passed to these functions unpredictably. Also, the pointer returned by these functions may be passed to subsequent invocations of **realloc()**.

SYSTEM V NOTES

The XPG2 versions of the allocation routines return NULL when passed a zero size (see **SYSTEM V DESCRIPTION** above).

BUGS

Since **realloc()** accepts a pointer to a block freed since the last call to **malloc()**, **calloc()**, **realloc()**, **valloc()**, or **memalign()**, a degradation of performance results. The semantics of **free()** should be changed so that the contents of a previously freed block are undefined.

NAME

mblen, mbstowcs, mbtowc, wcstombs, wctomb – multibyte character handling

SYNOPSIS

```
#include <stdlib.h>

int mblen(s, n)
char *s;
size_t n;

size_t mbstowcs(s, pwcs, n)
char *s;
wchar_t *pwcs;
size_t n;

int mbtowc(pwc, s, n)
wchar_t *pwc;
char *s;
size_t n;

int wcstombs(s, pwcs, n)
char *s;
wchar_t *pwcs;
size_t n;

int wctomb(s, wchar)
char *s;
wchar_t wcar;
```

DESCRIPTION

The behavior of these functions is affected by the **LC_CTYPE** category of the program's locale. For a stat-dependent encoding, each function is placed into its initial state by a call for which its character pointer argument, *s*, is a NULL pointer. Subsequent calls with *s* as other than a NULL pointer cause the internal stste of the function to be altered as necessary. A call with a *s* as a NULL pointer causes these functions to return a nonzero value if encodings have state dependency, and zero otherwise. After the **LC_CTYPE** category is changed, the shift state of these functions is indeterminate.

If *s* is not a NULL pointer, these functions work as follows:

mblen( )

Determines the number of bytes comprising the multibyte character pointed to by *s*.

mbstowcs( )

Converts a sequence of multibyte characters that begins in the initial shift state from the array pointed to by *s* into a sequence of corresponding codes and stores no more than *n* codes into the array pointed to by *pwcs*. No multibyte characters that follow a null character (which is converted into a code with value zero) will be examined or converted. Each multibyte character is converted as if by a call to **mbtowc( )**, except that the shift state of **mbtowc( )** is not affected.

No more than *n* elements will be modified in the array pointed to by *pwcs*. If copying takes place between objects that overlap, the behavior is undefined.

mbtowc( )

Determines the number of bytes that comprise the multibyte character pointed to by *s*. **mbtowc( )** then determines the code for value of type **wchar_t** that corresponds to that multibyte character. The value of the code corresponding to the null caharacter is zero. If the multibyte character is valid and *pwc* is not a null pointer, **mbtowc( )** stores the code in the object pointed to by *pwc*. At most *n* bytes of the array pointed to by *s* will be examined.

wcstowcs( )

Converts a sequence of codes that correspond to multibyte characters from the array pointed to by *pwcs* into a sequence of multibyte characters that begins in the initial shift state and stores these multibyte characters into the array pointed to by *s*, stopping if a multibyte character would exceed the limit of *n* total bytes or if a null character is stored. Each code is converted as if by a call to **wctomb( )**, except that the shift state of **wctomb( )** is not affected.

wctomb( )

Determines the number of bytes needed to represent the multibyte character corresponding to the code whose value is *wchar* (including any change in shift state). **wctomb()** stores the multibyte character representation in the array object pointed to by *s* (if *s* is not a null pointer). At most, **MB_CUR_MAX** characters are stored. If the value of *wchar* is zero, **wctomb( )** is left in the initial shift state.

RETURN VALUES

If *s* is a null pointer, **mblen( )**, **mbtowc( ),** and **wctomb( )** return a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state dependent encodings.

If *s* is not a null pointer, **mblen( )** and **mbtowc( )** either return 0 (if *s* points to the null character), or return the number of bytes that comprise the converted multibyte character (if the next *n* or fewer bytes form a valid multibyte character), or return –1 (if they do not form a valid multibyte character).

In no case will the value returned by **mbtowc( )** be greater than *n* or the value of the **MB_CUR_MAX** macro. If *s* is not a null pointer, **wctomb( )** returns –1 (if the value does not correspond to a valid multibyte character), or returns the number of bytes that comprise the multibyte character corresponding to *wchar*.

If an invalid multibyte character is encountered, **mbstowcs( )** and **wcstombs( )** return **(size_t)** –1. Otherwise, they return the number of bytes modified, not including a terminating null character, if any.

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*.

NOTES

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

BUGS

memcmp( ) uses native character comparison, which is signed on some machines and unsigned on other machines. Thus the sign of the value returned when one of the characters has its high-order bit set is implementation-dependent.

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

NAME

mktemp, mkstemp – make a unique file name

SYNOPSIS

```
char *mktemp(template)
char *template;

mkstemp(template)
char *template;
```

DESCRIPTION

mktemp( ) creates a unique file name, typically in a temporary filesystem, by replacing *template* with a unique file name, and returns the address of *template*. The string in *template* should contain a file name with six trailing Xs; **mktemp**( ) replaces the Xs with a letter and the current process ID. The letter will be chosen so that the resulting name does not duplicate an existing file. **mkstemp**( ) makes the same replacement to the template but returns a file descriptor for the template file open for reading and writing. **mkstemp**( ) avoids the race between testing whether the file exists and opening it for use.

Notes:

- **mktemp**( ) and **mkstemp**( ) actually *change* the template string which you pass; this means that you cannot use the same template string more than once — you need a fresh template for every unique file you want to open.
- When **mktemp**( ) or **mkstemp**( ) are creating a new unique filename they check for the prior existence of a file with that name. This means that if you are creating more than one unique filename, it is bad practice to use the same root template for multiple invocations of **mktemp**( ) or **mkstemp**( ).

SEE ALSO

getpid(2V), **open**(2V), **tmpfile**(3S), **tmpnam**(3S)

DIAGNOSTICS

mkstemp( ) returns an open file descriptor upon success. It returns –1 if no suitable file could be created.

mktemp( ) assigns the null string to *template* when it cannot create a unique name.

BUGS

It is possible to run out of letters.

NAME

mlock, munlock – lock (or unlock) pages in memory

SYNOPSIS

```
#include <sys/types.h>
int mlock(addr, len) caddr_t addr; size_t len;

int munlock(addr, len)
caddr_t addr;
size_t len;
```

DESCRIPTION

mlock( ) uses the mappings established for the address range [*addr, addr* + *len*) to identify memory object pages to be locked in memory. If the page identified by a mapping changes, such as occurs when a copy of a writable **MAP_PRIVATE** page is made upon the first store, the lock will be transferred to the newly copied private page.

munlock( ) removes locks established with **mlock**( ).

A given page may be locked multiple times by executing an **mlock**( ) through different mappings. That is, if two different processes lock the same page then the page will remain locked until both processes remove their locks. However, within a given mapping, page locks do not nest – multiple **mlock**( ) operations on the same address in the same process will all be removed with a single **munlock**( ). Of course, a page locked in one process and mapped in another (or visible through a different mapping in the locking process) is still locked in memory. This fact can be used to create applications that do nothing other than lock important data in memory, thereby avoiding page I/O faults on references from other processes in the system.

If the mapping through which an **mlock**( ) has been performed is removed, an **munlock**( ) is implicitly performed. An **munlock**( ) is also performed implicitly when a page is deleted through file removal or truncation.

Locks established with **mlock**( ) are not inherited by a child process after a **fork**(2V).

Due to the impact on system resources, the use of **mlock**( ) and **munlock**( ) is restricted to the super-user. Attempts to **mlock**( ) more memory than a system-specific limit will fail.

RETURN VALUES

mlock( ) and **munlock**( ) return:

0	on success.
–1	on failure and set **errno** to indicate the error.

ERRORS

EAGAIN	(**mlock**( ) only.) Some or all of the memory identified by the range [*addr, addr* + *len*) could not be locked due to insufficient system resources.
EINVAL	*addr* is not a multiple of the page size as returned by **getpagesize**(2).
ENOMEM	Addresses in the range [*addr, addr* + *len*) are invalid for the address space of a process, or specify one or more pages which are not mapped.
EPERM	The process's effective user ID is not super-user.

SEE ALSO

fork(2V), **mctl**(2), **mlockall**(3), **mmap**(2), **munmap**(2)

NAME

mlockall, munlockall – lock (or unlock) address space

SYNOPSIS

#include <sys/mman.h>

int mlockall(flags)
int flags;

int munlockall()

DESCRIPTION

mlockall() locks all pages mapped by an address space in memory. The value of *flags* determines whether the pages to be locked are simply those currently mapped by the address space, those that will be mapped in the future, or both. *flags* is built from the options defined in <sys/**mman.h**> as:

```
#define MCL_CURRENT     0x1        /* lock current mappings */
#define MCL_FUTURE      0x2        /* lock future mappings */
```

If MCL_FUTURE is specified to **mlockall()** , then as mappings are added to the address space (or existing mappings are replaced) they will also be locked, provided sufficient memory is available.

Mappings locked via **mlockall()** with any option may be explicitly unlocked with a **munlock()** call.

munlockall() removes address space locks and locks on mappings in the address space.

All conditions and constraints on the use of locked memory as exist for **mlock()** apply to **mlockall()** .

RETURN VALUES

mlockall() and **munlockall()** return:

0 on success.

–1 on failure and set **errno** to indicate the error.

ERRORS

EAGAIN (**mlockall()** only.) Some or all of the memory in the address space could not be locked due to sufficient resources.

EINVAL *flags* contains values other than MCL_CURRENT and MCL_FUTURE.

EPERM The process's effective user ID is not super-user.

SEE ALSO

mctl(2), **mlock**(3), **mmap**(2)

NAME

monitor, monstartup, moncontrol – prepare execution profile

SYNOPSIS

```
#include <a.out.h>

monitor(lowpc, highpc, buffer, bufsize, nfunc)
int (*lowpc)( ), (*highpc)( );
short buffer[ ];

monstartup(lowpc, highpc)
int (*lowpc)( ), (*highpc)( );

moncontrol(mode)
```

DESCRIPTION

There are two different forms of monitoring available. An executable program created by '**cc –p**' automatically includes calls for the **prof**(1) monitor, and includes an initial call with default parameters to its start-up routine **monstartup**. In this case, **monitor**( ) need not be called explicitly, except to gain fine control over **profil**(2) buffer allocation. An executable program created by '**cc –pg**' automatically includes calls for the **gprof**(1) monitor.

monstartup( ) is a high-level interface to **profil**(2). *lowpc* and *highpc* specify the address range that is to be sampled; the lowest address sampled is that of *lowpc* and the highest is just below *highpc*. **monstartup**( ) allocates space using **sbrk** (see **brk**(2)) and passes it to **monitor**( ) (as described below) to record a histogram of program-counter values, and calls to certain functions. Only calls to functions compiled with '**cc –p**' are recorded.

On Sun-2, Sun-3, and Sun-4 systems, an entire program can be profiled with:

```
extern etext( );
. . .
monstartup(N_TXTOFF(0), etext);
```

On Sun386i systems, the equivalent code sequence is:

```
extern etext( );
extern _start( );
. . .
monstartup(_start, etext);
```

etext lies just above all the program text, see **end**(3).

To stop execution monitoring and post results to the file **mon.out**, use:

```
monitor(0);
```

prof(1) can then be used to examine the results.

moncontrol( ) is used to selectively control profiling within a program. This works with both **prof**(1) and **gprof**(1). Profiling begins when the program starts. To stop the collection of profiling statistics, use:

```
moncontrol(0)
```

To resume the collection of statistics, use:

```
moncontrol(1)
```

This allows you to measure the cost of particular functions. Note: an output file is be produced upon program exit, regardless of the state of **moncontrol**.

monitor( ) is a low level interface to **profil**(2). *lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a (user supplied) array of *bufsize* short integers. At most *nfunc* call counts can be kept.

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. **monitor( )** divides the buffer into space to record the histogram of program counter samples over the range *lowpc* to *highpc*, and space to record call counts of functions compiled with the **cc −p**.

To profile the entire program on Sun-2, Sun-3, and Sun-4 systems using the low-level interface to profil(2), it is sufficient to use

```
extern etext( );
...
monitor(N_TXTOFF(0), etext, buf, bufsize, nfunc);
```

On Sun386i systems, the equivalent calls are:

```
extern etext( );
extern _start( );
...
monitor(_start, etext, buf, bufsize, nfunc);
```

FILES

mon.out

SEE ALSO

cc(1V), **prof**(1), **gprof**(1), **brk**(2), **profil**(2), **end**(3)

NAME

mp, madd, msub, mult, mdiv, mcmp, min, mout, pow, gcd, rpow, itom, xtom, mtox, mfree – multiple precision integer arithmetic

SYNOPSIS

```
#include <mp.h>

madd(a, b, c)
MINT *a, *b, *c;

msub(a, b, c)
MINT *a, *b, *c;

mult(a, b, c)
MINT *a, *b, *c;

mdiv(a, b, q, r)
MINT *a, *b, *q, *r;

mcmp(a,b)
MINT *a, *b;

min(a)
MINT *a;

mout(a)
MINT *a;

pow(a, b, c, d)
MINT *a, *b, *c, *d;

gcd(a, b, c)
MINT *a, *b, *c;

rpow(a, n, b)
MINT *a, *b;
short n;

msqrt(a, b, r)
MINT *a, *b, *r;

sdiv(a, n, q, r)
MINT *a, *q;
short n, *r;

MINT *itom(n)
short n;

MINT *xtom(s)
char *s;

char *mtox(a)
MINT *a;

void mfree(a)
MINT *a;
```

DESCRIPTION

These routines perform arithmetic on integers of arbitrary length. The integers are stored using the defined type MINT. Pointers to a MINT should be initialized using the function **itom()**, which sets the initial value to *n*. Alternatively, **xtom()** may be used to initialize a MINT from a string of hexadecimal digits. **mfree()** may be used to release the storage allocated by the **itom()** and **xtom()** routines.

madd(), **msub**() and **mult**() assign to their third arguments the sum, difference, and product, respectively, of their first two arguments. **mdiv**() assigns the quotient and remainder, respectively, to its third and fourth arguments. **sdiv**() is like **mdiv**() except that the divisor is an ordinary integer. **msqrt** produces the square root and remainder of its first argument. **mcmp**() compares the values of its arguments and returns 0 if the two values are equal, a value greater than 0 if the first argument is greater than the second, and a value less than 0 if the second argument is greater than the first. **rpow** raises *a* to the *n*th power and assigns this value to *b*. **pow**() raises *a* to the *b*th power, reduces the result modulo *c* and assigns this value to *d*. **min**() and **mout**() do decimal input and output. gcd() finds the greatest common divisor of the first two arguments, returning it in the third argument. **mtox**() provides the inverse of **xtom**(). To release the storage allocated by **mtox**(), use **free**() (see **malloc**(3V)).

Use the **–lmp** loader option to obtain access to these functions.

DIAGNOSTICS

Illegal operations and running out of memory produce messages and core images.

FILES

/usr/lib/libmp.a

SEE ALSO

malloc(3V)

NAME

msync – synchronize memory with physical storage

SYNOPSIS

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

int msync(addr, len, flags)
caddr_t addr; size_t len; int flags;
```

DESCRIPTION

msync( ) writes all modified copies of pages over the range [*addr*, *addr* + *len*) to their permanent storage locations. **msync**() optionally invalidates any copies so that further references to the pages will be obtained by the system from their permanent storage locations.

Values for *flags* are defined in **<sys/mman.h>** as:

```
#define MS_ASYNC        0x1     /* Return immediately */
#define MS_INVALIDATE   0x2     /* Invalidate mappings */
```

and are used to control the behavior of **msync**( ). One or more flags may be specified in a single call.

MS_ASYNC returns immediately once all I/O operations are scheduled; normally, **msync**( ) will not return until all I/O operations are complete. MS_INVALIDATE invalidates all cached copies of data from memory objects, requiring them to be re-obtained from the object's permanent storage location upon the next reference.

msync( ) should be used by programs that require a memory object to be in a known state, for example in building transaction facilities.

RETURN VALUES

msync( ) returns:

0 on success.

–1 on failure and sets **errno** to indicate the error.

ERRORS

EINVAL *addr* is not a multiple of the page size as returned by **getpagesize**(2).

flags is not some combination of MS_ASYNC or MS_INVALIDATE.

EIO An I/O error occurred while reading from or writing to the file system.

ENOMEM Addresses in the range [*addr*, *addr* + *len*) are outside the valid range for the address space of a process, or specify one or more pages that are not mapped.

EPERM MS_INVALIDATE was specified and one or more of the pages is locked in memory.

SEE ALSO

mctl(2), **mmap**(2)

NAME

ndbm, dbm_open, dbm_close, dbm_fetch, dbm_store, dbm_delete, dbm_firstkey, dbm_nextkey, dbm_error, dbm_clearerr – data base subroutines

SYNOPSIS

```
#include <ndbm.h>

typedef struct {
char *dptr;
int dsize;
} datum;

DBM *dbm_open(file, flags, mode)
char *file;
int flags, mode;

void dbm_close (db)
DBM *db;

datum dbm_fetch(db, key)
DBM *db;
datum key;

int dbm_store(db, key, content, flags)
DBM *db;
datum key, content;
int flags;

int dbm_delete(db, key)
DBM *db;
datum key;

datum dbm_firstkey(db)
DBM *db;

datum dbm_nextkey(db)
DBM *db;

int dbm_error(db)
DBM *db;

int dbm_clearerr(db)
DBM *db;
```

DESCRIPTION

These functions maintain key/content pairs in a data base. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses. This package replaces the earlier **dbm**(3X) library, which managed only a single database.

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

Before a database can be accessed, it must be opened by **dbm_open**. This will open and/or create the files *file*.**dir** and *file*.**pag** depending on the flags parameter (see **open**(2V)).

A database is closed by calling **dbm_close**.

Once open, the data stored under a key is accessed by **dbm_fetch( )** and data is placed under a key by **dbm_store**. The *flags* field can be either **DBM_INSERT** or **DBM_REPLACE**. **DBM_INSERT** will only insert new entries into the database and will not change an existing entry with the same key. **DBM_REPLACE** will replace an existing entry if it has the same key. A key (and its associated contents) is deleted by **dbm_delete**. A linear pass through all keys in a database may be made, in an (apparently) random order, by use of **dbm_firstkey()** and **dbm_nextkey**. **dbm_firstkey()** will return the first key in the database. **dbm_nextkey()** will return the next key in the database. This code will traverse the data base:

```
for (key = dbm_firstkey(db); key.dptr != NULL; key = dbm_nextkey(db))
```

dbm_error() returns non-zero when an error has occurred reading or writing the database. **dbm_clearerr()** resets the error condition on the named database.

SEE ALSO

ar(1V), **cat**(1V), **cp**(1), **tar**(1), **open**(2V), **dbm**(3X)

DIAGNOSTICS

All functions that return an **int** indicate errors with negative values. A zero return indicates no error. Routines that return a **datum** indicate errors with a NULL (0) *dptr*. If **dbm_store** called with a *flags* value of **DBM_INSERT** finds an existing entry with the same key it returns 1.

BUGS

The **.pag** file will contain holes so that its apparent size is about four times its actual content. Older versions of the UNIX operating system may create real file blocks for these holes when touched. These files cannot be copied by normal means (**cp**(1), **cat**(1V), **tar**(1), **ar**(1V)) without filling in the holes.

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

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 4096 bytes). Moreover all key/content pairs that hash together must fit on a single block. **dbm_store()** will return an error in the event that a disk block fills with inseparable data.

dbm_delete() does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by **dbm_firstkey()** and **dbm_nextkey()** depends on a hashing function, not on anything interesting.

There are no interlocks and no reliable cache flushing; thus concurrent updating and reading is risky.

NAME

nice – change nice value of a process

SYNOPSIS

int nice(incr)

DESCRIPTION

The nice value of the process is changed by *incr*. Positive nice values get less service than normal. See **nice**(1) for a discussion of the relationship of nice value and scheduling priority.

A nice value of 10 is recommended to users who wish to execute long-running programs without undue impact on system performance.

Negative increments are illegal, except when specified by the super-user. The nice value is limited to the range –20 (most urgent) to 19 (least). Requests for values above or below these limits result in the nice value being set to the corresponding limit.

The nice value of a process is passed to a child process by **fork**(2V). For a privileged process to return to normal nice value from an unknown state, **nice**( ) should be called successively with arguments –40 (goes to nice value –20 because of truncation), 20 (to get to 0), then 0 (to maintain compatibility with previous versions of this call).

SYSTEM V DESCRIPTION

The maximum allowed value for *incr* is 40 (least urgent).

RETURN VALUES

nice( ) returns:

0	on success.
–1	on failure and sets **errno** to indicate the error.

SYSTEM V RETURN VALUES

nice( ) returns the new nice value on success. On failure, it returns –1 and sets **errno** to indicate the error.

ERRORS

The nice value is not changed if:

EACCES	The value of *incr* specified was negative, and the effective user ID is not super-user.

SYSTEM V ERRORS

The nice value is not changed if:

EPERM	The value of *incr* specified was negative, or greater than 40, and the effective user ID is not super-user.

SEE ALSO

nice(1), **fork**(2V), **getpriority**(2), **pstat**(8), **renice**(8)

NAME

nl_langinfo – language information

SYNOPSIS

```
#include <nl_types.h>
#include <langinfo.h>

char *nl_langinfo(item)
nl_item item;
```

DESCRIPTION

nl_langinfo( ) returns a pointer to a null-terminated string containing information relevant to a particular language or cultural area defined in the program's locale. The manifest constant names and values of *item* are defined in **<langinfo.h>** . For example:

```
nl_langinfo(ABDAY_1);
```

would return a pointer to the string 'Dom' if the identified language was Portuguese, and 'Sun' if the identified language was English.

RETURN VALUES

In a locale where *langinfo* data is not defined, **nl_langinfo**() returns a pointer to the corresponding string in the "C" locale. In all locales **nl_langinfo**( ) returns a pointer to an empty string if *item* contains an invalid setting.

SEE ALSO

setlocale(3V), **environ**(5V)

NAME

nlist – get entries from symbol table

SYNOPSIS

```
#include <nlist.h>

int nlist(filename, nl)
char *filename;
struct nlist *nl;
```

DESCRIPTION

nlist() examines the symbol table from the executable image 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 pointed to by *nl* consists of an array of structures containing names, types and values. The *n_name* field of each such structure is taken to be a pointer to a character string representing a symbol name. The list is terminated by an entry with a NULL pointer (or a pointer to a null string) in the *n_name* field. For each entry in *nl*, if the named symbol is present in the executable image's symbol table, its value and type are placed in the *n_value* and *n_type* fields. If a symbol cannot be located, the corresponding *n_type* field of *nl* is set to zero.

RETURN VALUES

On success, **nlist()** returns the number of symbols that were not located in the symbol table. On failure, it returns –1 and sets all of the *n_type* fields in members of the array pointed to by *nl* to zero.

SYSTEM V RETURN VALUES

nlist() returns 0 on success.

SEE ALSO

a.out(5), **coff**(5)

NOTES

On Sun-2, Sun-3, and Sun-4 systems, type entries are set to 0 if the file cannot be read or if it does not contain a valid name list.

On Sun386i systems, the type entries may be zero even when the name list succeeded, but the value entries will be zero only when the file cannot be read or does not contain a valid name list. Therefore, on Sun386i systems, the value entry can be used to determine whether the command succeeded.

NAME

on_exit – name termination handler

SYNOPSIS

```
int on_exit(procp, arg)
void (*procp)();
caddr_t arg;
```

DESCRIPTION

on_exit() names a routine to be called after a program calls **exit**(3) or returns normally, and before its process terminates. The routine named is called as

(*procp)(status, arg);

where *status* is the argument with which **exit()** was called, or zero if *main* returns. Typically, *arg* is the address of an argument vector to (**procp*), but may be an integer value. Several calls may be made to **on_exit**, specifying several termination handlers. The order in which they are called is the reverse of that in which they were given to **on_exit**.

SEE ALSO

gprof(1), **tcov**(1), **exit**(3)

DIAGNOSTICS

on_exit() returns zero normally, or nonzero if the procedure name could not be stored.

NOTES

This call is specific to the SunOS operating system and should not be used if portability is a concern.

Standard I/O exit processing is always done last.

NAME

pause – stop until signal

SYNOPSIS

int pause( )

DESCRIPTION

pause() never returns normally. It is used to give up control while waiting for a signal from **kill**(2V) or an interval timer, see **getitimer**(2). Upon termination of a signal handler started during a pause, **pause**( ) will return.

RETURN VALUES

When it returns, **pause**( ) returns –1.

ERRORS

When it returns, **pause**( ) sets **errno** to:

EINTR A signal is caught by the calling process and control is returned from the signal-catching function.

SEE ALSO

kill(2V), **getitimer**(2), **select**(2), **sigpause**(2V)

NAME

perror, errno – system error messages

SYNOPSIS

```
void perror(s)
char *s;

#include <errno.h>

int sys_nerr;
char *sys_errlist[ ];
int errno;
```

DESCRIPTION

perror( ) produces a short error message on the standard error describing the last error encountered during a call to a system or library function. If *s* is not a NULL pointer and does not point to a null string, the string it points to is printed, followed by a colon, followed by a space, followed by the message and a NEWLINE. If *s* is a NULL pointer or points to a null string, just the message is printed, followed by a NEWLINE. 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** (see **intro**(2)), which is set when errors occur but not cleared when non-erroneous calls are made.

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

SEE ALSO

intro(2), **psignal**(3)

NAME

plock – lock process, text, or data segment 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

RETURN VALUES

plock( ) returns:

0	on success.
–1	on failure and sets **errno** to indicate the error.

ERRORS

EAGAIN Not enough memory.

EINVAL *op* is equal to **PROCLOCK** and a process lock, a text lock, or a data lock already exists on the calling process.

op is equal to **TXTLOCK** and a text lock, or a process lock already exists on the calling process.

op is equal to **DATLOCK** and a data lock, or a process lock already exists on the calling process.

op is equal to **UNLOCK** and no type of lock exists on the calling process.

EPERM The effective user ID of the calling process is not super-user.

SEE ALSO

execve(2V), **exit**(2V), **fork**(2V)

NAME

plot, openpl, erase, label, line, circle, arc, move, cont, point, linemod, space, closepl – graphics interface

SYNOPSIS

```
openpl( )

erase( )

label(s)
char s[ ];

line(x1, y1, x2, y2)

circle(x, y, r)

arc(x, y, x0, y0, x1, y1)

move(x, y)

cont(x, y)

point(x, y)

linemod(s)
char s[ ];

space(x0, y0, x1, y1)

closepl( )
```

AVAILABILITY

These routines are available with the *Graphics* software installation option. Refer to *Installing SunOS 4.1* for information on how to install optional software.

DESCRIPTION

LP These subroutines generate graphic output in a relatively device-independent manner. See **plot**(5) for a description of their effect. **openpl( )** must be used before any of the others to open the device for writing. **closepl( )** flushes the output.

String arguments to **label( )** and **linemod( )** are null-terminated and do not contain NEWLINE characters.

Various flavors of these functions exist for different output devices. They are obtained by the following **ld**(1) options:

–lplot	device-independent graphics stream on standard output for **plot**(1G) filters
–l300	GSI 300 terminal
–l300s	GSI 300S terminal
–l450	GSI 450 terminal
–l4014	Tektronix 4014 terminal
–lplotaed	AED 512 color graphics terminal
–lplotbg	BBN bitgraph graphics terminal
–lplotdumb	Dumb terminals without cursor addressing or line printers
–lplotgigi	DEC Gigi terminals
–lplot2648	Hewlett Packard 2648 graphics terminal
–lplot7221	Hewlett Packard 7221 graphics terminal
–lplotimagen	Imagen laser printer (default 240 dots-per-inch resolution).

FILES

/usr/lib/libplot.a
/usr/lib/lib300.a
/usr/lib/lib300s.a
/usr/lib/lib450.a
/usr/lib/lib4014.a
/usr/lib/libplotaed.a
/usr/lib/libplotbg.a
/usr/lib/libplotdumb.a
/usr/lib/libplotgigi.a
/usr/lib/libplot2648.a
/usr/lib/libplot7221.a
/usr/lib/libplotimagen.a

SEE ALSO

graph(1G), **ld**(1), **plot**(1G), **plot**(5)

NAME

popen, pclose – open or close a pipe (for I/O) from or to a process

SYNOPSIS

```
#include <stdio.h>

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

pclose(stream)
FILE *stream;
```

DESCRIPTION

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

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

Because open files are shared, a type **r** command may be used as an input filter, reading its standard input (which is also the standard output of the process doing the **popen()**) and providing filtered input on the stream, and a type **w** command may be used as an output filter, reading a stream of output written to the stream process doing the **popen()** and further filtering it and writing it to its standard output (which is also the standard input of the process doing the **popen()**).

popen() always calls **sh**(1), never **csh**(1).

SEE ALSO

csh(1), **sh**(1), **pipe**(2V), **wait**(2V), **fclose**(3V), **fopen**(3V), **system**(3)

DIAGNOSTICS

popen() returns a NULL pointer if the pipe or process cannot be created, or if it cannot allocate as much memory as it needs.

pclose() returns –1 if stream is not associated with a '**popened**' command.

BUGS

If the original and '**popened**' processes concurrently read or write a common file, neither should use buffered I/O, because the buffering gets all mixed up. Similar problems with an output filter may be forestalled by careful buffer flushing, for instance, with **fflush()**; see **fclose**(3V).

NAME

pmap_getmaps, pmap_getport, pmap_rmtcall, pmap_set, pmap_unset, xdr_pamp, xdr_pmaplist – library routines for RPC bind service

DESCRIPTION

These routines allow client C programs to make procedure calls to the RPC binder service. **portmap**(1) maintains a list of mappings between programs and their universal addresses.

Routines

#include <rpc/rpc.h>

struct pmaplist * pmap_getmaps(addr)
struct sockaddr_in *addr;

Return a list of the current RPC program-to-address mappings on the host located at IP address **addr*. This routine returns NULL if the remote portmap service could not be contacted. The command '**rpcinfo –p**' uses this routine (see **rpcinfo**(8C)).

u_short pmap_getport(addr, prognum, versnum, protocol)
struct sockaddr_in *addr;
u_long prognum, versnum, protocol;

Return the port number on which waits a service that supports program number *prognum*, version *versnum*, and speaks the transport protocol *protocol*. The address is returned in *addr*, which should be preallocated. The value of *protocol* can be either **IPPROTO_UDP** or **IPPROTO_TCP**. A return value of zero means that the mapping does not exist or that the RPC system failed to contact the remote **portmap** service. In the latter case, the global variable **rpc_createer** (see **rpc_clnt_create**(3N)) contains the RPC status. If the requested version number is not registered, but at least a version number is registered for the given program number, the call returns a port number. Note: **pmap_getport()** returns the port number in host byte order. Some other network routines may require the port number in network byte order. For example, if the port number is used as part of the **sockaddr_in** structure, then it should be converted to network byte order using **htons**(3N).

enum clnt_stat pmap_rmtcall(addr, prognum, versnum, procnum, inproc, in, outproc, out, timeout, portp)
struct sockaddr_in *addr;
u_long prognum, versnum, procnum;
char *in, *out;
xdrproc_t inproc, outproc;
struct timeval timeout;
u_long *portp;

Request that the **portmap** on the host at IP address **addr* make an RPC on the behalf of the caller to a procedure on that host. **portp* is modified to the program's port number if the procedure succeeds. The definitions of other parameters are discussed in **callrpc()** and **clnt_call()** (see **rpc_clnt_calls**(3N)).

Warning: If the requested remote procedure is not registered with the remote **portmap** then no error response is returned and the call times out. Also, no authentication is done.

bool_t pmap_set(prognum, versnum, protocol, port)
u_long prognum, versnum;
int protocol;
u_short port;

Registers a mapping between the triple [*prognum*,*versnum*,*protocol*] and *port* on the local machine's **portmap** service. The value of *protocol* can be either **IPPROTO_UDP** or **IPPROTO_TCP**. This routine returns TRUE if it succeeds, FALSE otherwise. It is called by servers to register themselves with the local **portmap**. Automatically done by **svc_register()**.

bool_t pmap_unset(prognum, versnum)
u_long prognum, versnum;

Deregisters all mappings between the triple [*prognum*,*versnum*,*] and ports on the local machine's **portmap** service. It is called by servers to deregister themselves with the local **portmap**. This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_pmap(xdrs, regp)
XDR *xdrs;
struct pmap *regp;

Used for creating parameters to various **portmap** procedures, externally. This routine is useful for users who wish to generate these parameters without using the **pmap** interface. This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_pmaplist(xdrs, rp)
XDR *xdrs;
struct pmaplist **rp;

Used for creating a list of port mappings, externally. This routine is useful for users who wish to generate these parameters without using the **pmap** interface. This routine returns TRUE if it succeeds, FALSE otherwise.

SEE ALSO

rpc(3N), **portmap**(8C), **rpcinfo**(8C)

NAME

printf, fprintf, sprintf – formatted output conversion

SYNOPSIS

```
#include <stdio.h>

int printf(format [ , arg . . . ] )
char *format;

int fprintf(stream, format [ , arg . . . ] )
FILE *stream;
char *format;

char *sprintf(s, format [ , arg . . . ] )
char *s, *format;
```

SYSTEM V SYNOPSIS

The routines above are available as shown, except:

```
int sprintf(s, format [ , arg . . . ] )
char *s, *format;
```

The following are provided for XPG2 compatibility:

```
#define nl_printf             printf
#define nl_fprintf     fprintf
#define nl_sprintf     sprintf
```

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 of these functions converts, formats, and prints its *args* under control of the *format*. The *format* is a character string which contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of zero or more *args*. The results are undefined if there are insufficient *args* for the format. If the format is exhausted while *args* remain, the excess *args* are simply ignored.

Each conversion specification is introduced by either the % character or by the character sequence *%digit*$, after which the following appear in sequence:

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

A field width or precision or both 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 will be changed to zero.

The flag characters and their meanings are:

– The result of the conversion will be left-justified within the field.

+ The result of a signed conversion will always begin with a sign (+ or –).

blank If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and + flags both appear, the blank flag will be ignored.

# This flag specifies that the value is to be converted to an "alternate form". For **c**, **d**, **i**, **s**, and **u** conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x** or **X** conversion, a non-zero result will have **0x** or **0X** prefixed to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For **g** and **G** conversions, trailing zeroes will *not* be removed from the result (which they normally are).

The conversion characters and their meanings are:

d,i,o,p,u,x,X
The integer *arg* is converted to signed decimal (**d** or **i**), unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal notation (**x**, **p**, and **X**), respectively; the letters **abcdef** are used for **x** and **p** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. For compatibility with older versions, padding with leading zeroes may alternatively be specified by prepending a zero to the field width. This does not imply an octal value for the field width. The default precision is 1. The result of converting a zero value with a precision of zero is a null string.

f The float or double *arg* is converted to decimal notation in the style "[–]ddd.ddd" where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed.

e,E The float or double *arg* is converted in the style "[–]d.ddde±ddd," where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, 6 digits are produced; if the precision is zero, no decimal point appears. The **E** format code will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits.

g,G The float or double *arg* is printed in style **f** or **e** (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style **e** or **E** will be used only if the exponent resulting from the conversion is less than –4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.

The **e**, **E**, **f**, **g**, and **G** formats print IEEE indeterminate values (infinity or not-a-number) as "Infinity" or "NaN" respectively.

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 until the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed. A NULL value for *arg* will yield undefined results.

n The argument *arg* is a pointer to an integer into which is written the number of characters written to the output so far by this call to one of the **printf**( ) functions. No argument is converted.

% Print a %; no argument is converted.

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

All forms of the **printf**( ) functions allow for the insertion of a language dependent radix character in the output string. The radix character is defined by the program's locale (category **LC_NUMERIC**). In the "C" locale, or in a locale where the radix character is not defined, the radix character defaults to '.'.

Conversions can be applied to the *n*th argument in the argument list, rather than the next unused argument. In this case, the conversion character % is replaced by the sequence %*digit*$, where *digit* is a decimal integer *n* in the range [1,9], giving the position of the argument in the argument list. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages.

In format strings containing the %*digit*$ form of a conversion specification, a field width or precision may be indicated by the sequence ***digit*$, where *digit* is a decimal integer in the range [1,9] giving the position in the argument list of an integer *arg* containing the field width or precision.

The format string can contain either numbered argument specifications (that is, %*digit*$ and ***digit*$), or unnumbered argument specifications (that is % and *), but not both. The results of mixing numbered and unnumbered specifications is undefined. When numbered argument specifications are used, specifying the *n*th argument requires that all the leading arguments, from the first to the (*n*-1)th be specified in the format string.

SYSTEM V DESCRIPTION

XPG2 requires that **nl_printf**, **nl_fprintf** and **nl_sprintf** be defined as **printf**, **fprintf** and **sprintf**, respectively for backward compatibility

RETURN VALUES

On success, **printf**( ) and **fprintf**( ) return the number of characters transmitted, excluding the null character. On failure, they return EOF.

sprintf( ) returns *s*.

SYSTEM V RETURN VALUES

On success, **sprintf**( ) returns the number of characters transmitted, excluding the null character. On failure, it returns EOF.

EXAMPLES

```
printf(format, weekday, month, day, hour, min);
```

In American usage, *format* could be a pointer to the string:

```
"%s, %s %d, %d:%.2d\n"
```

producing the message:

```
Sunday, July 3,10:02
```

Whereas for German usage, *format* could be a pointer to the string:

```
"%1$s, %3$d.%2$s,%4$d:%5$.2d\n"
```

producing the message:

```
Sonntag, 3.Juli,10:02
```

To print π to 5 decimal places:

```
printf("pi = %.5f", 4 * atan(1. 0));
```

SEE ALSO

econvert(3), **putc**(3S), **scanf**(3V), **setlocale**(3V), **varargs**(3), **vprintf**(3V)

BUGS

Very wide fields (>128 characters) fail.

NAME

prof – profile within a function

SYNOPSIS

```
#define MARK
#include <prof.h>

void MARK (name)
```

DESCRIPTION

MARK introduces a mark called *name* that is treated the same as a function entry point. Execution of the mark adds to a counter for that mark, and program-counter time spent is accounted to the immediately preceding mark or to the function if there are no preceding marks within the active function.

name may be any combination of up to six letters, numbers or underscores. Each *name* in a single compilation must be unique, but may be the same as any ordinary program symbol.

For marks to be effective, the symbol MARK must be defined before the header file **<prof.h>** is included. This may be defined by a preprocessor directive as in the synopsis, or by a command line argument, such as:

```
cc –p –DMARK foo.c
```

If MARK is not defined, the MARK **(name)** statements may be left in the source files containing them and will be ignored.

EXAMPLE

In this example, marks can be used to determine how much time is spent in each loop. Unless this example is compiled with MARK defined on the command line, the marks are ignored.

```
#include <prof.h>
func( )
{
        int i, j;
        .
        .
        .
        MARK (loop1);
        for (i = 0; i < 2000; i++) {
                . . .
        }
        MARK (loop2);
        for (j = 0; j < 2000; j++) {
                . . .
        }
}
```

SEE ALSO

prof(1), **profil**(2), **monitor**(3)

NAME

psignal, sys_siglist – system signal messages

SYNOPSIS

```
psignal(sig, s)
unsigned sig;
char *s;

char *sys_siglist[ ];
```

DESCRIPTION

psignal() produces a short message on the standard error file describing the indicated signal. First the argument string *s* is printed, then a colon, then the name of the signal and a NEWLINE. Most usefully, the argument string is the name of the program which incurred the signal. The signal number should be from among those found in **<signal.h>**.

To simplify variant formatting of signal names, the vector of message strings **sys_siglist()** is provided; the signal number can be used as an index in this table to get the signal name without the newline. The define NSIG defined in **<signal.h>** is the number of messages provided for in the table; it should be checked because new signals may be added to the system before they are added to the table.

SEE ALSO

perror(3), **signal**(3V)

NAME

putc, putchar, fputc, putw – put character or word on a stream

SYNOPSIS

```
#include <stdio.h>

int putc(c, stream)
char c;
FILE *stream;

int putchar(c)
char c;

int fputc(c, stream)
char c;
FILE *stream;

int putw(w, stream)
int w;
FILE *stream;
```

DESCRIPTION

putc() writes the character *c* onto the standard I/O output stream *stream* (at the position where the file pointer, if defined, is pointing). It returns the character written.

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 C **int** (word) w to the standard I/O output stream *stream* (at the position of the file pointer, if defined). The size of a word is the size of an integer and varies from machine to machine. **putw()** neither assumes nor causes special alignment in the file.

Output streams are by default buffered if the output refers to a file and line-buffered if the output refers to a terminal. When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as written; when it is buffered, many characters are saved up and written as a block. When it is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a NEWLINE character is written or terminal input is requested). **setbuf**(3V), **setbuffer()**, or **setvbuf()** may be used to change the stream's buffering strategy.

SEE ALSO

fclose(3V), **ferror**(3V), **fopen**(3V), **fread**(3S), **getc**(3V), **printf**(3V), **puts**(3S), **setbuf**(3V)

DIAGNOSTICS

On success, **putc()**, **fputc()**, and **putchar()** return the value that was written. On error, those functions return the constant **EOF**. **putw()** returns **ferror(stream)**, so that it returns 0 on success and 1 on failure.

BUGS

Because it is implemented as a macro, **putc()** treats a *stream* argument with side effects improperly. In particular, **putc(c, *f++);** does not work sensibly. **fputc()** should be used instead.

Errors can occur long after the call to **putc()**.

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 will change the environment. The space used by *string* is no longer used once a new string-defining *name* is passed to **putenv**().

SEE ALSO

execve(2V), **getenv**(3V), **malloc**(3V), **environ**(5V)

DIAGNOSTICS

putenv() returns non-zero if it was unable to obtain enough space using **malloc**(3V) 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**(3V) to enlarge the environment.

After **putenv**() is called, environmental variables are not in alphabetical order.

A potential error is to call **putenv**() with an automatic variable as the argument, then exit the calling function while *string* is still part of the environment.

NAME

putpwent – write password file entry

SYNOPSIS

```
#include <pwd.h>

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

DESCRIPTION

putpwent() is the inverse of **getpwent**(3V). 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 lines in the password file **/etc/passwd**.

FILES

/etc/passwd

SEE ALSO

getpwent(3V)

DIAGNOSTICS

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

WARNING

The above routine uses **<stdio.h>**, which increases the size of programs, not otherwise using standard I/O, more than might be expected.

BUGS

This routine is of limited utility, since most password files are maintained as Network Information Service (NIS) files, and cannot be updated with this routine.

NOTES

The Network Information Service (NIS) was formerly known as Sun Yellow Pages (YP). The functionality of the two remains the same; only the name has changed.

NAME

puts, fputs – put a string on a stream

SYNOPSIS

```
#include <stdio.h>

puts(s)
char *s;

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

DESCRIPTION

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

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

Neither function writes the terminal null character.

DIAGNOSTICS

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

NOTES

puts( ) appends a NEWLINE while **fputs**( ) does not.

SEE ALSO

ferror(3V), **fopen**(3V), **fread**(3S), **printf**(3V), **putc**(3S)

NAME

pwdauth, grpauth – password authentication routines

SYNOPSIS

int pwdauth(user, password)
char *user;
char *password;

int grpauth(group, password)
char *group;
char *password;

DESCRIPTION

pwdauth() and **grpauth()** determine whether the given guess at a *password* is valid for the given *user* or *group*. If the *password* is valid, the functions return 0.

A *password* is valid if the password when encrypted matches the encrypted password in the appropriate file. For **pwdauth()**, if the **password.adjunct** file exists, the encrypted password will be in either the local or the Network Information Service (NIS) version of that file. Otherwise, either the local or NIS **passwd** file will be used. For **grpauth()**, the **group.adjunct** file (if it exists) or the **group** file (otherwise) will be checked on the local machine and then using the NIS service. In all cases, the local files will be checked before the NIS files. Also, if the adjunct files exist, the main file will never be used for authentication even if they include encrypted passwords.

Both **pwdauth()** and **grpauth()** interface to the authentication daemon, **rpc.pwdauthd**, to do the checking of the adjunct files. This daemon must be running on any system that provides password authentication.

FILES

/etc/passwd
/etc/group

SEE ALSO

getgraent(3), **getgrent**(3V), **getpwaent**(3), **getpwent**(3V), **pwdauthd**(8C)

NOTES

The Network Information Service (NIS) was formerly known as Sun Yellow Pages (YP). The functionality of the two remains the same; only the name has changed.

NAME

qsort – quicker sort

SYNOPSIS

```
qsort(base, nel, width, compar)
char *base;
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. *width* is the size, in bytes, of each element 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

sort(1V), **bsearch(3)**, **lsearch(3)**, **string(3)**

EXAMPLE

The following program sorts a simple array:

```
static	int intcompare(i,j)
int *i, *j;
{
	return(*i – *j);
}

main()
{
	int a[10];
	int i;

	a[0] = 9;
	a[1] = 8;
	a[2] = 7;
	a[3] = 6;
	a[4] = 5;
	a[5] = 4;
	a[6] = 3;
	a[7] = 2;
	a[8] = 1;
	a[9] = 0;

	qsort(a,10,sizeof(int),intcompare)

	for (i=0; i<10; i++) printf(" %d",a[i]);
	printf("\n");
}
```

NAME

rand, srand – simple random number generator

SYNOPSIS

srand(seed)
int seed;

rand()

DESCRIPTION

rand() uses a multiplicative congruential random number generator with period 2^{32} to return successive pseudo-random numbers in the range from 0 to $2^{31}-1$.

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.

SYSTEM V DESCRIPTION

rand() returns successive pseudo-random numbers in the range from 0 to $2^{15}-1$.

SEE ALSO

drand48(3), **random**(3)

NOTES

The spectral properties of **rand()** leave a great deal to be desired. **drand48**(3) and **random**(3) provide much better, though more elaborate, random-number generators.

BUGS

The low bits of the numbers generated are not very random; use the middle bits. In particular the lowest bit alternates between 0 and 1.

NAME

random, srandom, initstate, setstate – better random number generator; routines for changing generators

SYNOPSIS

```
long random()

srandom(seed)
int seed;

char *initstate(seed, state, n)
unsigned seed;
char *state;
int n;

char *setstate(state)
char *state;
```

DESCRIPTION

random() uses a non-linear additive feedback random number generator employing a default table of size 31 long integers to return successive pseudo-random numbers in the range from 0 to $2^{31}-1$. The period of this random number generator is very large, approximately $16\times(2^{31}-1)$.

random/srandom have (almost) the same calling sequence and initialization properties as **rand/srand**. The difference is that **rand**(3V) produces a much less random sequence — in fact, the low dozen bits generated by rand go through a cyclic pattern. All the bits generated by **random()** are usable. For example,

```
random()&01
```

will produce a random binary value.

Unlike **srand**, **srandom()** does not return the old seed; the reason for this is that the amount of state information used is much more than a single word. (Two other routines are provided to deal with restarting/changing random number generators). Like **rand**(3V), however, **random()** will by default produce a sequence of numbers that can be duplicated by calling **srandom()** with *1* as the seed.

The **initstate()** routine allows a state array, passed in as an argument, to be initialized for future use. The size of the state array (in bytes) is used by **initstate()** to decide how sophisticated a random number generator it should use — the more state, the better the random numbers will be. (Current ''optimal'' values for the amount of state information are 8, 32, 64, 128, and 256 bytes; other amounts will be rounded down to the nearest known amount. Using less than 8 bytes will cause an error). The seed for the initialization (which specifies a starting point for the random number sequence, and provides for restarting at the same point) is also an argument. **initstate()** returns a pointer to the previous state information array.

Once a state has been initialized, the **setstate()** routine provides for rapid switching between states. **setstate()** returns a pointer to the previous state array; its argument state array is used for further random number generation until the next call to **initstate()** or **setstate()**.

Once a state array has been initialized, it may be restarted at a different point either by calling **initstate()** (with the desired seed, the state array, and its size) or by calling both **setstate()** (with the state array) and **srandom()** (with the desired seed). The advantage of calling both **setstate()** and **srandom()** is that the size of the state array does not have to be remembered after it is initialized.

With 256 bytes of state information, the period of the random number generator is greater than 2^{69}, which should be sufficient for most purposes.

SEE ALSO

rand(3V)

EXAMPLES

```
/* Initialize and array and pass it in to initstate. */

static long state1[32] = {
        3,
        0x9a319039, 0x32d9c024, 0x9b663182, 0x5da1f342,
        0x7449e56b, 0xbeb1dbb0, 0xab5c5918, 0x946554fd,
        0x8c2e680f, 0xeb3d799f, 0xb11ee0b7, 0x2d436b86,
        0xda672e2a, 0x1588ca88, 0xe369735d, 0x904f35f7,
        0xd7158fd6, 0x6fa6f051, 0x616e6b96, 0xac94efdc,
        0xde3b81e0, 0xdf0a6fb5, 0xf103bc02, 0x48f340fb,
        0x36413f93, 0xc622c298, 0xf5a42ab8, 0x8a88d77b,
        0xf5ad9d0e, 0x8999220b, 0x27fb47b9
        };

main()
{
        unsigned seed;
        int n;

        seed = 1;
        n = 128;
        initstate(seed, (char *) state1, n);

        setstate(state1);
        printf("%d\n",random());
}
```

DIAGNOSTICS

If **initstate()** is called with less than 8 bytes of state information, or if **setstate()** detects that the state information has been garbled, error messages are printed on the standard error output.

WARNINGS

initstate() casts *state* to (**long** *), so *state* must be long-aligned. If it is not long-aligned, on some architectures the program will dump core.

BUGS

random() is only 2/3 as fast as **rand**(3V).

NAME

rcmd, rresvport, ruserok – routines for returning a stream to a remote command

SYNOPSIS

```
int rcmd(ahost, inport, locuser, remuser, cmd, fd2p)
char **ahost;
unsigned short inport;
char *locuser, *remuser, *cmd;
int *fd2p

int rresvport(port)
int *port;

ruserok(rhost, super-user, ruser, luser)
char *rhost;
int super-user;
char *ruser, *luser;
```

DESCRIPTION

rcmd( ) is a routine used by the super-user to execute a command on a remote machine using an authentication scheme based on reserved port numbers. **rresvport**( ) is a routine which returns a descriptor to a socket with an address in the privileged port space. **ruserok**( ) is a routine used by servers to authenticate clients requesting service with **rcmd**. All three functions are present in the same file and are used by the **rshd**(8C) server (among others).

rcmd( ) looks up the host **ahost* using **gethostbyname** (see **gethostent**(3N)), returning –1 if the host does not exist. Otherwise **ahost* is set to the standard name of the host and a connection is established to a server residing at the well-known Internet port *inport*.

If the connection succeeds, a socket in the Internet domain of type SOCK_STREAM is returned to the caller, and given to the remote command as its standard input (file descriptor 0) and standard output (file descriptor 1). If *fd2p* is non-zero, then an auxiliary channel to a control process will be set up, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (file descriptor 2) on this channel, and will also accept bytes on this channel as signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the standard error (file descriptor 2) of the remote command will be made the same as its standard output and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

The protocol is described in detail in **rshd**(8C).

The **rresvport**( ) routine is used to obtain a socket with a privileged address bound to it. This socket is suitable for use by **rcmd**( ) and several other routines. Privileged Internet ports are those in the range 0 to 1023. Only the super-user is allowed to bind an address of this sort to a socket.

ruserok( ) takes a remote host's name, as returned by a **gethostbyaddr** (see **gethostent**(3N)) routine, two user names and a flag indicating whether the local user's name is that of the super-user. It then checks the files **/etc/hosts.equiv** and, possibly, **.rhosts** in the local user's home directory to see if the request for service is allowed. A 0 is returned if the machine name is listed in the **/etc/hosts.equiv** file, or the host and remote user name are found in the **.rhosts** file; otherwise **ruserok**( ) returns –1. If the super-user flag is 1, the checking of the **/etc/hosts.equiv** file is bypassed.

FILES

/etc/hosts.equiv
.rhosts

SEE ALSO

rlogin(1C), **rsh**(1C), **intro**(2), **gethostent**(3N), **rexec**(3N), **rexecd**(8C), **rlogind**(8C), **rshd**(8C)

DIAGNOSTICS

rcmd() returns a valid socket descriptor on success. It returns −1 on error and prints a diagnostic message on the standard error.

rresvport() returns a valid, bound socket descriptor on success. It returns −1 on error with the global value **errno** set according to the reason for failure. The error code EAGAIN is overloaded to mean "All network ports in use."

NAME

realpath – return the canonicalized absolute pathname

SYNOPSIS

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

char *realpath(path, resolved_path)
char *path;
char resolved_path[MAXPATHLEN];
```

DESCRIPTION

realpath() expands all symbolic links and resolves references to '/./', '/../' and extra '/' characters in the null terminated string named by *path* and stores the canonicalized absolute pathname in the buffer named by *resolved_path*. The resulting path will have no symbolic links components, nor any '/./' or '/../' components.

RETURN VALUES

realpath() returns a pointer to the *resolved_path* on success. On failure, it returns NULL, sets **errno** to indicate the error, and places in *resolved_path* the absolute pathname of the *path* component which could not be resolved.

ERRORS

EACCES Search permission is denied for a component of the path prefix of *path*.

EFAULT *resolved_path* extends outside the process's allocated address space.

ELOOP Too many symbolic links were encountered in translating *path*.

EINVAL *path* or *resolved_path* was NULL.

EIO An I/O error occurred while reading from or writing to the file system.

ENAMETOOLONG The length of the path argument exceeds {PATH_MAX}.

A pathname component is longer than {NAME_MAX} (see **sysconf**(2V)) while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).

ENOENT The named file does not exist.

SEE ALSO

readlink(2), **getwd**(3)

WARNINGS

It indirectly invokes the **readlink(2)** system call and **getwd(3)** library call (for relative path names), and hence inherits the possibility of hanging due to inaccessible file system resources.

NAME

regex, re_comp, re_exec – regular expression handler

SYNOPSIS

```
char *re_comp(s)
char *s;

re_exec(s)
char *s;
```

DESCRIPTION

re_comp() compiles a string into an internal form suitable for pattern matching. **re_exec()** checks the argument string against the last string passed to **re_comp()**.

re_comp() returns a NULL pointer if the string *s* was compiled successfully; otherwise a string containing an error message is returned. If **re_comp()** is passed 0 or a null string, it returns without changing the currently compiled regular expression.

re_exec() returns 1 if the string *s* matches the last compiled regular expression, 0 if the string *s* failed to match the last compiled regular expression, and –1 if the compiled regular expression was invalid (indicating an internal error).

The strings passed to both **re_comp()** and **re_exec()** may have trailing or embedded NEWLINE characters; they are terminated by null characters. The regular expressions recognized are described in the manual entry for **ed**(1), given the above difference.

SEE ALSO

ed(1), **ex**(1), **grep**(1V)

DIAGNOSTICS

re_exec() returns –1 for an internal error.

re_comp() returns one of the following strings if an error occurs:

No previous regular expression

Regular expression too long

unmatched \(

missing]

too many \(\) pairs

unmatched \)

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

This page describes general-purpose regular expression matching routines.

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*) Returns the argument *c* by the next call to **GETC()** or **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 that have memory allocation to manage.

ERRORS

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 too long.

The syntax of the **compile()** routine is as follows:

compile(instring, expbuf, endbuf, eof)

The first parameter *instring* is never used explicitly by the **compile()** routine but is useful for programs that pass down different pointers to input characters. It is sometimes used in the **INIT()** declaration (see below). Programs that call functions to input characters or have characters in an external array can pass down a value of ((char *) 0) for this parameter.

The next parameter *expbuf* is a character pointer. It points to the place where the compiled regular expression 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 that marks the end of the regular expression. For example, in an editor like **ed**(1), this character would usually a '/'.

Each program that includes this file must have a **#define** statement for **INIT()**. This definition will be placed right after the declaration for the function **compile()** and '{' (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**(1V).

There are other functions in this file that perform actual regular expression matching, one of which is the function **step()**. The call to **step()** is as follows:

step(string, expbuf)

The first parameter to **step()** is a pointer to a string of characters to be checked for a match. This string should be null-terminated

The second parameter *expbuf* is the compiled regular expression that 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** will point to the first character of *string* and **loc2** will point to the null character at the end of *string*.

step() uses the external variable **circf** which is set by **compile()** if the regular expression begins with '^'. If this is set then **step()** will try 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 will advance its pointer to the string to be matched as far as possible and will recursively call itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, **advance()** will back up along the string until it finds a match or reaches the point in the string that initially matched the * or \{ \}. It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external character pointer **locs** is equal to the point in the string at sometime during the backing up process, **advance()** will break out of the loop that backs up and will return zero. This could be used by an editor like **ed**(1) or **sed**(1V) 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 is an example of how the regular expression macros and calls could look in a command like **grep**(1V):

```
#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), **grep**(1V), **sed**(1V)

BUGS

The handling of **circf** is difficult.

NAME

resolver, res_mkquery, res_send, res_init, dn_comp, dn_expand – resolver routines

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

res_mkquery(op, dname, class, type, data, datalen, newrr, buf, buflen)
int op;
char *dname;
int class, type;
char *data;
int datalen;
struct rrec *newrr;
char *buf;
int buflen;

res_send(msg, msglen, answer, anslen)
char *msg;
int msglen;
char *answer;
int anslen;

res_init()

dn_comp(exp_dn, comp_dn, length, dnptrs, lastdnptr)
u_char *exp_dn, *comp_dn;
int length;
u_char **dnptrs, **lastdnptr;

dn_expand(msg, msglen, comp_dn, exp_dn, length)
u_char *msg, *eomorig, *comp_dn, exp_dn;
int length;
```

DESCRIPTION

These routines are used for making, sending and interpreting packets to Internet domain name servers. You can link a program with the resolver library using the **–lresolv** argument on the linking command line.

Global information that is used by the resolver routines is kept in the variable *_res*. Most of the values have reasonable defaults and can be ignored. Options are a simple bit mask and are OR'ed in to enable. Options stored in *_res.options* are defined in **<resolv.h>** and are as follows.

RES_INIT True if the initial name server address and default domain name are initialized (that is, **res_init**() has been called).

RES_DEBUG Print debugging messages.

RES_AAONLY Accept authoritative answers only. **res_send**() continues until it finds an authoritative answer or finds an error. Currently this is not implemented.

RES_USEVC Use TCP connections for queries instead of UDP.

RES_STAYOPEN Used with **RES_USEVC** to keep the TCP connection open between queries. This is useful only in programs that regularly do many queries. UDP should be the normal mode used.

RES_IGNTC Unused currently (ignore truncation errors, that is, do not retry with TCP).

RES_RECURSE Set the recursion desired bit in queries. This is the default. **res_send**() does not do iterative queries and expects the name server to handle recursion.

RES_DEFNAMES	Append the default domain name to single label queries. This is the default.
RES_DNSRCH	Search up the domain tree from the default domain, in all but the top level. This is the default.

res_init( ) reads the initialization file to get the default domain name and the Internet addresses of the initial name servers. If no **nameserver** line exists, the host running the resolver is tried. **res_mkquery**( ) makes a standard query message and places it in *buf*. **res_mkquery**( ) returns the size of the query or −1 if the query is larger than *buflen*. *op* is usually **QUERY** but can be any of the query types defined in **<nameser.h>**. *dname* is the domain name. If *dname* consists of a single label and the **RES_DEFNAMES** flag is enabled (the default), *dname* is appended with the current domain name. The current domain name is defined in a system file and can be overridden by the environment variable **LOCALDOMAIN**. *newrr* is currently unused but is intended for making update messages.

res_send( ) sends a query to name servers and returns an answer. It calls **res_init**( ) if **RES_INIT** is not set, send the query to the local name server, and handle timeouts and retries. The length of the message is returned or −1 if there were errors.

dn_expand( ) Expands the compressed domain name *comp_dn* to a full domain name. Expanded names are converted to upper case. *msg* is a pointer to the beginning of the message, *exp_dn* is a pointer to a buffer of size *length* for the result. The size of compressed name is returned or −1 if there was an error.

dn_comp( ) Compresses the domain name *exp_dn* and stores it in *comp_dn*. The size of the compressed name is returned or −1 if there were errors. *length* is the size of the array pointed to by *comp_dn*. *dnptrs* is a list of pointers to previously compressed names in the current message. The first pointer points to the beginning of the message and the list ends with NULL. *lastdnptr* is a pointer to the end of the array pointed to *dnptrs*. A side effect is to update the list of pointers for labels inserted into the message by **dn_comp**( ) as the name is compressed. If *dnptr* is NULL, do not try to compress names. If *lastdnptr* is NULL, do not update the list.

FILES

/etc/resolv.conf	see **resolv.conf(5)**
/usr/lib/libresolv.a	

SEE ALSO

resolv.conf(5), **named(8C)**

System and Network Administration

NOTES

/usr/lib/libresolv.a is necessary for compiling programs.

NAME

rexec – return stream to a remote command

SYNOPSIS

```
rem = rexec(ahost, inport, user, passwd, cmd, fd2p);
char **ahost;
u_short inport;
char *user, *passwd, *cmd;
int *fd2p;
```

DESCRIPTION

rexec() looks up the host **ahost* using **gethostbyname()** (see **gethostent**(3N)), returning –1 if the host does not exist. Otherwise **ahost* is set to the standard name of the host. If a username and password are both specified, then these are used to authenticate to the foreign host; otherwise the environment and then the user's **.netrc** file in his home directory are searched for appropriate information. If all this fails, the user is prompted for the information.

The port **inport** specifies which well-known DARPA Internet port to use for the connection; it will normally be the value returned from the call '**getservbyname("exec", "tcp")**' (see **getservent**(3N)). The protocol for connection is described in detail in **rexecd**(8C).

If the call succeeds, a socket of type **SOCK_STREAM** is returned to the caller, and given to the remote command as its standard input and standard output. If *fd2p* is non-zero, then a auxiliary channel to a control process will be setup, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the standard error (unit 2 of the remote command) will be made the same as its standard output and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

SEE ALSO

gethostent(3N), **getservent**(3N), **rcmd**(3N), **rexecd**(8C)

BUGS

There is no way to specify options to the **socket()** call that **rexec()** makes.

NAME

rpc – library routines for remote procedure calls

SYNOPSIS AND DESCRIPTION

RPC routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a request to the server. Upon receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends back a reply. Finally, the procedure call returns to the client.

All RPC routines require the header **<rpc/rpc.h>** to be included.

The RPC routines have been grouped by usage on the following man pages.

portmap(3N) Library routines for the RPC bind service, **portmap**(8C). The routines documented on this page include:

pmap_getmaps()
pmap_getport()
pmap_rmtcall()
pmap_set()
pmap_unset()
xdr_pmap()
xdr_pmaplist()

rpc_clnt_auth(3N) Library routines for client side remote procedure call authentication. The routines documented on this page include:

auth_destroy()
authnone_create()
authunix_create()
authunix_create_default()

rpc_clnt_calls(3N) Library routines for client side calls. The routines documented on this page include:

callrpc()
clnt_broadcast()
clnt_call()
clnt_freeres()
clnt_geterr()
clnt_perrno()
clnt_perror()
clnt_sperrno()
clnt_sperror()

rpc_clnt_create(3N) Library routines for dealing with the creation and manipulation of CLIENT handles. The routines documented on this page include:

clnt_control()
clnt_create()
clnt_create_vers()
clnt_destroy()
clnt_pcreateerror()
clntraw_create()
clnt_spcreateerror()
clnttcp_create()
clntudp_bufcreate()
clntudp_create()
rpc_createrr()

rpc_svc_calls(3N) Library routines for registerring servers. The routines documented on this page include:

- **registerrpc()**
- **svc_register()**
- **svc_unregister()**
- **xprt_register()**
- **xprt_unregister()**

rpc_svc_create(3N) Library routines for dealing with the creation of server side handles. The routines documented on this page include:

- **svc_destroy()**
- **svcfd_create()**
- **svcraw_create()**
- **svctcp_create()**
- **svcudp_bufcreate()**

rpc_svc_err(3N) Library routines for server side remote procedure call errors. The routines documented on this page include:

- **svcerr_auth()**
- **svcerr_decode()**
- **svcerr_noproc()**
- **svcerr_noprog()**
- **svcerr_progvers()**
- **svcerr_systemerr()**
- **svcerr_weakauth()**

rpc_svc_reg(3N) Library routines for RPC servers. The routines documented on this page include:

- **svc_fds()**
- **svc_fdset()**
- **svc_freeargs()**
- **svc_getargs()**
- **svc_getcaller()**
- **svc_getreq()**
- **svc_getreqset()**
- **svc_run()**
- **svc_sendreply()**

rpc_xdr(3N) XDR library routines for remote procedure calls. The routines documented on this page include:

- **xdr_accepted_reply()**
- **xdr_authunix_parms()**
- **xdr_callhdr()**
- **xdr_callmsg()**
- **xdr_opaque_auth()**
- **xdr_rejected_reply()**
- **xdr_replymsg()**

secure_rpc(3N) Library routines for secure remote procedure calls. The routines documented on this page include:

authdes_create()
authdes_getucred()
get_mayaddress()
getnetname()
host2netname()
key_decryptsession()
key_encryptsession()
key_gendes()
key_setsecret()
netname2host()
netname2user()
user2netname()

SEE ALSO

portmap(3N), **rpc_clnt_auth**(3N), **rpc_clnt_calls**(3N), **rpc_clnt_create**(3N), **rpc_svc_calls**(3N), **rpc_svc_create**(3N), **rpc_svc_err**(3N), **rpc_svc_reg**(3N), **rpc_xdr**(3N), **secure_rpc**(3N), **xdr**(3N), **publickey**(5), **portmap**(8C), **keyserv**(8C)

Network Programming

NAME

auth_destroy, authnone_create, authunix_create, authunix_create_default – library routines for client side remote procedure call authentication

DESCRIPTION

RPC routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a request to the server. Upon receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends back a reply. Finally, the procedure call returns to the client.

RPC allows various authentication types. Currently, it supports **AUTH_NONE**, **AUTH_UNIX**, **AUTH_DES**. For routines relating to the **AUTH_DES** type, see **secure_rpc**(3N).

These routines are called after creating the CLIENT handle. The client's authentication information is passed to the server when the RPC call is made.

Routines

The following routines require that the header **<rpc.h>**. be included. The **AUTH** data structure is defined in the RPC/XDR Library Definitions of the *Network Programming*.

#include <rpc/rpc.h>

void auth_destroy(auth)
AUTH *auth;

Destroy the authentication information associated with *auth*. Destruction usually involves deallocation of private data structures. The use of *auth* is undefined after calling **auth_destroy()**.

AUTH * authnone_create( )

Create and return an RPC authentication handle that passes no usable authentication information with each remote procedure call. This is the default authentication used by RPC.

AUTH * authunix_create(host, uid, gid, grouplen, gidlistp)
char *host;
int uid, gid, grouplen, *gidlistp;

Create and return an RPC authentication handle that contains authentication information. The parameter *host* is the name of the machine on which the information was created; *uid* is the user's user ID; *gid* is the user's current group ID; *grouplen* and *gidlistp* refer to a counted array of groups to which the user belongs. Warning: It is not very difficult to impersonate a user.

AUTH * authunix_create_default( )

Call **authunix_create( )** with the appropriate parameters.

SEE ALSO

rpc(3N), **rpc_clnt_create**(3N), **rpc_clnt_calls**(3N)

NAME

callrpc, clnt_broadcast, clnt_call, clnt_freeres, clnt_geterr, clnt_perrno, clnt_perror, clnt_sperrno, clnt_sperror – library routines for client side calls

DESCRIPTION

RPC routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a request to the server. Upon receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends back a reply. Finally, the procedure call returns to the client.

The **clnt_call( )**, **callrpc( )** and **clnt_broadcast( )** routines handle the client side of the procedure call. The remaining routines deal with error handling in the case of errors.

Routines

The **CLIENT** data structure is defined in the RPC/XDR Library Definition of the *Network Programming*.

#include <rpc/rpc.h>

```
int callrpc(host, prognum, versnum, procnum, inproc, in, outproc, out)
char *host;
u_long prognum, versnum, procnum;
char *in;
xdrproc_t inproc;
char *out;
xdrproc_t outproc;
```

Call the remote procedure associated with *prognum*, *versnum*, and *procnum* on the machine, *host*. The parameter *in* is the address of the procedure's argument, and *out* is the address of where to place the result; *inproc* is an XDR function used to encode the procedure's parameters, and *outproc* is an XDR function used to decode the procedure's results. This routine returns 0 if it succeeds, or the value of **enum clnt_stat** cast to an integer if it fails. Use **clnt_perrno( )** to translate failure statuses into messages.

Warning: Calling remote procedures with this routine uses UDP/IP as the transport; see **clntudp_create( )** on **rpc_clnt_create**(3N) for restrictions. You do not have control of timeouts or authentication using this routine.

```
enum clnt_stat clnt_broadcast(prognum, versnum, procnum, inproc, in, outproc, out, eachresult)
u_long prognum, versnum, procnum;
char *in;
xdrproc_t inproc;
char *out;
xdrproc_t outproc;
bool_t eachresult;
```

Like **callrpc( )**, except the call message is broadcast to all locally connected broadcast nets. Each time the caller receives a response, this routine calls **eachresult( )**, whose form is:

```
int eachresult(out, addr)
char *out;
struct sockaddr_in *addr;
```

where *out* is the same as *out* passed to **clnt_broadcast( )**, except that the remote procedure's output is decoded there; *addr* points to the address of the machine that sent the results. If **eachresult( )** returns 0 **clnt_broadcast( )** waits for more replies; otherwise it returns with appropriate status. If **eachresult( )** is NULL, **clnt_broadcast( )** returns without waiting for any replies.

Note: **clnt_broadcast( )** uses **AUTH_UNIX** style of authentication.

Warning: Broadcast packets are limited in size to the maximum transfer unit of the data link. For Ethernet, the callers argument size should not exceed 1400 bytes.

enum clnt_stat clnt_call(clnt, procnum, inproc, in, outproc, out, timeout)
CLIENT *clnt;
u_long procnum;
xdrproc_t inproc, outproc;
char *in, *out;
struct timeval timeout;

Call the remote procedure *procnum* associated with the client handle, *clnt*, which is obtained with an RPC client creation routine such as **clnt_create( )** (see **rpc_clnt_create**(3N). The parameter *in* is the address of the procedure's argument, and *out* is the address of where to place the result; *inproc* is an XDR function used to encode the procedure's parameters in XDR, and *outproc* is used to decode the procedure's results; *timeout* is the time allowed for a response from the server.

bool_t clnt_freeres(clnt, outproc, out)
CLIENT *clnt;
xdrproc_t outproc;
char *out;

Free any data allocated by the RPC/XDR system when it decoded the results of an RPC call. The parameter *out* is the address of the results, and *outproc* is the XDR routine describing the results. This routine returns TRUE if the results were successfully freed, and FALSE otherwise. Note: This is equivalent to doing **xdr_free(outproc, out)** (see **xdr_simple**(3N)).

void clnt_geterr(clnt, errp)
CLIENT *clnt;
struct rpc_err *errp;

Copy the error structure out of the client handle to the structure at address *errp*. *errp* should point to preallocated space.

void clnt_perrno(stat)
enum clnt_stat stat;

Print a message to the standard error corresponding to the condition indicated by *stat*. A NEWLINE is appended at the end of the message. Used after **callrpc( )** or **clnt_broadcast( )**.

void clnt_perror(clnt, str)
CLIENT *clnt;
char *str;

Print a message to the standard error indicating why an RPC call failed; *clnt* is the handle used to do the call. The message is prepended with string *s* and a colon. A NEWLINE is appended at the end of the message. Used after **clnt_call( )**.

char *clnt_sperrno(stat)
enum clnt_stat stat;

Take the same arguments as **clnt_perrno( )**, but instead of sending a message to the standard error indicating why an RPC failed, return a pointer to a string which contains the message. **clnt_sperrno( )** does not append a NEWLINE at the end of the message.

clnt_sperrno( ) is used instead of **clnt_perrno( )** if the program does not have a standard error (as a program running as a server quite likely does not), or if the programmer does not want the message to be output with **printf**(3V), or if a message format different than that supported by **clnt_perrno( )** is to be used.

```
char *clnt_sperror(clnt, str)
CLIENT *clnt;
char *str;
```

Like **clnt_perror( )**, except that (like **clnt_sperrno( )**) it returns a string instead of printing to the standard error. Unlike **clnt_perror( )**, it does not append the message with a NEWLINE.

Note: **clnt_sperror( )** returns pointer to a static buffer that is overwritten on each call.

SEE ALSO

printf(3V), **rpc**(3N), **rpc_clnt_auth**(3N), **rpc_clnt_create**(3N), **xdr_simple**(3N)

NAME

clnt_control, clnt_create, clnt_create_vers, clnt_destroy, clnt_pcreateerror, clntraw_create, clnt_spcreateerror, clnttcp_create, clntudp_bufcreate, rpc_createrr – library routines for dealing with creation and manipulation of CLIENT handles

DESCRIPTION

RPC routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a request to the server. Upon receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends back a reply. Finally, the procedure call returns to the client.

The CLIENT data structure is defined in the RPC/XDR Library Definition of the *Network Programming*.

```
#include <rpc/rpc.h>
```

```
bool_t clnt_control(clnt, request, info)
CLIENT *clnt;
int request;
char *info;
```

Change or retrieve various information about a client object. *request* indicates the type of operation, and *info* is a pointer to the information. For both UDP and TCP, the supported values of *request* and their argument types and what they do are:

CLSET_TIMEOUT	struct timeval	set total timeout
CLGET_TIMEOUT	struct timeval	get total timeout
CLGET_FD	int	get associated socket
CLSET_FD_CLOSE	void	close socket on **clnt_destroy()**
CLSET_FD_NCLOSE	void	leave socket open on **clnt_destroy()**

Note: If you set the timeout using **clnt_control()**, the timeout parameter passed to **clnt_call()** (see **rpc_clnt_calls(3N)**) will be ignored in all future calls.

CLGET_SERVER_ADDR	struct sockaddr_in	get server's address

The following operations are valid for UDP only:

CLSET_RETRY_TIMEOUT	struct timeval	set the retry timeout
CLGET_RETRY_TIMEOUT	struct timeval	get the retry timeout

The retry timeout is the time that UDP RPC waits for the server to reply before retransmitting the request.

This routine returns TRUE on success, and FALSE on failure.

```
CLIENT * clnt_create(host, prognum, versnum, protocol)
char *host;
u_long prognum, versnum;
char *protocol;
```

Generic client creation routine for program *prognum* and version *versnum*. *host* identifies the name of the remote host where the server is located. *protocol* indicates which kind of transport protocol to use. The currently supported values for this field are "udp" and "tcp". Default timeouts are set, but they can be modified using **clnt_control()**. If successful it returns a client handle, otherwise it returns NULL.

Warning: Using UDP has its shortcomings. Since UDP-based RPC messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take arguments or return results larger than 8 Kbytes. Use TCP instead.

Note: If the requested version number *versnum* is not registered with the **portmap(8C)** service on *host*, but at least a version number for the given program number is registered, **clnt_create()** returns a handle. The version mismatch will be discovered by a **clnt_call()** later (see **rpc_clnt_calls(3N)**).

CLIENT * clnt_create_vers(host, prognum, vers_outp, vers_low, vers_high, protocol)
char *host;
u_long prognum;
u_long *vers_outp;
u_long vers_low, vers_high;
char *protocol;

This is a generic client creation routine which also checks for the version available. *host* identifies the name of the remote host where the server is located. *protocol* indicates which kind of transport protocol to use. The currently supported values for this field are "udp" and "tcp". If the routine is successful it returns a client handle created for the highest version between *vers_low* and *vers_high* that is supported by the server. *vers_outp* is set to this value. That is, after a successful return *vers_low* <= **vers_outp* <= *vers_high*. If no version between *vers_low* and *vers_high* is supported by the server then the routine fails and returns NULL. Default timeouts are set, but can be modified using **clnt_control()**.

Note: **clnt_create()** returns a valid client handle even if the particular version number supplied to **clnt_create()** is not registered with the portmap service. This mismatch will be discovered by a **clnt_call()** later (see **rpc_clnt_calls(3N)**). However, **clnt_create_vers()** does this for you and returns a valid handle only if a version within the range supplied is supported by the server.

void clnt_destroy(clnt)
CLIENT *clnt;

Destroy the client's RPC handle. Destruction usually involves deallocation of private data structures, including *clnt* itself. Use of *clnt* is undefined after calling **clnt_destroy()**. If the RPC library opened the associated socket, or **CLSET_FD_CLOSE** was set using **clnt_control()**. **clnt_destroy()** closes the socket.

void clnt_pcreateerror(str)
char *str;

Print a message to the standard error indicating why a client handle could not be created. The message is prepended with string *s* and a colon. Used when routines such as **clnt_create()**, **clntraw_create()**, **clnttcp_create()**, or **clntudp_create()** fails.

CLIENT * clntraw_create(prognum, versnum)
u_long prognum, versnum;

Create an RPC client for the remote program *prognum*, version *versnum*. The transport used to pass messages to the service is actually a buffer within the process's address space, so the corresponding RPC server should live in the same address space; also see **svcraw_create()** (see **rpc_svc_create(3N)**). This allows simulation of RPC and getting RPC overheads, such as round trip times, without any kernel interference. If successful it returns a client handle, otherwise it returns NULL.

char * clnt_spcreateerror(str)
char *str;

> Like **clnt_pcreateerror**(), except that it returns a string instead of printing to the standard error. It, however, does not append the message with a NEWLINE.
>
> Note: **clnt_spcreateerror**() returns a pointer to a static buffer that is overwritten on each call.

CLIENT * **clnttcp_create(addr, prognum, versnum, sockp, sendsz, recvsz)**
struct sockaddr_in *addr;
u_long prognum, versnum;
int *sockp;
u_int sendsz, recvsz;

> Create a client handle for the remote program *prognum*, version *versnum*; the client uses TCP/IP as a transport. The remote program is located at Internet address *addr*. If **addr–>sin_port** is zero, it is set to the port on which the remote program is listening (the remote **portmap** service is consulted for this information). The parameter *sockp* is a pointer to a socket; if it is **RPC_ANYSOCK**, then a new socket is opened and *sockp* is updated. Since TCP-based RPC uses buffered I/O, the user may specify the size of the send and receive buffers with the parameters *sendsz* and *recvsz*; values of zero choose defaults. If successful it returns a client handle, otherwise it returns NULL.
>
> Warning: If **addr–>sin_port** is zero and the requested version number *versnum* is not registered with the remote portmap service, it returns a handle if at least a version number for the given program number is registered. The version mismatch will be discovered by a **clnt_call()** later (see **rpc_clnt_calls**(3N)).

CLIENT * **clntudp_bufcreate(addr, prognum, versnum, wait, sockp, sendsz, recvsz)**
struct sockaddr_in *addr;
u_long prognum, versnum;
struct timeval wait;
int *sockp;
u_int sendsz;
u_int recvsz;

> Create a client handle for the remote program *prognum*, on *versnum*; the client uses UDP/IP as the transport. The remote program is located at the Internet address *addr*. If **addr–>sin_port** is zero, it is set to port on which the remote program is listening on (the remote **portmap** service is consulted for this information). The parameter *sockp* is a pointer to a socket; if it is **RPC_ANYSOCK**, then a new socket is opened and **sockp** is updated. The UDP transport resends the call message in intervals of *wait* time until a response is received or until the call times out. The total time for the call to time out is specified by **clnt_call**() (see **rpc_clnt_calls**(3N)). If successful it returns a client handle, otherwise it returns NULL.
>
> The user can specify the maximum packet size for sending and receiving by using *sendsz* and *recvsz* arguments for UDP-based RPC messages.
>
> Warning: If **addr–>sin_port** is zero and the requested version number *versnum* is not registered with the remote portmap service, it returns a handle if at least a version number for the given program number is registered. The version mismatch is discovered by a **clnt_call**() later (see **rpc_clnt_calls**(3N)).

CLIENT * clntudp_create(addr, prognum, versnum, wait, sockp)
struct sockaddr_in *addr;
u_long prognum, versnum;
struct timeval wait;
int *sockp;

Create a client handle for the remote program *prognum*, version *versnum*; the client uses UDP/IP as the transport. The remote program is located at the Internet address *addr*. If - **addr–>sin_port** is zero, then it is set to actual port that the remote program is listening on (the remote **portmap** service is consulted for this information). The parameter *sockp* is a pointer to a socket; if it is **RPC_ANYSOCK**, a new socket is opened and *sockp* is updated. The UDP transport resends the call message in intervals of *wait* time until a response is received or until the call times out. The total time for the call to time out is specified by **clnt_call( )** (see **rpc_clnt_calls**(3N)). If successful it returns a client handle, otherwise it returns NULL.

Warning: Since UDP-based RPC messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take arguments or results larger than 8 Kbytes. TCP should be used instead.

Warning: If **addr–>sin_port** is zero and the requested version number *versnum* is not registered with the remote portmap service, it returns a handle if any version number for the given program number is registered. The version mismatch is be discovered by a **clnt_call( )** later (see **rpc_clnt_calls**(3N)).

struct rpc_createerr rpc_createerr;

A global variable whose value is set by any RPC client handle creation routine that fails. It is used by the routine **clnt_pcreateerror( )** to print the reason for the failure.

SEE ALSO

portmap(3N), **rpc**(3N), **rpc_clnt_auth**(3N), **rpc_clnt_calls**(3N), **rpc_svc_create**(3N)

NAME

registerrpc, svc_register, svc_unregister, xprt_register, xprt_unregister – library routines for registerring servers

DESCRIPTION

These routines are a part of the RPC library which allows the RPC servers to register themselves with **portmap(8C)**, and it associates the given program and version number with the dispatch function.

Routines

The SVCXPRT data structure is defined in the RPC/XDR Library Definition of the *Network Programming*.

#include <rpc/rpc.h>

```
int registerrpc(prognum, versnum, procnum, procname, inproc, outproc)
u_long prognum, versnum, procnum;
char *(*procname) () ;
xdrproc_t inproc, outproc;
```

Register procedure *procname* with the RPC service package. If a request arrives for program *prognum*, version *versnum*, and procedure *procnum*, *procname* is called with a pointer to its parameter; *progname* must be a procedure that returns a pointer to its static result; *inproc* is used to decode the parameters while *outproc* is used to encode the results. This routine returns 0 if the registration succeeded, −1 otherwise.

Warning: Remote procedures registered in this form are accessed using the UDP/IP transport; see **svcudp_create()** on **rpc_svc_create(3N)** for restrictions. This routine should not be used more than once for the same program and version number.

```
bool_t svc_register(xprt, prognum, versnum, dispatch, protocol)
SVCXPRT *xprt;
u_long prognum, versnum;
void (*dispatch) ();
u_long protocol;
```

Associates *prognum* and *versnum* with the service dispatch procedure, *dispatch*. If *protocol* is zero, the service is not registered with the **portmap** service. If *protocol* is non-zero, a mapping of the triple [*prognum, versnum, protocol*] to **xprt->xp_port** is established with the local **portmap** service (generally *protocol* is zero, **IPPROTO_UDP** or **IPPROTO_TCP**). The procedure *dispatch* has the following form:

```
dispatch(request, xprt)
struct svc_req *request;
SVCXPRT *xprt;
```

The svc_register() routine returns TRUE if it succeeds, and FALSE otherwise.

```
void svc_unregister(prognum, versnum)
u_long prognum, versnum;
```

Remove all mapping of the pair [*prognum,versnum*] to dispatch routines, and of the triple [*prognum,versnum,**] to port number.

```
void xprt_register(xprt)
SVCXPRT *xprt;
```

After RPC service transport handles are created, they should register themselves with the RPC service package. This routine modifies the global variable **svc_fds**. Service implementors usually do not need this routine.

void xprt_unregister(xprt)
SVCXPRT *xprt;

> Before an RPC service transport handle is destroyed, it should unregister itself with the RPC service package. This routine modifies the global variable svc_fds. Service implementors usually do not need this routine directly.

SEE ALSO

portmap(3N), **rpc**(3N), **rpc_svc_err**(3N), **rpc_svc_create**(3N), **rpc_svc_reg**(3N), **portmap**(8C)

NAME

svc_destroy, svcfd_create, svcraw_create, svctcp_create, svcudp_bufcreate – library routines for dealing with the creation of server handles

DESCRIPTION

RPC routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a request to the server. Upon receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends back a reply. Finally, the procedure call returns to the client.

The **SVCXPRT** data structure is defined in the RPC/XDR Library Definitions of the *Network Programming*.

#include <rpc/rpc.h>

void svc_destroy(xprt)
SVCXPRT *xprt;

Destroy the RPC service transport handle, *xprt*. Destruction usually involves deallocation of private data structures, including *xprt* itself. Use of *xprt* is undefined after calling this routine.

SVCXPRT * svcfd_create(fd, sendsz, recvsz)
int fd;
u_int sendsz;
u_int recvsz;

Create a service on top of any open and bound descriptor and return the handle to it. Typically, this descriptor is a connected socket for a stream protocol such as TCP. *sendsz* and *recvsz* indicate sizes for the send and receive buffers. If they are zero, a reasonable default is chosen. It returns NULL if it fails.

SVCXPRT * svcraw_create()

This routine creates a RPC service transport, to which it returns a pointer. The transport is a buffer within the process's address space, so the corresponding RPC client must live in the same address space; see **clntraw_create()** on **rpc_clnt_create**(3N). This routine allows simulation of RPC and getting RPC overheads (such as round trip times), without any kernel interference. This routine returns NULL if it fails.

SVCXPRT * svctcp_create(sock, sendsz, recvsz)
int sock;
u_int sendsz, recvsz;

This routine creates a TCP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket *sock*. If sock is **RPC_ANYSOCK**, then a new socket is created. If the socket is not bound to a local TCP port, then this routine binds it to an arbitrary port. Upon completion, **xprt–>xp_sock** is the transport's socket descriptor, and **xprt–>xp_port** is the port number on which it is listening. This routine returns NULL if it fails. Since TCP-based RPC uses buffered I/O, users may specify the size of buffers with *sendsz* and *recvsz*; values of zero choose defaults.

SVCXPRT * svcudp_bufcreate(sock, sendsz, recvsz)
int sock;
u_int sendsz, recvsz;

This routine creates a UDP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket *sock*. If sock is **RPC_ANYSOCK** , then a new socket is created. If the socket is not bound to a local UDP port, then this routine binds it to an arbitrary port. Upon completion, **xprt−>xp_sock** is the service's socket descriptor, and **xprt−>xp_port** is the service's port number. This routine returns NULL if it fails.

The user specifies the maximum packet size for sending and receiving UDP-based RPC messages by using the *sendsz* and *recvsz* parameters.

SEE ALSO

rpc(3N), **rpc_clnt_create**(3N), **rpc_svc_calls**(3N), **rpc_svc_err**(3N), **rpc_svc_reg**(3N), **portmap**(8C)

NAME

svcerr_auth, svcerr_decode, svcerr_noproc, svcerr_noprog, svcerr_progvers, svcerr_systemerr, svcerr_weakauth – library routines for server side remote procedure call errors

DESCRIPTION

RPC routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a request to the server. Upon receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends back a reply. Finally, the procedure call returns to the client.

These routines can be called by the server side dispatch function if there is any error in the transaction with the client.

Routines

The **SVCXPRT** data structure is defined in the RPC/XDR Library Definitions of the *Network Programming*.

#include <rpc/rpc.h>

void svcerr_auth(xprt, why)
SVCXPRT *xprt;
enum auth_stat why;

Called by a service dispatch routine that refuses to perform a remote procedure call due to an authentication error.

void svcerr_decode(xprt)
SVCXPRT *xprt;

Called by a service dispatch routine that cannot successfully decode the remote parameters. See **svc_getargs( )** in **rpc_svc_reg**(3N).

void svcerr_noproc(xprt)
SVCXPRT *xprt;

Called by a service dispatch routine that does not implement the procedure number that the caller requests.

void svcerr_noprog(xprt)
SVCXPRT *xprt;

Called when the desired program is not registered with the RPC package. Service implementors usually do not need this routine.

void svcerr_progvers(xprt)
SVCXPRT *xprt;

Called when the desired version of a program is not registered with the RPC package. Service implementors usually do not need this routine.

void svcerr_systemerr(xprt)
SVCXPRT *xprt;

Called by a service dispatch routine when it detects a system error not covered by any particular protocol. For example, if a service can no longer allocate storage, it may call this routine.

void svcerr_weakauth(xprt)
SVCXPRT *xprt;

Called by a service dispatch routine that refuses to perform a remote procedure call due to insufficient authentication parameters. The routine calls **svcerr_auth(xprt, AUTH_TOOWEAK)**.

SEE ALSO

rpc(3N), **rpc_svc_calls**(3N), **rpc_svc_create**(3N), **rpc_svc_reg**(3N)

NAME

svc_fds, svc_fdset, svc_freeargs, svc_getargs, svc_getcaller, svc_getreq, svc_getreqset, svc_getcaller, svc_run, svc_sendreply – library routines for RPC servers

DESCRIPTION

RPC routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a request to the server. Upon receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends back a reply. Finally, the procedure call returns to the client.

These routines are associated with the server side of the RPC mechanism. Some of them are called by the server side dispatch function, while others (such as **svc_run()**) are called when the server is initiated.

Routines

The SVCXPRT data structure is defined in the RPC/XDR Library Definitions of the *Network Programming*.

#include <rpc/rpc.h>

int svc_fds;

Similar to **svc_fdset**, but limited to 32 descriptors. This interface is obsoleted by **svc_fdset**.

fd_set svc_fdset;

A global variable reflecting the RPC server's read file descriptor bit mask; it is suitable as a parameter to the **select()** system call. This is only of interest if a service implementor does not call **svc_run()**, but rather does their own asynchronous event processing. This variable is read-only (do not pass its address to **select()**!), yet it may change after calls to **svc_getreqset()** or any creation routines.

bool_t svc_freeargs(xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;

Free any data allocated by the RPC/XDR system when it decoded the arguments to a service procedure using **svc_getargs()**. This routine returns TRUE if the results were successfully freed, and FALSE otherwise.

bool_t svc_getargs(xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;

Decode the arguments of an RPC request associated with the RPC service transport handle, *xprt*. The parameter *in* is the address where the arguments will be placed; *inproc* is the XDR routine used to decode the arguments. This routine returns TRUE if decoding succeeds, and FALSE otherwise.

struct sockaddr_in * svc_getcaller(xprt)
SVCXPRT *xprt;

The approved way of getting the network address of the caller of a procedure associated with the RPC service transport handle, *xprt*.

void svc_getreq(rdfds)
int rdfds;

Similar to **svc_getreqset()**, but limited to 32 descriptors. This interface is obsoleted by **svc_getreqset()**.

void svc_getreqset(rdfdsp)
fd_set *rdfdsp;

This routine is only of interest if a service implementor does not use **svc_run()**, but instead implements custom asynchronous event processing. It is called when the **select()** system call has determined that an RPC request has arrived on some RPC **socket(s)** ; *rdfdsp* is the resultant read file descriptor bit mask. The routine returns when all sockets associated with the value of *rdfdsp* have been serviced.

void svc_run()

Normally, this routine only returns in the case of some errors. It waits for RPC requests to arrive, and calls the appropriate service procedure using **svc_getreq()** when one arrives. This procedure is usually waiting for a **select()** system call to return.

bool_t svc_sendreply(xprt, outproc, out)
SVCXPRT *xprt;
xdrproc_t outproc;
char *out;

Called by an RPC service's dispatch routine to send the results of a remote procedure call. The parameter *xprt* is the request's associated transport handle; *outproc* is the XDR routine which is used to encode the results; and *out* is the address of the results. This routine returns TRUE if it succeeds, FALSE otherwise.

SEE ALSO

select(2), **rpc(3N)**, **rpc_svc_calls(3N)**, **rpc_svc_create(3N)**, **rpc_svc_err(3N)**

NAME

xdr_accepted_reply, xdr_authunix_parms, xdr_callhdr, xdr_callmsg, xdr_opaque_auth, xdr_rejected_reply, xdr_replymsg – XDR library routines for remote procedure calls

DESCRIPTION

These routines are used for describing the RPC messages in XDR language. They should normally be used by those who do not want to use the RPC package.

Routines

The **XDR** data structure is defined in the RPC/XDR Library Definitions of the *Network Programming*.

#include <rpc/rpc.h>

```
bool_t xdr_accepted_reply(xdrs, arp)
XDR *xdrs;
struct accepted_reply *arp;
```

Used for encoding RPC reply messages. It encodes the status of the RPC call in the XDR language format and in the case of success, it encodes the call results as well. This routine is useful for users who wish to generate RPC-style messages without using the RPC package. This routine returns TRUE if it succeeds, FALSE otherwise.

```
bool_t xdr_authunix_parms(xdrs, aup)
XDR *xdrs;
struct authunix_parms *aup;
```

Used for describing UNIX credentials. It encludes machine name, user ID, group ID list, etc. This routine is useful for users who wish to generate these credentials without using the RPC authentication package. This routine returns TRUE if it succeeds, FALSE otherwise.

```
void xdr_callhdr(xdrs, chdrp)
XDR *xdrs;
struct rpc_msg *chdrp;
```

Used for describing RPC call header messages. It encodes the static part of the call message header in the XDR language format. It includes information such as transaction ID, RPC version number, program number, and version number. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

```
bool_t xdr_callmsg(xdrs, cmsgp)
XDR *xdrs;
struct rpc_msg *cmsgp;
```

Used for describing RPC call messages. It includes all the RPC call information such as transaction ID, RPC version number, program number, version number, authentication information, etc. This routine is useful for users who wish to generate RPC-style messages without using the RPC package. This routine returns TRUE if it succeeds, FALSE otherwise.

```
bool_t xdr_opaque_auth(xdrs, ap)
XDR *xdrs;
struct opaque_auth *ap;
```

Used for describing RPC authentication information messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package. This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_rejected_reply(xdrs, rrp)
XDR *xdrs;
struct rejected_reply *rrp;

Used for describing RPC reply messages. It encodes the rejected RPC message in the XDR language format. The message is rejected either because of version number mismatch or because of authentication errors. This routine is useful for users who wish to generate RPC-style messages without using the RPC package. This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_replymsg(xdrs, rmsgp)
XDR *xdrs;
struct rpc_msg *rmsgp;

Used for describing RPC reply messages. It encodes the RPC reply message in the XDR language format. This reply could be an acceptance, rejection, or NULL. This routine is useful for users who wish to generate RPC style messages without using the RPC package. This routine returns TRUE if it succeeds, FALSE otherwise.

SEE ALSO

rpc(3N)

NAME

rtime – get remote time

SYNOPSIS

```
#include <sys/types.h>
#include <sys/time.h>
#include <netinet/in.h>

int rtime(addrp, timep, timeout)
struct sockaddr_in *addrp;
struct timeval *timep;
struct timeval *timeout;
```

DESCRIPTION

rtime( ) consults the Internet Time Server at the address pointed to by *addrp* and returns the remote time in the **timeval** struct pointed to by *timep*. Normally, the UDP protocol is used when consulting the Time Server. The *timeout* parameter specifies how long the routine should wait before giving up when waiting for a reply. If *timeout* is specified as NULL, however, the routine will instead use TCP and block until a reply is received from the time server.

The routine returns 0 if it is successful. Otherwise, it returns –1 and **errno** is set to reflect the cause of the error.

NAME

scandir, alphasort – scan a directory

SYNOPSIS

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

scandir(dirname, &namelist, select, compar)
char *dirname;
struct direct **namelist;
int (*select)( );
int (*compar)( );

alphasort(d1, d2)
struct direct **d1, **d2;
```

DESCRIPTION

scandir( ) reads the directory **dirname** and builds an array of pointers to directory entries using **malloc**(3V). The second parameter is a pointer to an array of structure pointers. The third parameter is a pointer to a routine which is called with a pointer to a directory entry and should return a non zero value if the directory entry should be included in the array. If this pointer is NULL, then all the directory entries will be included. The last argument is a pointer to a routine which is passed to **qsort**(3) to sort the completed array. If this pointer is NULL, the array is not sorted. **alphasort( )** is a routine which will sort the array alphabetically.

scandir( ) returns the number of entries in the array and a pointer to the array through the parameter *namelist.*

SEE ALSO

directory(3V), **malloc**(3V), **qsort**(3)

DIAGNOSTICS

Returns –1 if the directory cannot be opened for reading or if **malloc**(3V) cannot allocate enough memory to hold all the data structures.

NAME

scanf, fscanf, sscanf – formatted input conversion

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;
```

SYSTEM V SYNOPSIS

The following are provided for XPG2 compatibility:

```
#define nl_scanfscanf
#define nl_fscanf       fscanf
#define nl_sscanf       sscanf
```

DESCRIPTION

scanf( ) reads from the standard input stream **stdin**. **fscanf**( ) reads from the named input stream. **sscanf**( ) reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string *format*, described below, and a set of *pointer* arguments indicating where the converted input should be stored. The results are undefined in there are insufficient *args* for the format. If the format is exhausted while *args* remain, the excess *args* are simply ignored.

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

- White-space characters (SPACE, TAB, or NEWLINE) which, except in two cases described below, cause input to be read up to the next non-white-space character.
- An ordinary character (not '%'), which must match the next character of the input stream.
- Conversion specifications, consisting of the character '%' or the character sequence %*digit*$, 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.

Conversion specifications are introduced by the character % or the character sequence %*digit*$. 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 character 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 characters 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 as described for **string_to_decimal**(3), with *fortran_conventions* zero.

s A character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating \0, which will be added automatically. The input field is terminated by a white space character.

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

[Indicates string data; the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, which we will call the *scanset*, and a right bracket; the input field is the maximal sequence of input characters consisting entirely of characters in the scanset. The circumflex (ˆ), when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters *not* contained in the remainder of the scanset string. There are some conventions used in the construction of the scanset. A range of characters may be represented by the construct *first–last*, thus [0123456789] may be expressed [0–9]. Using this convention, *first* must be lexically less than or equal to *last*, or else the dash will stand for itself. The dash will also stand for itself whenever it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset, and in this case it will not be syntactically interpreted as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating \0, which will be added automatically. 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.

Avoid this common error: because **printf**(3V) does not require that the lengths of conversion descriptors and actual parameters match, coders sometimes are careless with the **scanf**( ) functions. But converting %f to &double or %lf to &float *does not work*; the results are quite incorrect.

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. The constant EOF is returned upon end of input. Note: this is different from 0, which means that no conversion was done; if conversion was intended, it was frustrated by an inappropriate character in the input.

If the input ends before the first conflict or conversion, EOF is returned. If the input ends after the first conflict or conversion, the number of successfully matched items is returned.

Conversions can be applied to the *n*th argument in the argument list, rather than the next unused argument. In this case, the conversion character % (see below) is replaced by the sequence %digit$, where *digit* is a decimal integer *n* in the range [1,9], giving the position of the argument in the argument list. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages.

The format string can contain either form of a conversion specification, that is % or %*digit*$, although the two forms cannot be mixed within a single format string.

All forms of the scanf() functions allow for the detection of a language dependent radix character in the input string. The radix character is defined by the program's locale (category **LC_NUMERIC**). In the "C" locale, or in a locale where the radix character is not defined, the radix character defaults to '.'.

SYSTEM V DESCRIPTION

FORMFEED is allowed as a white space character in control strings.

XPG2 requires that **nl_scanf**, **nl_fscanf** and **nl_sscanf** be defined as **scanf**, **fscanf** and **sscanf**, respectively for backward compatibility.

RETURN VALUES

If any items are converted, scanf(), fscanf() and sscanf() return the number of items converted successfully. This number may smaller than the number of items requested. If no items are converted, these functions return 0. scanf(), fscanf() and sscanf() return EOF on end of input.

EXAMPLES

The call:

```
int i, n; float x; char name[50];
n = scanf("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign to *n* the value 3, to *i* the value 25, to *x* the value 5.432, and *name* will contain **thompson\0**. Or:

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

with input:

```
011 56789 0123 56a72
```

will assign **9** to *j*, **56** to *i*, **789.0** to *x*, skip **0123**, and place the string **56\0** in *name*. The next call to getchar() (see getc(3V)) will return a. Or:

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

with input:

```
0x11 0xy johnson
```

will assign **17** to *i*, **0** to *j*, **6** to *s*, will place the string **xy\0** in *name*, and will assign **8** to *e*. Thus, the length of *name* is *e* − *s* = 2. The next call to getchar() (see getc(3V)) will return a SPACE.

SEE ALSO

getc(3V), **printf**(3V), setlocale(3V), stdio(3V), **string_to_decimal**(3), **strtol**(3)

WARNINGS

Trailing white space (including a NEWLINE) is left unread unless matched in the control string.

BUGS

The success of literal matches and suppressed assignments is not directly determinable.

NAME

authdes_create, authdes_getucred, get_myaddress, getnetname, host2netname, key_decryptsession, key_encryptsession, key_gendes, key_setsecret, netname2host, netname2user, user2netname – library routines for secure remote procedure calls

DESCRIPTION

RPC routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a request to the server. Upon receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends back a reply. Finally, the procedure call returns to the client.

RPC allows various authentication flavors The **authdes_getucred()** and **authdes_create()** routines implement the DES authentication flavor. See **rpc_clnt_auth**(3N) for routines relating to the AUTH_NONE and AUTH_UNIX authentication types.

> Note: Both the client and server should have their keys in the **publickey**(5) database. Also, the keyserver daemon **keyserv**(8C) must be running on both the client and server hosts for the DES authentication system to work.

Routines

```
#include <rpc/rpc.h>
```

```
AUTH * authdes_create(netname, window, syncaddr, deskeyp)
char *netname;
unsigned window;
struct sockaddr_in *syncaddr;
des_block *deskeyp;
```

> **authdes_create()** is an interface to the RPC secure authentication system, known as DES authentication.
>
> Used on the client side, **authdes_create()** returns an authentication handle that enables the use of the secure authentication system. The first parameter *netname* is the network name of the owner of the server process. This field usually represents a *host* derived from the utility routine **host2netname()**, but could also represent a user name using **user2netname()**. The second field is window on the validity of the client credential, given in seconds. A small window is more secure than a large one, but choosing too small of a window will increase the frequency of resynchronizations because of clock drift. The third parameter *syncaddr* is optional. If it is NULL, then the authentication system will assume that the local clock is always in sync with the server's clock, and will not attempt to synchronize with the server. If an address is supplied then the system will use it for consulting the remote time service whenever resynchronization is required. This parameter is usually the address of the RPC server itself. The final parameter *deskeyp* is also optional. If it is NULL, then the authentication system will generate a random DES key to be used for the encryption of credentials. If *deskeyp* is supplied then it is used instead.

```
int authdes_getucred(adc, uidp, gidp, gidlenp, gidlistp)
struct authdes_cred *adc;
short *uidp;
short *gidp;
short *gidlenp;
int *gidlistp;
```

> **authdes_getucred()**, is a DES authentication routine used by the server for converting a DES credential, which is operating system independent, into a UNIX credential. *uidp* points to the user ID of the user associated with *adc*; *gidp* refers to the user's current group ID; *gidlistp* refers to an array of groups to which the user belongs and *gidlenp* has the count of the entries in this array.

This routine differs from the utility routine **netname2user()** in that **authdes_getucred()** pulls its information from a cache, and does not have to do a NIS name service lookup every time it is called to get its information. Returns 1 if it succeeds and 0 if it fails.

void get_myaddress(addr)
struct sockaddr_in *addr;

Return the machine's IP address in *addr*. The port number is always set to **htons(PMAPPORT)**.

int getnetname(netname)
char netname[MAXNETNAMELEN];

Return the unique, operating-system independent netname of the caller in the fixed-length array *netname*. Returns 1 if it succeeds and 0 if it fails.

int host2netname(netname, host, domain)
char netname[MAXNETNAMELEN];
char *host;
char *domain;

Convert from a domain-specific hostname to an operating-system independent netname. This routine is normally used to get the netname of the server, which is then used to get an authentication handle by calling **authdes_create()**. This routine should be used if the owner of the server process is the machine that is, the user with effective user ID zero. Returns 1 if it succeeds and 0 if it fails. This routine is the inverse of **netname2host()**.

int key_decryptsession(netname, deskeyp)
char *netname;
des_block *deskeyp;

An interface routine to the keyserver daemon, which is associated with RPC's secure authentication system (DES authentication). User programs rarely need to call it, or its associated routines **key_encryptsession()**, **key_gendes()** and **key_setsecret()**. System commands such as **login** and the RPC library are the main clients of these four routines.

key_decryptsession() takes the netname of a server and a DES key, and decrypts the key by using the public key of the server and the secret key associated with the effective user ID of the calling process. Returns 0 if it succeeds and −1 if it fails. This routine is the inverse of **key_encryptsession()**.

int key_encryptsession(netname, deskeyp)
char *netname;
des_block *deskeyp;

A keyserver interface routine. It takes the netname of the server and a des key, and encrypts it using the public key of the server and the secret key associated with the effective user ID of the calling process. Returns 0 if it succeeds and −1 if it fails. This routine is the inverse of **key_decryptsession()**.

int key_gendes(deskeyp)
des_block *deskeyp;

A keyserver interface routine. It is used to ask the keyserver for a secure conversation key. Choosing one at "random" is usually not good enough, because the common ways of choosing random numbers, such as using the current time, are very easy to guess. Returns 0 if it succeeds and −1 if it fails.

int key_setsecret(keyp)
char *keyp;

A keyserver interface routine. It is used to set the secret key for the effective user ID of the calling process. Returns 0 if it succeeds and −1 if it fails.

int netname2host(netname, host, hostlen)
char *netname;
char *host;
int hostlen;

Convert an operating-system independent netname to a domain-specific hostname. *hostlen* specifies the size of the array pointed to by *host*. It returns 1 if it succeeds and 0 if it fails. This routine is the inverse of **host2netname( )**.

int netname2user(netname, uidp, gidp, gidlenp, gidlistp)
char *name;
int *uidp;
int *gidp;
int *gidlenp;
int *gidlistp;

Convert an operating-system independent netname to a domain-specific user ID. *uidp* points to the user ID of the user; *gidp* refers to the user's current group ID; *gidlistp* refers to an array of groups to which the user belongs and *gidlenp* has the count of the entries in this array. It returns 1 if it succeeds and 0 if it fails. This routine is the inverse of **user2netname( )**.

int user2netname(netname, uid, domain)
char name[MAXNETNAMELEN];
int uid;
char *domain;

Convert a domain-specific username to an operating-system independent netname. *uid* is the user ID of the owner of the server process. This routine is normally used to get the netname of the server, which is then used to get an authentication handle by calling **authdes_create( )**. Returns 1 if it succeeds and 0 if it fails. This routine is the inverse of **netname2user( )**.

SEE ALSO

login(1), **chkey**(1), **rpc**(3N), **rpc_clnt_auth**(3N), **publickey**(5), **keyserv**(8C), **newkey**(8)

NAME

setbuf, setbuffer, setlinebuf, setvbuf – assign buffering to a stream

SYNOPSIS

```
#include <stdio.h>

void setbuf(stream, buf)
FILE *stream;
char *buf;

void setbuffer(stream, buf, size)
FILE *stream;
char *buf;
int size;

int setlinebuf(stream) FILE *stream;

int setvbuf(stream, buf, type, size)
FILE *stream;
char *buf;
int type, size;
```

DESCRIPTION

The three types of buffering available are unbuffered, block buffered, and line buffered. When an output stream is unbuffered, information appears on the destination file or terminal as soon as written; when it is block buffered many characters are saved up and written as a block; when it is line buffered characters are saved up until a NEWLINE is encountered or input is read from **stdin**. **fflush**() (see **fclose**(3V)) may be used to force the block out early. A buffer is obtained from **malloc**(3V) upon the first **getc**(3V) or **putc**(3S) on the file. By default, output to a terminal is line buffered, except for output to the standard stream **stderr** which is unbuffered. All other input/output is fully buffered.

setbuf( ) can be used after a stream has been opened but before it is read or written. It causes the array pointed to by *buf* to be used instead of an automatically allocated buffer. If *buf* is the NULL pointer, input/output will be completely unbuffered. A manifest constant **BUFSIZ**, defined in the **<stdio.h>** header file, tells how big an array is needed:

```
char buf[BUFSIZ];
```

setbuffer( ), an alternate form of **setbuf**( ), can be used after a stream has been opened but before it is read or written. It uses the character array *buf* whose size is determined by the *size* argument instead of an automatically allocated buffer. If *buf* is the NULL pointer, input/output will be completely unbuffered.

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

_IOFBF fully buffers the input/output.

_IOLBF line buffers the output; the buffer will be flushed when a NEWLINE is written, the buffer is full, or input is requested.

_IONBF completely unbuffers the input/output.

If *buf* is not the NULL pointer, the array it points to will be used for buffering, instead of an automatically allocated buffer. *size* specifies the size of the buffer to be used.

setlinebuf( ) is used to change the buffering on a stream from block buffered or unbuffered to line buffered. Unlike **setbuf**( ), **setbuffer**( ), and **setvbuf**( ), it can be used at any time that the file descriptor is active.

A file can be changed from unbuffered or line buffered to block buffered by using **freopen()** (see **fopen**(3V)). A file can be changed from block buffered or line buffered to unbuffered by using **freopen()** followed by **setbuf()** with a buffer argument of NULL.

SYSTEM V DESCRIPTION

If *buf* is not NULL and *stream* refers to a terminal device, **setbuf()** sets *stream* for line buffered input/output.

RETURN VALUES

setlinebuf() returns no useful value.

setvbuf() returns 0 on success. If an illegal value for *type* or *size* is provided, **setvbuf()** returns a non-zero value. **setvbuf()**

SEE ALSO

fclose(3V), **fopen**(3V), **fread**(3S), **getc**(3V), **malloc**(3V), **printf**(3V), **putc**(3S), **puts**(3S)

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, sigsetjmp, siglongjmp – non-local goto

SYNOPSIS

```
#include <setjmp.h>

int setjmp(env)
jmp_buf env;

void longjmp(env, val)
jmp_buf env;
int val;

int _setjmp(env)
jmp_buf env;

void _longjmp(env, val)
jmp_buf env;
int val;

int sigsetjmp(env, savemask)
sigjmp_buf env;
int savemask;

void siglongjmp(env, val)
sigjmp_buf env;
int val;
```

DESCRIPTION

setjmp() and **longjmp()** are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

The macro **setjmp()** saves its stack environment in *env* for later use by **longjmp()**. A normal call to **setjmp()** returns zero. **setjmp()** also saves the register environment. If a **longjmp()** call will be made, the routine which called **setjmp()** should not return until after the **longjmp()** has returned control (see below).

longjmp() restores the environment saved by the last call of **setjmp**, and then returns in such a way that execution continues as if the call of **setjmp()** had just returned the value *val* to the function that invoked **setjmp()**; however, if *val* were zero, execution would continue as if the call of **setjmp()** had returned one. This ensures that a "return" from **setjmp()** caused by a call to **longjmp()** can be distinguished from a regular return from **setjmp()**. The calling function must not itself have returned in the interim, otherwise **longjmp()** will be returning control to a possibly non-existent environment. All memory-bound data have values as of the time **longjmp()** was called. The CPU and floating-point data registers are restored to the values they had at the time that **setjmp()** was called. But, because the **register** storage class is only a hint to the C compiler, variables declared as **register** variables may not necessarily be assigned to machine registers, so their values are unpredictable after a **longjmp()**. This is especially a problem for programmers trying to write machine-independent C routines.

setjmp() and **longjmp()** save and restore the signal mask (see **sigsetmask**(2)), while **_setjmp()** and **_longjmp()** manipulate only the C stack and registers. If the *savemask* flag to **sigsetjmp()** is non-zero, the signal mask is saved, and a subsequent **siglongjmp()** using the same *env* will restore the signal mask. If the *savemask* flag is zero, the signal mask is not saved, and a subsequent **siglongjmp()** using the same *env* will not restore the signal mask. In all other ways, **_setjmp()** and **sigsetjmp()** function in the same way that **setjmp()** does, and **_longjmp()** and **siglongjmp()** function in the same way that **longjmp()** does.

None of these functions save or restore any floating-point status or control registers, in particular the MC68881 **fpsr**, **fpcr**, or **fpiar**, the Sun-3 FPA **fpamode** or **fpastatus**, and the Sun-4 **%fsr**. See **ieee_flags**(3M) to save and restore floating-point status or control information.

SYSTEM V DESCRIPTION

setjmp() and **longjmp()** manipulate only the C stack and registers; they do not save or restore the signal mask. **_setjmp()** behaves identically to **setjmp()**, and **_longjmp()** behaves identically to **longjmp()**.

EXAMPLE

The following code fragment indicates the flow of control of the **setjmp()** and **longjmp()** combination:

```
function declaration
...
        jmp_buf         my_environment;
        ...
        if (setjmp(my_environment)) {
                /* register variables have unpredictable values */
                code after the return from longjmp
                ...
        } else {
                /* do not modify register vars in this leg of code */
                this is the return from setjmp
                ...
        }
```

SEE ALSO

cc(1V), **sigsetmask**(2), **sigvec**(2), **ieee_flags**(3M), **signal**(3V), **setjmp**(3V)

BUGS

setjmp() does not save the current notion of whether the process is executing on the signal stack. The result is that a **longjmp()** to some place on the signal stack leaves the signal stack state incorrect.

On Sun-2 and Sun-3 systems **setjmp()** also saves the register environment. Therefore, all data that are bound to registers are restored to the values they had at the time that **setjmp()** was called. All memory-bound data have values as of the time **longjmp()** was called. However, because the **register** storage class is only a hint to the C compiler, variables declared as **register** variables may not necessarily be assigned to machine registers, so their values are unpredictable after a **longjmp()**. When using compiler options that specify automatic register allocation (see **cc**(1V)), the compiler will not attempt to assign variables to registers in routines that call **setjmp()**.

NAME

setlocale, nl_init – set international environment

SYNOPSIS

```
#include <locale.h>

char *setlocale(category, locale)
int category;
char *locale;

int nl_init(lang)
char *lang;
```

DESCRIPTION

setlocale() selects the appropriate piece of the program's locale as specified by *category*, and may be used to change or query the program's international environment. The entire locale may be changed by calling **setlocale()** with *category* set to **LC_ALL**. The other possible values for *category* query or change only a part of the program's complete international locale:

LC_CTYPE
Affects the behavior of the character classification and conversion functions. See **ctype**(3V), and **mblen**(3).

LC_COLLATE
Affects the behavior of the string collation functions **strcoll** (3) and **strxfrm**(3V).

LC_TIME
Affects the behavior of the time conversion functions. See **printf**(3V), **scanf**(3V), **strtod**(3), and **ctime**(3V) for **strftime()**, **strptime()**, and **ctime()**.

LC_NUMERIC
Affects the radix character for the formatted input/output functions and the string conversion functions, **gcvt**(3V), **printf**(3V), **strtod**(3), **gconvert()**, **sgconvert()** (see **econvert**(3)), **file_to_decimal()**, and **func_to_decimal()** (see **string_to_decimal**(3)). Also affects the non-monetary formatting information returned by the **localeconv()** function.

LC_MONETARY
Affects the monetary formatting information returned by the **localeconv()** function.

LC_MESSAGES
Affects the behavior of functions that present messages, namely **gettext()**, and **textdomain()**.

The *locale* argument is a pointer to a character string containing the required setting of *category*. The following preset values of *locale* are defined for all settings of *category*:

"C" Specifies the minimal environment for C translation. If **setlocale()** is not invoked, the **"C"** locale is the default. Operational behavior within the **"C"** locale is defined separately for each interface function.

At program startup, the equivalent of:

"" In this case, **setlocale()** will first check the value of the corresponding environment variable (for example, **LC_CTYPE** for the **LC_CTYPE** category) and if valid (that is, points to the name of a valid locale), **setlocale()** sets the specified category of the international environment to that value and returns the string corresponding to the locale set (that is, the value of the environment variable, not **""**). If the value is invalid, **setlocale()** returns a NULL pointer and the international environment is not changed by this call.

If the environment variable corresponding to the specified category is not set or is set to the empty string, **setlocale()** will examine the **LANG** environment variable. If both the **LANG** environment variable, and the environment variable corresponding to the specified category are not set or are set to the empty string, then the **LC_default** environment variable is examined. If this contains a valid setting, then the category is set to the value of **LC_default**. If the LANG environment variable is set and valid this will set the category to the corresponding value of LANG. If **LC_default** is not set, then **setlocale()** returns that category to the default "C" locale.

To set all categories in the international environment, **setlocale()** is invoked in the following manner:

setlocale (LC_ALL, "");

To satisfy this request, **setlocale()** first checks all the relevant environment variables **LC_CTYPE, LC_COLLATE, LC_TIME, LC_NUMERIC, LC_MONETARY, LC_MESSAGES.** If any one of these relevant environment variables is invalid, this call to **setlocale()** will return a NULL pointer, and the international environment will not be changed. If all the relevant environment variables are valid, **setlocale()** sets the international environment to reflect the values of the environment variables. The categories are set in the following order:

LC_CTYPE
LC_COLLATE
LC_TIME
LC_NUMERIC
LC_MONETARY
LC_MESSAGES

Using this scheme, the categories corresponding to the environment variables will override the value of the **LANG** and **LC_default** environment variables for a particular category.

nl_init() is equivalent to

setlocale(LC_ALL, "");

and is supplied for compatibility with X/Open XPG2.

RETURN VALUES

If a valid string is given for the *locale* parameter, and the selection can be honored, **setlocale()** returns the string associated with the specified *category* for the new locale. If the selection cannot be honored, **setlocale()** returns a null pointer and the program's locale is not changed.

A NULL pointer for *locale* causes **setlocale()** to return the string associated with the *category* for the program's current locale; the program's locale is not changed. The string contains information relating to each piece part of the whole international environment. This inquiry can fail by returning a null pointer if any *category* is invalid.

The string returned by such a **setlocale()** call is such that a subsequent call with the string and its associated category will restore that part of the program's locale. The string returned by:

ptr = setlocale(LC_ALL, (char *) 0);

is such that in a subsequent call:

setlocale(LC_ALL, ptr);

will reset each and every category to the state when the string was first returned. The string returned must not be modified by the program, but will be overwritten by a subsequent call to **setlocale()**.

FILES

/etc/locale/*locale***/***category*

locale is the directory that contains numerous files (*categories*), each relating to a single category of a valid *locale* as selected by category argument to **setlocale()**. Generally this is classed as a private directory. This directory is searched by **setlocale()**, prior to searching:

/usr/share/lib/locale/*locale***/***category*

locale is the directory that contains numerous files (*categories*), each relating to a single category of a valid *locale* as selected by category argument to **setlocale()**. Generally this data is classed as global and sharable.

DIAGNOSTICS

setlocale( ) returns a null pointer if a relevant environment variable has an invalid setting. **setlocale( )** also returns a null pointer if *category* is invalid.

NAME

setuid, seteuid, setruid, setgid, setegid, setrgid – set user and group ID

SYNOPSIS

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

int setuid(uid)
uid_t uid;

int seteuid(euid)
uid_t euid;

int setruid(ruid)
uid_t ruid;

int setgid(gid)
gid_t gid;

int setegid(egid)
gid_t egid;

int setrgid(rgid)
gid_t rgid;
```

DESCRIPTION

setuid( ) (**setgid**( )) sets both the real and effective user ID (group ID) of the current process as specified by *uid* (*gid*) (see NOTES).

seteuid( ) (**setegid**( )) sets the effective user ID (group ID) of the current process.

setruid( ) (**setrgid**( )) sets the real user ID (group ID) of the current process.

These calls are only permitted to the super-user or if the argument is the real or effective user (group) ID of the calling process.

SYSTEM V DESCRIPTION

If the effective user ID of the calling process is not super-user, but if its real user (group) ID is equal to *uid* (*gid*), or if the saved set-user (group) ID from **execve**(2V) is equal to *uid* (*gid*), then the effective user (group) ID is set to *uid* (*gid*).

RETURN VALUES

These functions return:

0	on success.
−1	on failure and set **errno** to indicate the error as for **setreuid**(2) (**setregid**(2)).

ERRORS

EINVAL	The value of *uid* (*gid*) is invalid (less than 0 or greater than 65535).
EPERM	The process does not have super-user privileges and *uid* (*gid*) does not matches neither the real user (group) ID of the process nor the saved set-user-ID (set-group-ID) of the process.

SEE ALSO

execve(2V), **getgid**(2V), **getuid**(2V), **setregid**(2), **setreuid**(2)

NOTES

For **setuid**( ) to behave as described above, {_POSIX_SAVED_IDS} must be in effect (see **sysconf**(2V)). {_POSIX_SAVED_IDS} is always in effect on SunOS systems, but for portability, applications should call **sysconf**( ) to determine whether {_POSIX_SAVED_IDS} is in effect for the current system.

NAME

sigaction – examine and change signal action

SYNOPSIS

```
#include <signal.h>

int sigaction(sig, act, oact)
int sig;
struct sigaction *act, *oact;
```

DESCRIPTION

sigaction( ) allows the calling process to examine and specify (or both) the action to be associated with a specific signal. *sig* specifies the signal. Acceptable values are defined in **<signal.h>**.

The structure **sigaction**( ), used to describe an action to be taken, is defined in the header **<signal.h>** as follows:

```
struct sigaction {
        void (*sa_handler)();    /* SIG_DFL, SIG_IGN, or pointer to a function */
        sigset_t sa_mask;        /* Additional signals to be blocked during
                                    execution of signal-catching function */
        int sa_flags;            /* Special flags to affect behavior of signal */
};
```

If **act** is not **NULL**, it points to a structure specifying the action to be associated with the specified signal. If **oact** is not **NULL**, the action previously associated with the signal is stored in the location pointed to by the **oact.** If **act** is NULL, signal handling is unchanged by this function. Thus, the call can be used to enquire about the current handling of a given signal. The **sa_handler** field of the **sigaction** structure identifies the action to be associated with the specified signal. If the **sa_handler** field specifies a signal-catching function, the **sa_mask** field identifies a set of signals that shall be added to the process's signal mask before the signal-catching function mask is invoked. The **SIGKILL** and **SIGSTOP** signals shall not be added to the signal mask using this mechanism; this restriction shall be enforced by the system without causing an error to be indicated.

The **sa_flags** field can be used to modify the behavior of the specified signal. The following flag bit, defined in the header **<signal.h>**, can be set in **sa_flags**:

```
#define SA_ONSTACK          0x0001 /* take signal on signal stack */
#define SA_INTERRUPT        0x0002 /* do not restart system on signal return */
#define SA_RESETHAND        0x0004 /* reset handler to SIG_DFL when signal taken */
#define SA_NOCLDSTOP        0x0008 /* don't send a SIGCHLD on child stop */
```

If *sig* is **SIGCHILD** and the **SA_NOCLDSTOP** flag is not set in **sa_flags**, and the implementation supports the **SIGCHILD** signal, a **SIGCHILD** signal shall be generated for the calling process whenever any of its child processes stop. If *sig* is **SIGCHILD** and the **SA_NOCLDSTOP** flag is set in **sa_flags**, the implementation shall not generate a **SIGCHILD** signal in this way.

If the **SA_ONSTACK** bit is set in the flags for that signal, the system will deliver the signal to the process on the signal stack specified with **sigstack**(2), rather than delivering the signal on the current stack.

If a caught signal occurs during certain system calls, the call is restarted by default. The call can be forced to terminate prematurely with an EINTR error return by setting the **SA_INTERRUPT** bit in the flags for that signal. **SA_INTERRUPT** is not available in 4.2BSD, hence it should not be used if backward compatibility is needed. The affected system calls are **read**(2V) or **write**(2V) on a slow device (such as a terminal or pipe or other socket, but not a file) and during a **wait**(2V).

Once a signal handler is installed, it remains installed until another **sigvec**( ) call is made, or an **execve**(2V) is performed, unless the **SA_RESETHAND** bit is set in the flags for that signal. In that case, the value of the handler for the caught signal is set to **SIG_DFL** before entering the signal-catching function, unless the signal is **SIGILL** or **SIGTRAP**. Also, if this bit is set, the bit for that signal in the signal mask will not be set; unless the signal mask associated with that signal blocks that signal, further occurrences of that signal will not be blocked. The **SA_RESETHAND** flag is not available in 4.2BSD, hence it should not be used if backward compatibility is needed.

When a signal is caught by a signal-catching function installed by **sigaction()** a new signal mask is calculated and installed for the duration of the signal-catching function (or until a call to either **sigprocmask()** or **sigsuspend()**). This mask is formed by taking the union of the current signal mask and the value of the sa_**mask** for the signal being delivered, and then including the signal being delivered. If and when the user's signal handler returns normally, the original signal mask is restored.

Once an action is installed for a specific signal, it remains installed until another action is explicitly requested (by another call to **sigaction()**), or until one of the **exec** functions is called.

If the previous action for *sig* had been established by **signal()** defined in the C standard, the values of the fields returned in the structure pointed to by the **oact** are unspecified, and in particular **oact–>sv_handler** is not necessarily the same value passed to **signal()**. However, if a pointer to the same structure or a copy thereof is passed to a subsequent call to **sigaction()** using **act**, handling of the signal shall be as if the original call to **signal()** were repeated.

If **sigaction()** fails, no new signal handler is installed.

RETURN VALUES

sigaction() returns:

0 on success.

–1 on failure and sets **errno** to indicate the error.

ERRORS

EINVAL *sig* is an invalid or unsupported signal number.

An attempt was made to catch a signal that cannot be ignored. See **<signal.h>**.

SEE ALSO

kill(2V), **sigpause(2V)**, **sigprocmask(2V)**, **signal(3V)**, **sigsetops(3V)**

NAME

sigfpe – signal handling for specific SIGFPE codes

SYNOPSIS

#include <signal.h>

#include <floatingpoint.h>

sigfpe_handler_type sigfpe(code, hdl)
sigfpe_code_type code;
sigfpe_handler_type hdl;

DESCRIPTION

This function allows signal handling to be specified for particular **SIGFPE** codes. A call to **sigfpe**( ) defines a new handler *hdl* for a particular **SIGFPE** *code* and returns the old handler as the value of the function **sigfpe**( ) . Normally handlers are specified as pointers to functions; the special cases **SIGFPE_IGNORE**, **SIGFPE_ABORT**, and **SIGFPE_DEFAULT** allow ignoring, specifying core dump using **abort**(3), or default handling respectively.

For these IEEE-related codes:

FPE_FLTINEX_TRAP	fp_inexact - floating inexact result
FPE_FLTDIV_TRAP	fp_division - floating division by zero
FPE_FLTUND_TRAP	fp_underflow - floating underflow
FPE_FLTOVF_TRAP	fp_overflow - floating overflow
FPE_FLTBSUN_TRAP	fp_invalid - branch or set on unordered
FPE_FLTOPERR_TRAP	fp_invalid - floating operand error
FPE_FLTNAN_TRAP	fp_invalid - floating Not-A-Number

default handling is defined to be to call the handler specified to **ieee_handler**(3M).

For all other **SIGFPE** codes, default handling is to core dump using **abort**(3).

The compilation option **–ffpa** causes fpa recomputation to replace the default abort action for code **FPE_FPA_ERROR**. Note: **SIGFPE_DEFAULT** will restore abort rather than FPA recomputation for this code.

Three steps are required to intercept an IEEE-related **SIGFPE** code with **sigfpe**( ):

1) Set up a handler with **sigfpe**( ).

2) Enable the relevant IEEE trapping capability in the hardware, perhaps by using assembly-language instructions.

3) Perform a floating-point operation that generates the intended IEEE exception.

Unlike **ieee_handler**(3M), **sigfpe**( ) never changes floating-point hardware mode bits affecting IEEE trapping. No IEEE-related **SIGFPE** signals will be generated unless those hardware mode bits are enabled.

SIGFPE signals can be handled using **sigvec**(2), **signal**(3V), **sigfpe**(3), or **ieee_handler**(3M). In a particular program, to avoid confusion, use only one of these interfaces to handle **SIGFPE** signals.

EXAMPLE

A user-specified signal handler might look like this:

```
void sample_handler( sig, code, scp, addr )
        int sig ;           /* sig == SIGFPE always */
        int code ;
        struct sigcontext *scp ;
        char *addr ;
        {
                /*
                   Sample user-written sigfpe code handler.
                   Prints a message and continues.
                   struct sigcontext is defined in <signal.h>.
                 */
                printf(" ieee exception code %x occurred at pc %X \n",code,scp->sc_pc);
        }
```

and it might be set up like this:

```
        extern void sample_handler();
        main()
        {
                sigfpe_handler_type hdl, old_handler1, old_handler2;
        /*
         * save current overflow and invalid handlers; set the new
         * overflow handler to sample_handler() and set the new
         * invalid handler to SIGFPE_ABORT (abort on invalid)
         */
                hdl = (sigfpe_handler_type) sample_handler;
                old_handler1 = sigfpe(FPE_FLTOVF_TRAP, hdl);
                old_handler2 = sigfpe(FPE_FLTOPERR_TRAP, SIGFPE_ABORT);
                ...
        /*
         * restore old overflow and invalid handlers
         */
                sigfpe(FPE_FLTOVF_TRAP,   old_handler1);
                sigfpe(FPE_FLTOPERR_TRAP, old_handler2);
        }
```

SEE ALSO

sigvec(2), abort(3), floatingpoint(3), ieee_handler(3M), signal(3V)

DIAGNOSTICS

sigfpe() returns BADSIG if *code* is not zero or a defined SIGFPE code.

NAME

siginterrupt – allow signals to interrupt system calls

SYNOPSIS

int siginterrupt(sig, flag)
int sig, flag;

DESCRIPTION

siginterrupt() is used to change the system call restart behavior when a system call is interrupted by the specified signal. If the flag is false (0), then system calls will be restarted if they are interrupted by the specified signal and no data has been transferred yet. System call restart is the default behavior on 4.2BSD, and on SunOS in the 4.2 environment, when the **signal** (3V) routine is used.

If the flag is true (1), then restarting of system calls is disabled. If a system call is interrupted by the specified signal and no data has been transferred, the system call will return −1 with **errno** set to EINTR. Interrupted system calls that have started transferring data will return the amount of data actually transferred. System call interrupt is the signal behavior found on older version of the UNIX operating systems, such as 4.1BSD and System V UNIX. It is the default behavior on SunOS in the System V environment when the **signal()** routine is used; therefore, this routine is useful in that environment only if a signal that a **sigvec**(2) specified should restart system calls is to be changed not to restart them.

Note: the new 4.2BSD signal handling semantics are not altered in any other way. Most notably, signal handlers always remain installed until explicitly changed by a subsequent **sigvec()** call, and the signal mask operates as documented in **sigvec()**, unless the **SV_RESETHAND** bit has been used to specify that the pre-4.2BSD signal behavior is to be used. Programs may switch between restartable and interruptible system call operation as often as desired in the execution of a program.

Issuing a **siginterrupt()** call during the execution of a signal handler will cause the new action to take place on the next signal to be caught.

NOTES

This library routine uses an extension of the **sigvec**(2) system call that is not available in 4.2BSD, hence it should not be used if backward compatibility is needed.

RETURN VALUES

siginterrupt() returns:

0	on success.
−1	if an invalid signal number was supplied.

SEE ALSO

sigblock(2), **sigpause**(2V), **sigsetmask**(2), **sigvec**(2), **signal**(3V)

NAME

signal – simplified software signal facilities

SYNOPSIS

```
#include <signal.h>

void (*signal(sig, func))()
void (*func)();
```

DESCRIPTION

signal() is a simplified interface to the more general **sigvec**(2) facility. Programs that use **signal**() in preference to **sigvec**() are more likely to be portable to all systems.

A signal is generated by some abnormal event, initiated by a user at a terminal (quit, interrupt, stop), by a program error (bus error, etc.), by request of another program (kill), or when a process is stopped because it wishes to access its control terminal while in the background (see **termio**(4)). Signals are optionally generated when a process resumes after being stopped, when the status of child processes changes, or when input is ready at the control terminal. Most signals cause termination of the receiving process if no action is taken; some signals instead cause the process receiving them to be stopped, or are simply discarded if the process has not requested otherwise. Except for the **SIGKILL** and **SIGSTOP** signals, the **signal**() call allows signals either to be ignored or to interrupt to a specified location. The following is a list of all signals with names as in the include file **<signal.h>**:

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGILL	4*	illegal instruction
SIGTRAP	5*	trace trap
SIGABRT	6*	abort (generated by **abort**(3) routine)
SIGEMT	7*	emulator trap
SIGFPE	8*	arithmetic exception
SIGKILL	9	kill (cannot be caught, blocked, or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe or other socket with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGURG	16•	urgent condition present on socket
SIGSTOP	17†	stop (cannot be caught, blocked, or ignored)
SIGTSTP	18†	stop signal generated from keyboard
SIGCONT	19•	continue after stop
SIGCHLD	20•	child status has changed
SIGTTIN	21†	background read attempted from control terminal
SIGTTOU	22†	background write attempted to control terminal
SIGIO	23•	I/O is possible on a descriptor (see **fcntl**(2V))
SIGXCPU	24	cpu time limit exceeded (see **getrlimit**(2))
SIGXFSZ	25	file size limit exceeded (see **getrlimit**(2))
SIGVTALRM	26	virtual time alarm (see **getitimer**(2))
SIGPROF	27	profiling timer alarm (see **getitimer**(2))
SIGWINCH	28•	window changed (see **termio**(4) and **win**(4S))
SIGLOST	29*	resource lost (see **lockd**(8C))
SIGUSR1	30	user-defined signal 1
SIGUSR2	31	user-defined signal 2

The starred signals in the list above cause a core image if not caught or ignored.

If *func* is **SIG_DFL**, the default action for signal *sig* is reinstated; this default is termination (with a core image for starred signals) except for signals marked with • or †. Signals marked with • are discarded if the action is **SIG_DFL**; signals marked with † cause the process to stop. If *func* is **SIG_IGN** the signal is subsequently ignored and pending instances of the signal are discarded. Otherwise, when the signal occurs further occurrences of the signal are automatically blocked and *func* is called.

A return from the function unblocks the handled signal and continues the process at the point it was interrupted. **Unlike previous signal facilities, the handler *func* remains installed after a signal has been delivered.**

If a caught signal occurs during certain system calls, terminating the call prematurely, the call is automatically restarted. In particular this can occur during a **read**(2V) or **write**(2V) on a slow device (such as a terminal; but not a file) and during a **wait**(2V).

The value of **signal**() is the previous (or initial) value of *func* for the particular signal.

After a **fork**(2V) or **vfork**(2) the child inherits all signals. An **execve**(2V) resets all caught signals to the default action; ignored signals remain ignored.

SYSTEM V DESCRIPTION

If *func* is **SIG_IGN** the signal is subsequently ignored and pending instances of the signal are discarded. Otherwise, when the signal occurs, *func* is called. Further occurrences of the signal are not automatically blocked. The value of *func* for the caught signal is reset to **SIG_DFL** before *func* is called, unless the signal is **SIGILL** or **SIGTRAP**.

A return from the function continues the process at the point at which it was interrupted. The handler *func* does not remain installed after a signal has been delivered.

If a caught signal occurs during certain system calls, causing the call to terminate prematurely, the call is interrupted. In particular this can occur during a **read**(2V) or **write**(2V) on a slow device (such as a terminal; but not a file) and during a **wait**(2V). After the signal catching function returns, the interrupted system call may return a −1 to the calling process with **errno** set to EINTR.

RETURN VALUES

signal() returns the previous action on success. On failure, it returns −1 and sets **errno** to indicate the error.

ERRORS

signal() will fail and no action will take place if one of the following occurs:

EINVAL *sig* was not a valid signal number.

An attempt was made to ignore or supply a handler for **SIGKILL** or **SIGSTOP**.

SEE ALSO

kill(1), **execve**(2V), **fork**(2V), **getitimer**(2), **getrlimit**(2), **kill**(2V), **ptrace**(2), **read**(2V), **sigblock**(2), **sigpause**(2V), **sigsetmask**(2), **sigstack**(2), **sigvec**(2), **vfork**(2), **wait**(2V), **write**(2V), **setjmp**(3V), **termio**(4)

NOTES

The handler routine can be declared:

```
void handler(sig, code, scp, addr)
int sig, code;
struct sigcontext *scp;
char *addr;
```

Here *sig* is the signal number; *code* is a parameter of certain signals that provides additional detail; *scp* is a pointer to the **sigcontext** structure (defined in **<signal.h>**), used to restore the context from before the signal; and *addr* is additional address information. See **sigvec**(2) for more details.

NAME

sigsetops, sigaddset, sigdelset, sigfillset, sigemptyset, sigismember – manipulate signal sets

SYNOPSIS

```
#include <signal.h>

int sigaddset(set, signo)
sigset_t *set;
int signo;

int sigdelset(set, signo)
sigset_t *set;
int signo;

int sigfillset(set)
sigset_t *set;

int sigemptyset(set)
sigset_t *set;

int sigismember(set, signo)
sigset_t *set
int signo;
```

DESCRIPTION

The **sigsetops** primitives manipulate sets of signals. They operate on data objects addressable by the application. They do not operate on any set of signals known to the system, such as the set blocked from delivery to a process or the set pending for a process.

sigaddset( ) and **sigdelset**( ) respectively add and delete the individual signal specified by the value of **signo** from the signal set pointed to by *set*.

sigemptyset( ) initializes the signal set pointed to by *set* such that all signals defined in this standard are excluded.

sigfillset( ) initializes the signal set pointed to by *set* such that all signals defined in this standard are included.

Applications shall call either **sigemptyset**( ) or **sigfillset**( ) at least once for each object of type **sigset_t** prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of **sigaddset**( ), **sigdelset**( ), **sigismember**( ), **sigaction**( ), **sigprocmask**( ), **sigpending**( ), or **sigsuspend**( ) the results are undefined.

sigismember( ) tests whether the signal specified by the value of **signo** is a member of the set pointed to by *set*.

RETURN VALUES

sigismember( ) returns:

1	if the specified signal is a member of *set*.
0	if the specified signal is not a member of *set*.
−1	if an error is detected, and sets **errno** to indicate the error.

The other functions return:

0	on success.
−1	on failure and set **errno** to indicate the error.

ERRORS

For each of the following conditions, if the condition is detected, **sigaddset**( ), **sigdelset**( ), and **sigismember**( ) set errno to:

EINVAL	**signo** is an invalid or unsupported signal number.

SEE ALSO

sigaction(3V), **sigpending**(2V), **sigprocmask**(2V)

NAME

sleep – suspend execution for interval

SYNOPSIS

int sleep(seconds)
unsigned seconds;

SYSTEM V SYNOPSIS

unsigned sleep(seconds)
unsigned seconds;

DESCRIPTION

sleep() suspends the current process from execution for the number of seconds specified by the argument. The actual suspension time may be an arbitrary amount longer because of other activity in the system.

sleep() is implemented by setting an interval timer and pausing until it expires. The previous state of this timer is saved and restored. If the sleep time exceeds the time to the expiration of the previous value of the timer, the process sleeps only until the timer would have expired, and the signal which occurs with the expiration of the timer is sent one second later.

SYSTEM V DESCRIPTION

sleep() suspends the current process from execution until either the number of real time seconds specified by *seconds* have elapsed or a signal is delivered to the calling process and its action is to invoke a signal-catching function or to terminate the process. The suspension time may be an arbitrary amount longer than requested because of other activity in the system. The value returned by **sleep()** will be the "unslept" amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested **sleep()** time, or premature arousal due to another caught signal.

RETURN VALUES

sleep() returns no useful value.

SYSTEM V RETURN VALUES

If **sleep()** returns because the requested time has elapsed, it returns 0. If **sleep()** returns due to the delivery of a signal, it returns the "unslept" amount in seconds.

SEE ALSO

getitimer(2), **sigpause**(2V), **usleep**(3)

NOTES

SIGALRM should *not* be blocked or ignored during a call to **sleep()**. Only a prior call to **alarm**(3V) should generate SIGALRM for the calling process during a call to **sleep()**. A signal-catching function should *not* interrupt a call to **sleep()** to call **siglongjmp()** or **longjmp()** to restore an environment saved prior to the **sleep()** call.

WARNINGS

sleep() is slightly incompatible with **alarm**(3V). Programs that do not execute for at least one second of clock time between successive calls to **sleep()** indefinitely delay the alarm signal. Use System V **sleep()**. Each **sleep(3V)** call postpones the alarm signal that would have been sent during the requested sleep period to occur one second later.

NAME

sputl, sgetl – access long integer data in a machine-independent fashion

SYNOPSIS

```
void sputl(value, buffer)
long value;
char *buffer;

long sgetl(buffer)
char *buffer;
```

DESCRIPTION

sputl() takes the four bytes of the long integer **values** and places them in memory starting at the address pointed to by *buffer*. The ordering of the bytes is the same across all machines.

sgetl() retrieves the four bytes in memory starting at the address pointed to by *buffer* and returns the long integer value in the byte ordering of the host machine.

The combination of **sputl()** and **sgetl()** provides a machine-independent way of storing long numeric data in a file in binary form without conversion to characters.

NAME

ssignal, gsignal – software signals

SYNOPSIS

#include <signal.h>

int (*ssignal (sig, action))( )
int sig, (*action)( );

int gsignal (sig)
int sig;

DESCRIPTION

ssignal( ) and **ssignal**( ) implement a software facility similar to **signal**(3V).

Software signals made available to users are associated with integers in the inclusive range 1 through 15. A call to **ssignal**( ) associates a procedure, *action*, with the software signal *sig*; the software signal, *sig*, is raised by a call to **ssignal**( ). 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**.

ssignal( ) 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*. **ssignal**( ) returns the value returned to it by the action function.
>
> If the action for *sig* is **SIG_IGN**, **ssignal**( ) returns the value 1 and takes no other action.
>
> If the action for *sig* is **SIG_DFL**, **ssignal**( ) returns the value 0 and takes no other action.
>
> If *sig* has an illegal value or no action was ever specified for *sig*, **ssignal**( ) returns the value 0 and takes no other action.

SEE ALSO

signal(3V)

NAME

stdio – standard buffered input/output package

SYNOPSIS

#include <stdio.h>

FILE *stdin;
FILE *stdout;
FILE *stderr;

DESCRIPTION

The functions described in section 3S constitute a user-level I/O buffering scheme. The in-line macros getc(3V) and **putc**(3S) handle characters quickly. The macros **getchar**( ) (see **getc**(3V)) and **putchar**( ) (see **putc**(3S)), and the higher level routines **fgetc**( ), **getw**( ) (see **getc**(3V)), **gets**(3S), **fgets**( ) (see **gets**(3S)), **scanf**(3V), **fscanf**( ) (see **scanf**(3V)), **fread**(3S), **fputc**( ), **putw**( ) (see **putc**(3S)), **puts**(3S), **fputs**( ) (see **puts**(3S)), **printf**(3V), **fprintf**( ) (see **printf**(3V)), **fwrite**( ) (see **fread**(3S)) 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**(3V) 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>** include 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 EOF or error by most integer functions that deal with streams (see the individual descriptions for details).

Any module that uses this package must include the header file of pertinent macro definitions, as follows:

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

The functions and constants mentioned in sections labeled 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**( ), **feof**( ), **ferror**( ), **fileno**( ), and **clearerr**( ).

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 **fopen**( ) will cause it to become buffered or line-buffered. When an output stream is unbuffered, information is written to the destination file or terminal as soon as it is output to the stream; when it is buffered, many characters are saved up and written as a block. When it is line-buffered, each line of output is written to the destination file or terminal as soon as the line is completed (that is, as soon as a NEWLINE character is output or, if the output stream is **stdout** or **stderr**, as soon as input is read from **stdin**). **setbuf**(3V), **setbuffer**( ), **setlinebuf**( ), or **setvbuf**( ) (see **setbuf**(3V)) can be used to change the stream's buffering strategy.

SYSTEM V DESCRIPTION

When an output stream is line-buffered, each line of output is written to the destination file or terminal as soon as the line is completed (that is, as soon as a NEWLINE character is output or as soon as input is read from a line-buffered stream).

Output saved up on *all* line-buffered streams is written when input is read from *any* line-buffered stream. Input read from a stream that is not line-buffered does not flush output on line-buffered streams.

RETURN VALUES

The value **EOF** is returned uniformly to indicate that a **FILE** pointer has not been initialized with **fopen()**, input (output) has been attempted on an output (input) stream, or a **FILE** pointer designates corrupt or otherwise unintelligible **FILE** data.

SEE ALSO

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

NOTES

The line buffering of output to terminals is almost always transparent, but may cause confusion or malfunctioning of programs which use standard I/O routines but use **read**(2V) to read from the standard input, as calls to **read()** do not cause output to line-buffered streams to be flushed.

In cases where a large amount of computation is done after printing part of a line on an output terminal, it is necessary to call **fflush()** (see **fclose**(3V)) on the standard output before performing the computation so that the output will appear.

BUGS

The standard buffered functions do not interact well with certain other library and system functions, especially **vfork**(2).

NAME

strcoll, strxfrm – compare or transform strings using collating information

SYNOPSIS

```
#include <string.h>

int strcoll(s1, s2)
char *s1;
char *s2;

size_t strxfrm(s1, s2, n)
char *s1;
char *s2;
size_t n;
```

DESCRIPTION

strcoll( ) compares the string pointed to by *s1* to the string pointed to by *s2*. These strings are interpreted as appropriate to the **LC_COLLATE** category of the current locale.

strxfrm( ) transforms the string pointed to by *s2* and places the resulting string into the array pointed to by *s1*. The transformation is such that if **string( )** is applied to two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the **strcoll( )** function applied to the same two original strings. No more than *n* characters are placed into the resulting array pointed to by *s1*, including the terminating null character. If *n* is zero, *s1* is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUES

On success, **strcoll( )** returns an integer greater than, equal to or less than zero, respectively, if the string pointed to by *s1* is greater than, equal to or less than the string pointed to by *s2* when both are interpreted as appropriate to the current locale. On failure, **strcoll( )** sets **errno** to indicate the error, but returns no special value.

strxfrm( ) returns the length of the transformed string, not including the terminating null character. If the value returned is *n* or more, the contents of the array pointed to by *s1* are indeterminate. On failure, **strxfrm( )** returns **(size_t)**–1, and sets **errno** to indicate the error.

ERRORS

EINVAL *s1* or *s2* contain characters outside the domain of the collating sequence.

SEE ALSO

string(3)

NAME

strcat, strncat, strdup, strcmp, strncmp, strcasecmp, strncasecmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strstr, strtok, index, rindex – string operations

SYNOPSIS

```
#include <string.h>

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

char *strncat(s1, s2, n)
char *s1, *s2;
int n;

char *strdup(s1)
char *s1;

int strcmp(s1, s2)
char *s1, *s2;

int strncmp(s1, s2, n)
char *s1, *s2;
int n;

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

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

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

char *strncpy(s1, s2, n)
char *s1, *s2;
int n;

int strlen(s)
char *s;

char *strchr(s, c)
char *s;
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 *strstr(s1, s2)
char *s1, *s2;

char *strtok(s1, s2)
char *s1, *s2;
```

```
#include <strings.h>

char *index(s, c)
char *s, c;

char *rindex(s, c)
char *s, c;
```

DESCRIPTION

These functions operate on null-terminated strings. They do not check for overflow of any receiving string.

strcat( ) appends a copy of string *s2* to the end of string *s1*. **strncat( )** appends at most *n* characters. Each returns a pointer to the null-terminated result.

strcmp( ) compares its arguments and returns an integer greater than, equal to, or less than 0, according as *s1* is lexicographically greater than, equal to, or less than *s2*. **strncmp( )** makes the same comparison but compares at most *n* characters. Two additional routines **strcasecmp( )** and **strncasecmp( )** compare the strings and ignore differences in case. These routines assume the ASCII character set when equating lower and upper case characters.

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**(3V). If the new string cannot be created, a NULL pointer is returned.

strcpy( ) copies string *s2* to *s1* until the null character has been copied. **strncpy( )** copies string *s2* to *s1* until either the null character has been copied or *n* characters have been copied. If the length of *s2* is less than *n*, **strncpy( )** pads *s1* with null characters. If the length of *s2* is *n* or greater, *s1* will not be null-terminated. Both functions return *s1*.

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

strchr( ) (**strrchar( )**) 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.

index( ) (**rindex( )**) 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. These functions are identical to **strchr( )** (**strchr( )**) and merely have different names.

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*.

strstr( ) returns a pointer to the first occurrence of the pattern string *s2* in *s1*. For example, if *s1* is "**string thing**" and *s2* is "**ing**", **strstr( )** returns "**ing thing**". If *s2* does not occur in *s1*, **strstr( )** returns NULL.

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) will work through the string *s1* immediately following that token. In this way subsequent calls will work through the string *s1* until no tokens remain. The separator string *s2* may be different from call to call. When no token remains in *s1*, a NULL pointer is returned.

NOTES

For user convenience, all these functions, except for **index()** and **rindex()**, are declared in the optional **<string.h>** header file. All these functions, including **index()** and **rindex()** but excluding **strchr()**, **strrchr()**, **strpbrk()**, **strspn()**, **strcspn()**, and **strtok()** are declared in the optional **<strings.h>** include file; these headers are set this way for backward compatibility.

SEE ALSO

malloc(3V), **bstring(3)**

WARNINGS

strcmp() and **strncmp()** use native character comparison, which is signed on the Sun, but may be unsigned on other machines. Thus the sign of the value returned when one of the characters has its high-order bit set is implementation-dependent.

strcasecmp() and **strncasecmp()** use native character comparison as above and assume the *ASCII* character set.

On the Sun processor, as well as on many other machines, you can *not* use a NULL pointer to indicate a null string. A NULL pointer is an error and results in an abort of the program. If you wish to indicate a null string, you must have a pointer that points to an explicit null string. On some implementations of the C language on some machines, a NULL pointer, if dereferenced, would yield a null string; this highly non-portable trick was used in some programs. Programmers using a NULL pointer to represent an empty string should be aware of this portability issue; even on machines where dereferencing a NULL pointer does not cause an abort of the program, it does not necessarily yield a null string.

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

NAME

string_to_decimal, file_to_decimal, func_to_decimal – parse characters into decimal record

SYNOPSIS

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

void string_to_decimal(pc,nmax,fortran_conventions,pd,pform,pechar)
char **pc;
int nmax;
int fortran_conventions;
decimal_record *pd;
enum decimal_string_form *pform;
char **pechar;

void file_to_decimal(pc,nmax,fortran_conventions,pd,pform,pechar,pf,pnread)
char **pc;
int nmax;
int fortran_conventions;
decimal_record *pd;
enum decimal_string_form *pform;
char **pechar;
FILE *pf;
int *pnread;

void func_to_decimal(pc,nmax,fortran_conventions,pd,pform,pechar,pget,pnread,punget)
char **pc;
int nmax;
int fortran_conventions;
decimal_record *pd;
enum decimal_string_form *pform;
char **pechar;
int (*pget)();
int *pnread;
int (*punget)();
```

DESCRIPTION

The **char_to_decimal()** functions parse a numeric token from at most *nmax* characters in a string ***pc* or file **pf* or function *(*pget)*() into a decimal record **pd*, classifying the form of the string in **pform* and **pechar*. The accepted syntax is intended to be sufficiently flexible to accomodate many languages:

> *whitespace value*

or

> *whitespace sign value*

where *whitespace* is any number of characters defined by *isspace* in **<ctype.h>**, *sign* is either of [+–], and *value* can be *number*, *nan*, or *inf*. *inf* can be INF (*inf_form*) or INFINITY (*infinity_form*) without regard to case. *nan* can be NAN (*nan_form*) or NAN(*nstring*) (*nanstring_form*) without regard to case; *nstring* is any string of characters not containing ')' or the null character; *nstring* is copied to *pd*–>ds and, currently, not used subsequently. *number* consists of

> *significant*

or

> *significant efield*

where *significant* must contain one or more digits and may contain one point; possible forms are

digits	*(int_form)*
digits.	*(intdot_form)*
.digits	*(dotfrac_form)*
digits.digits	*(intdotfrac_form)*

efield consists of

echar digits

or

echar sign digits

where *echar* is one of [Ee], and *digits* contains one or more digits.

When *fortran_conventions* is nonzero, additional input forms are accepted according to various Fortran conventions:

0 no Fortran conventions
1 Fortran list-directed input conventions
2 Fortran formatted input conventions, ignore blanks (**BN**)
3 Fortran formatted input conventions, blanks are zeros (**BZ**)

When *fortran_conventions* is nonzero, *echar* may also be one of [Dd], and *efield* may also have the form

sign digits.

When *fortran_conventions*>= 2, blanks may appear in the *digits* strings for the integer, fraction, and exponent fields and may appear between *echar* and the exponent sign and after the infinity and NaN forms. If *fortran_conventions*== 2, the blanks are ignored. When *fortran_conventions*== 3, the blanks that appear in *digits* strings are interpreted as zeros, and other blanks are ignored.

When *fortran_conventions* is zero, the current locale's decimal point character is used as the decimal point; when *fortran_conventions* is nonzero, the period is used as the decimal point.

The form of the accepted decimal string is placed in _*peform_. If an *efield* is recognized, _*pechar_ is set to point to the *echar*.

On input, _*pc_ points to the beginning of a character string buffer of length >= *nmax*. On output, _*pc_ points to a character in that buffer, one past the last accepted character. **string_to_decimal**( ) gets its characters from the buffer; **file_to_decimal**( ) gets its characters from _*pf_ and records them in the buffer, and places a null after the last character read. **func_to_decimal**( ) gets its characters from an int function _(*pget)_( ).

The scan continues until no more characters could possibly fit the acceptable syntax or until *nmax* characters have been scanned. If the *nmax* limit is not reached then at least one extra character will usually be scanned that is not part of the accepted syntax. **file_to_decimal**( ) and **func_to_decimal**( ) set _*pnread_ to the number of characters read from the file; if greater than *nmax*, some characters were lost. If no characters were lost, **file_to_decimal**( ) and **func_to_decimal**( ) attempt to push back, with **ungetc**(3S) or _(*punget)_( ), as many as possible of the excess characters read, adjusting _*pnread_ accordingly. If all unget calls are successful, then _**pc_ will be a null character. No push back will be attempted if _(*punget)_( ) is NULL.

Typical declarations for **pget*( ) and **punget*( ) are:

```
int xget( )
{... }
int (*pget)( ) = xget;
int xunget(c)
char c ;
{...}
int (*punget)( ) = xunget;
```

If no valid number was detected, *pd*–>**fpclass** is set to **fp_signaling**, **pc* is unchanged, and **pform* is set to **invalid_form**.

atof( ) and **strtod**(3) use **string_to_decimal**( ). **scanf**(3V) uses **file_to_decimal**( ).

SEE ALSO

ctype(3V), **localeconv**(3), **scanf**(3V), **setlocale**(3V), **strtod**(3), **ungetc**(3S)

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, using **string_to_decimal**(3), with *fortran_conventions* set to 0.

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 for historical compatibility, 0.0 is returned, although a NaN would better match the IEEE Floating-Point Standard's intent.

The radix character is defined by the program's locale (category LC_NUMERIC). In the "C" locale, or in a locale where the radix character is not defined. the radix character defaults to a period '.'.

atof(str) is equivalent to **strtod(str, (char **)**NULL). Thus, when **atof(str)** returns 0.0 there is no way to determine whether *str* contained a valid numerical string representing 0.0 or an invalid numerical string.

SEE ALSO

scanf(3V), **string_to_decimal**(3)

DIAGNOSTICS

Exponent overflow and underflow produce the results specified by the IEEE Standard. In addition, **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**(3V)) 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(3V), **scanf**(3V), **strtod**(3)

BUGS

Overflow conditions are ignored.

NAME

stty, gtty – set and get terminal state

SYNOPSIS

```
#include <sgtty.h>

stty(fd, buf)
int fd;
struct sgttyb *buf;

gtty(fd, buf)
int fd;
struct sgttyb *buf;
```

DESCRIPTION

Note: this interface is obsoleted by **ioctl**(2).

stty() sets the state of the terminal associated with *fd*. **stty**() retrieves the state of the terminal associated with *fd*. To set the state of a terminal the call must have write permission.

The **stty**() call is actually

```
ioctl(fd, TIOCSETP, buf)
```

while the **gtty**() call is

```
ioctl(fd, TIOCGETP, buf)
```

See **ioctl**(2) and **ttcompat**(4M) for an explanation.

DIAGNOSTICS

If the call is successful 0 is returned, otherwise –1 is returned and the global variable **errno** contains the reason for the failure.

SEE ALSO

ioctl(2), **ttcompat**(4M)

NAME

swab – swap bytes

SYNOPSIS

```
void
swab(from, to, nbytes)
char *from, *to;
```

DESCRIPTION

swab() copies *nbytes* bytes pointed to by *from* to the position pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between high-ender machines (IBM 360's, MC68000's, etc) and low-end machines (such as Sun386i systems).

nbytes should be even and positive. If *nbytes* is odd and positive, swab() uses *nbytes* – 1 instead. If *nbytes* is negative, swab() does nothing.

The *from* and *to* addresses should not overlap in portable programs.

NAME

syslog, openlog, closelog, setlogmask – control system log

SYNOPSIS

```
#include <syslog.h>

openlog(ident, logopt, facility)
char *ident;

syslog(priority, message, parameters ... )
char *message;

closelog()

setlogmask(maskpri)
```

DESCRIPTION

syslog() passes *message* to **syslogd**(8), which logs it in an appropriate system log, writes it to the system console, forwards it to a list of users, or forwards it to the **syslogd** on another host over the network. The message is tagged with a priority of *priority*. The message looks like a **printf**(3V) string except that **%m** is replaced by the current error message (collected from **errno**). A trailing NEWLINE is added if needed.

Priorities are encoded as a *facility* and a *level*. The facility describes the part of the system generating the message. The level is selected from an ordered list:

LOG_EMERG A panic condition. This is normally broadcast to all users.

LOG_ALERT A condition that should be corrected immediately, such as a corrupted system database.

LOG_CRIT Critical conditions, such as hard device errors.

LOG_ERR Errors.

LOG_WARNING Warning messages.

LOG_NOTICE Conditions that are not error conditions, but that may require special handling.

LOG_INFO Informational messages.

LOG_DEBUG Messages that contain information normally of use only when debugging a program.

If special processing is needed, **openlog()** can be called to initialize the log file. The parameter *ident* is a string that is prepended to every message. *logopt* is a bit field indicating logging options. Current values for *logopt* are:

LOG_PID Log the process ID with each message. This is useful for identifying specific daemon processes (for daemons that fork).

LOG_CONS Write messages to the system console if they cannot be sent to **syslogd**. This option is safe to use in daemon processes that have no controlling terminal, since **syslog()** forks before opening the console.

LOG_NDELAY Open the connection to **syslogd** immediately. Normally the open is delayed until the first message is logged. This is useful for programs that need to manage the order in which file descriptors are allocated.

LOG_NOWAIT Do not wait for child processes that have been forked to log messages onto the console. This option should be used by processes that enable notification of child termination using **SIGCHLD**, since **syslog()** may otherwise block waiting for a child whose exit status has already been collected.

The *facility* parameter encodes a default facility to be assigned to all messages that do not have an explicit facility already encoded:

LOG_KERN Messages generated by the kernel. These cannot be generated by any user processes.

LOG_USER Messages generated by random user processes. This is the default facility identifier if none is specified.

LOG_MAIL The mail system.

LOG_DAEMON System daemons, such as **ftpd**(8C), **routed**(8C), etc.

LOG_AUTH The authorization system: **login**(1), **su**(1V), **getty**(8), etc.

LOG_LPR The line printer spooling system: **lpr**(1), **lpc**(8), **lpd**(8), etc.

LOG_NEWS Reserved for the USENET network news system.

LOG_UUCP Reserved for the UUCP system; it does not currently use **syslog**.

LOG_CRON The **cron/at** facility; **crontab**(1), **at**(1), **cron**(8), etc.

LOG_LOCAL0–7 Reserved for local use.

closelog( ) can be used to close the log file.

setlogmask( ) sets the log priority mask to *maskpri* and returns the previous mask. Calls to syslog( ) with a priority not set in *maskpri* are rejected. The mask for an individual priority *pri* is calculated by the macro **LOG_MASK**(*pri*); the mask for all priorities up to and including *toppri* is given by the macro **LOG_UPTO**(*toppri*). The default allows all priorities to be logged.

EXAMPLES

This call logs a message at priority **LOG_ALERT**:

```
syslog(LOG_ALERT, "who: internal error 23");
```

The FTP daemon **ftpd** would make this call to **openlog**( ) to indicate that all messages it logs should have an identifying string of **ftpd**, should be treated by **syslogd** as other messages from system daemons are, should include the process ID of the process logging the message:

```
openlog("ftpd", LOG_PID, LOG_DAEMON);
```

Then it would make the following call to **setlogmask**( ) to indicate that messages at priorities from **LOG_EMERG** through **LOG_ERR** should be logged, but that no messages at any other priority should be logged:

```
setlogmask(LOG_UPTO(LOG_ERR));
```

Then, to log a message at priority **LOG_INFO**, it would make the following call to syslog:

```
syslog(LOG_INFO, "Connection from host %d", CallingHost);
```

A locally-written utility could use the following call to **syslog**( ) to log a message at priority **LOG_INFO** to be treated by **syslogd** as other messages to the facility **LOG_LOCAL2** are:

```
syslog(LOG_INFO|LOG_LOCAL2, "error: %m");
```

SEE ALSO

at(1), **crontab**(1), **logger**(1), **login**(1), **lpr**(1), **su**(1V), **printf**(3V), **syslog.conf**(5), **cron**(8), **ftpd**(8C), **getty**(8), **lpc**(8), **lpd**(8), **routed**(8C), **syslogd**(8)

NAME

system – issue a shell command

SYNOPSIS

system(string)
char *string;

DESCRIPTION

system( ) gives the *string* to **sh(1)** as input, just as if the string had been typed as a command from a terminal. The current process performs a **wait(2V)** system call, and waits until the shell terminates. **system( )** then returns the exit status returned by **wait(2V)**. Unless the shell was interrupted by a signal, its termination status is contained in the 8 bits higher up from the low-order 8 bits of the value returned by **wait( )**.

SEE ALSO

sh(1), **execve(2V)**, **wait(2V)**, **popen(3S)**

DIAGNOSTICS

Exit status 127 (may be displayed as "32512") indicates the shell could not be executed.

NAME

t_accept – accept a connect request

SYNOPSIS

```
#include <tiuser.h>

int t_accept(fd, resfd, call)
int fd;
int resfd;
struct t_call *call;
```

DESCRIPTION

t_accept() is issued by a transport user to accept a connect request. *fd* identifies the local transport endpoint where the connect indication arrived, *resfd* specifies the local transport endpoint where the connection is to be established, and *call* contains information required by the transport provider to complete the connection. *call* points to a **t_call** structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

The *netbuf* structure 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 transport function will replace the user value of *len* on return. *maxlen* generally has significance only when *buf* is used to receive output from the transport function. In this case, it specifies the physical size of the buffer, and the maximum value of *len* that can be set by the function. If *maxlen* is not large enough to hold the returned information, a TBUFOVFLW error will generally result. However, certain functions may return part of the data and not generate an error. In *call*, *addr* is the address of the caller, *opt* indicates any protocol-specific parameters associated with the connection, *udata* points to any user data to be returned to the caller, and *sequence* is the value returned by **t_listen**(3N) that uniquely associates the response with a previously received connect indication.

A transport user may accept a connection on either the same, or on a different, local transport endpoint than the one on which the connect indication arrived. If the same endpoint is specified (*resfd* = *fd*), the connection can be accepted unless the following condition is true: The user has received other indications on that endpoint but has not responded to them (with **t_accept()** or **t_snddis**(3N)). For this condition, **t_accept()** will fail and set **t_errno** to TBADF.

If a different transport endpoint is specified (*resfd* != *fd*), the endpoint must be bound to a protocol address and must be in the **T_IDLE** state (see **t_getstate**(3N)) before the **t_accept()** is issued.

For both types of endpoints, **t_accept()** will fail and set **t_errno** to TLOOK if there are indications (such as a connect or disconnect) waiting to be received on that endpoint.

The values of parameters specified by *opt* and the syntax of those values are protocol specific. The *udata* field enables the called transport user to send user data to the caller and the amount of user data must not exceed the limits supported by the transport provider as returned by **t_open**(3N) or **t_getinfo**(3N). If the *len* field of *udata* is zero, no data will be sent to the caller.

RETURN VALUES

t_accept() returns:

0	on success.
−1	on failure and sets **t_errno** to indicate the error.

ERRORS

TACCES
The user does not have permission to accept a connection on the responding transport endpoint.

The user does not have permission to use the specified options.

TBADDATA
The amount of user data specified was not within the bounds allowed by the transport provider.

TBADF
The specified file descriptor does not refer to a transport endpoint.

The user is illegally accepting a connection on the same transport endpoint on which the connect indication arrived.

TBADOPT
The specified options were in an incorrect format or contained illegal information.

TBADSEQ
An invalid sequence number was specified.

TLOOK
An asynchronous event has occurred on the transport endpoint referenced by *fd* and requires immediate attention.

TNOTSUPPORT
This function is not supported by the underlying transport provider.

TOUTSTATE
The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

The transport endpoint referred to by *resfd* is not in the **T_IDLE** state.

TSYSERR
The function failed due to a system error and set **errno** to indicate the error.

SEE ALSO

intro(3), **t_connect**(3N), **t_getstate**(3N), **t_listen**(3N), **t_open**(3N), **t_rcvconnect**(3N)

Network Programming

NAME

t_alloc – allocate a library structure

SYNOPSIS

```
#include <tiuser.h>

char *t_alloc(fd, struct_type, fields)
int fd;
int struct_type;
int fields;
```

DESCRIPTION

t_alloc() dynamically allocates memory for the various transport function argument structures as specified below. **t_alloc()** allocates memory for the specified structure and for buffers referenced by the structure.

The structure to allocate is specified by *struct_type*, and can be one of the following (each of of these structures may be used as an argument to one or more transport functions):

T_BIND	**struct t_bind**
T_CALL	**struct t_call**
T_OPTMGMT	**struct t_optmgmt**
T_DIS	**struct t_discon**
T_UNITDATA	**struct t_unitdata**
T_UDERROR	**struct t_uderr**
T_INFO	**struct t_info**

Each of the above structures, except **T_INFO**, contains at least one field of type '**struct netbuf**'. The *maxlen*, *len*, and *buf* members of the **netbuf** structure are described in **t_accept**(3N). For each field of this type, the user may specify that the buffer for that field should be allocated as well. The *fields* argument specifies this option, where the argument is the bitwise-OR of any of the following:

T_ADDR	The *addr* field of the **t_bind**, **t_call**, **t_unitdata**, or **t_uderr** structures.
T_OPT	The *opt* field of the **t_optmgmt**, **t_call**, **t_unitdata**, or **t_uderr** structures.
T_UDATA	The *udata* field of the **t_call**, **t_discon**, or **t_unitdata** structures.
T_ALL	All relevant fields of the given structure.

For each field specified in *fields*, **t_alloc()** allocates memory for the buffer associated with the field, and initializes the *buf* pointer and *maxlen* field accordingly. The length of the buffer allocated is based on the same size information returned to the user on **t_open**(3N) and **t_getinfo**(3N). Thus, *fd* must refer to the transport endpoint through which the newly allocated structure is passed, so that the appropriate size information can be accessed. If the size value associated with any specified field is −1 or −2 (see **t_open**(3N) or **t_getinfo**(3N)), **t_alloc()** is unable to determine the size of the buffer to allocate and fails, setting **t_errno** to TSYSERR and **errno** to EINVAL . For any field not specified in *fields*, *buf* is set to NULL and *maxlen* is set to zero.

Use of **t_alloc()** to allocate structures helps ensure the compatibility of user programs with future releases of the transport interface.

RETURN VALUES

On success, **t_alloc()** returns a pointer to the type of structure specified by **struct_type**. On failure, it returns NULL and sets **t_errno** to indicate the error.

ERRORS

TBADF	The specified file descriptor does not refer to a transport endpoint.
TSYSERR	The function failed due to a system error and set **errno** to indicate the error.

SEE ALSO

intro(3), **t_free**(3N), **t_getinfo**(3N), **t_open**(3N)

Network Programming

NAME

t_bind – bind an address to a transport endpoint

SYNOPSIS

```
#include <tiuser.h>

int t_bind(fd, req, ret)
int fd;
struct t_bind *req;
struct t_bind *ret;
```

DESCRIPTION

t_bind() associates a protocol address with the transport endpoint specified by *fd* and activates that transport endpoint. In connection mode, the transport provider may begin accepting or requesting connections on the transport endpoint. In connectionless mode, the transport user may send or receive data units through the transport endpoint.

The *req* and *ret* arguments point to a **t_bind**() structure containing the following members:

```
        struct netbuf addr;
        unsigned qlen;
```

The *maxlen*, *len*, and *buf* members of the *netbuf* structure are described in **t_accept**(3N). The *addr* field of the **t_bind**() structure specifies a protocol address and the *qlen* field is used to indicate the maximum number of outstanding connect indications.

req is used to request that an address, represented by the *netbuf* structure, be bound to the given transport endpoint. *len* specifies the number of bytes in the address and *buf* points to the address buffer. *maxlen* has no meaning for the *req* argument. On return, *ret* contains the address that the transport provider actually bound to the transport endpoint; this may be different from the address specified by the user in *req*. In *ret*, the user specifies *maxlen* which is the maximum size of the address buffer and *buf* which points to the buffer where the address is to be placed. On return, *len* specifies the number of bytes in the bound address and *buf* points to the bound address. If *maxlen* is not large enough to hold the returned address, an error will result.

If the requested address is not available, or if no address is specified in *req* (the *len* field of *addr* in *req* is 0) the transport provider will assign an appropriate address to be bound, and will return that address in the *addr* field of *ret*. The user can compare the addresses in *req* and *ret* to determine whether the transport provider bound the transport endpoint to a different address than that requested.

req may be NULL if the user does not wish to specify an address to be bound. Here, the value of *qlen* is assumed to be 0, and the transport provider must assign an address to the transport endpoint. Similarly, *ret* may be NULL if the user does not care what address was bound by the transport provider and is not interested in the negotiated value of *qlen*. It is valid to set *req* and *ret* to NULL for the same call, in which case the transport provider chooses the address to bind to the transport endpoint and does not return that information to the user.

The *qlen* field has meaning only when initializing a connection-mode service. It specifies the number of outstanding connect indications the transport provider should support for the given transport endpoint. An outstanding connect indication is one that has been passed to the transport user by the transport provider. A value of *qlen* greater than 0 is only meaningful when issued by a passive transport user that expects other users to call it. The value of *qlen* will be negotiated by the transport provider and may be changed if the transport provider cannot support the specified number of outstanding connect indications. On return, the *qlen* field in *ret* will contain the negotiated value.

t_bind() allows more than one transport endpoint to be bound to the same protocol address (however, the transport provider must support this capability also), but binding more than one protocol address to the same transport endpoint is not allowed. If a user binds more than one transport endpoint to the same protocol address, only one endpoint can be used to listen for connect indications associated with that protocol address. In other words, only one **t_bind**() for a given protocol address may specify a value of *qlen* greater than 0. In this way, the transport provider can identify which transport endpoint should be notified of an incoming connect indication. If a user attempts to bind a protocol address to a second transport endpoint with a value of *qlen* greater than 0, the transport provider will assign another address to be bound to that endpoint. If a user accepts a connection on the transport endpoint that is being used as the listening endpoint, the bound protocol address will be found to be busy for the duration of that connection. No other transport endpoints may be bound for listening while that initial listening endpoint is in the data transfer phase. This will prevent more than one transport endpoint bound to the same protocol address from accepting connect indications.

RETURN VALUES

t_bind() returns:

0	on success.
−1	on failure and sets **t_errno** to indicate the error.

ERRORS

TACCES The user does not have permission to use the specified address.

TBADADDR The specified protocol address was in an incorrect format or contained illegal information.

TBADF The specified file descriptor does not refer to a transport endpoint.

TBUFOVFLW The number of bytes allowed for an incoming argument is not sufficient to store the value of that argument. The transport provider's state will change to T_IDLE and the information to be returned in *ret* will be discarded.

TNOADDR The transport provider could not allocate an address.

TOUTSTATE The function was issued in the wrong sequence.

TSYSERR The function failed due to a system error and set **errno** to indicate the error.

SEE ALSO

intro(3), **t_open**(3N), **t_optmgmt**(3N), **t_unbind**(3N)

Network Programming

NAME

t_close – close a transport endpoint

SYNOPSIS

```
#include <tiuser.h>

int t_close(fd)
int fd;
```

DESCRIPTION

t_close( ) informs the transport provider that the user is finished with the transport endpoint specified by *fd*, and frees any local library resources associated with the endpoint. In addition, **t_close**( ) closes the file associated with the transport endpoint.

t_close( ) should be called from the **T_UNBND** state (see **t_getstate**(3N)). However, **t_close**( ) does not check state information, so it may be called from any state to close a transport endpoint. If this occurs, the local library resources associated with the endpoint will be freed automatically. In addition, **close**(2V) will be issued for that file descriptor; the close will be abortive if no other process has that file open, and will break any transport connection that may be associated with that endpoint.

RETURN VALUES

t_close( ) returns:

0	on success.
–1	on failure and sets **t_errno** to indicate the error.

ERRORS

TBADF	The specified file descriptor does not refer to a transport endpoint.

SEE ALSO

close(2V), **t_getstate**(3N), **t_open**(3N), **t_unbind**(3N)

Network Programming

NAME

t_connect – establish a connection with another transport user

SYNOPSIS

```
#include <tiuser.h>

int t_connect(fd, sndcall, rcvcall)
int fd;
struct t_call *sndcall;
struct t_call *rcvcall;
```

DESCRIPTION

t_connect( ) enables a transport user to request a connection to the specified destination transport user. *fd* identifies the local transport endpoint where communication will be established, while *sndcall* and *rcvcall* point to a **t_call**( ) structure which contains the following members:

```
        struct netbuf addr;
        struct netbuf opt;
        struct netbuf udata;
        int sequence;
```

sndcall specifies information needed by the transport provider to establish a connection and *rcvcall* specifies information that is associated with the newly established connection.

The *maxlen*, *len*, and *buf* members of the *netbuf* structure are described in **t_accept**(3N). In *sndcall*, *addr* specifies the protocol address of the destination transport user, *opt* presents any protocol-specific information that might be needed by the transport provider, *udata* points to optional user data that may be passed to the destination transport user during connection establishment, and *sequence* has no meaning for this function.

On return in *rcvcall*, *addr* returns the protocol address associated with the responding transport endpoint, *opt* presents any protocol-specific information associated with the connection, *udata* points to optional user data that may be returned by the destination transport user during connection establishment, and *sequence* has no meaning for this function.

opt implies no structure on the options that may be passed to the transport provider. The transport provider is free to specify the structure of any options passed to it. These options are specific to the underlying protocol of the transport provider. The user may choose not to negotiate protocol options by setting the *len* field of *opt* to 0. In this case, the transport provider may use default options.

udata enables the caller to pass user data to the destination transport user and receive user data from the destination user during connection establishment. However, the amount of user data must not exceed the limits supported by the transport provider as returned by **t_open**(3N) or **t_getinfo**(3N). If the *len* field of *udata* is 0 in *sndcall*, no data will be sent to the destination transport user.

On return, the *addr*, *opt*, and *udata* fields of *rcvcall* will be updated to reflect values associated with the connection. Thus, the *maxlen* field of each argument must be set before issuing this function to indicate the maximum size of the buffer for each. However, *rcvcall* may be NULL in which case no information is given to the user on return from **t_connect**( ).

By default, **t_connect**( ) executes in synchronous mode, and will wait for the destination user's response before returning control to the local user. A successful return (a return value of 0) indicates that the requested connection has been established. However, if **T_NDELAY** is set (using **t_open**( ) or **fcntl**), **t_connect**( ) executes in asynchronous mode. In this case, the call will not wait for the remote user's response, but will return control immediately to the local user and return –1 with **t_errno** set to TNODATA to indicate that the connection has not yet been established. In this way, the function simply initiates the connection establishment procedure by sending a connect request to the destination transport user.

RETURN VALUES

t_connect( ) returns:

0 on success.

−1 on failure and sets **t_errno** to indicate the error.

ERRORS

TACCES The user does not have permission to use the specified address or options.

TBADADDR The specified protocol address was in an incorrect format or contained illegal information.

TBADDATA The amount of user data specified was not within the bounds allowed by the transport provider.

TBADF The specified file descriptor does not refer to a transport endpoint.

TBADOPT The specified protocol options were in an incorrect format or contained illegal information.

TBUFOVFLW The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. If executed in synchronous mode, the transport provider's state, as seen by the user, changes to **T_DATAXFER** and the connect indication information to be returned in *rcvcall* is discarded.

TLOOK An asynchronous event has occurred on this transport endpoint and requires immediate attention.

TNODATA **T_NDELAY** was set, so the function successfully initiated the connection establishment procedure, but did not wait for a response from the remote user.

TNOTSUPPORT This function is not supported by the underlying transport provider.

TOUTSTATE The function was issued in the wrong sequence.

TSYSERR The function failed due to a system error and set **errno** to indicate the error.

SEE ALSO

intro(3), **t_accept**(3N), **t_getinfo**(3N), **t_listen**(3N), **t_open**(3N), **t_optmgmt**(3N), **t_rcvconnect**(3N)

Network Programming

NAME

t_error – produce error message

SYNOPSIS

```
#include <tiuser.h>

void t_error(errmsg)
char *errmsg;

extern int t_errno;
extern char *t_errlist[ ];
extern int t_nerr;
```

DESCRIPTION

t_error( ) produces a message on the standard error output which describes the last error received during a call to a transport function. The argument string *errmsg* is a user-supplied error message that gives context to the error. **t_error( )** prints the user-supplied error message followed by a colon and a standard error message for the current error defined in **t_errno**. To simplify variant formatting of messages, the array of message strings **t_errlist** is provided; **t_errno** can be used as an index in this table to get the message string without the NEWLINE. **t_nerr** is the largest message number provided for in the **t_errlist** table.

t_errno is only set when an error occurs and is not cleared on successful calls.

EXAMPLE

If a **t_connect**(3N) function fails on transport endpoint *fd*2 because a bad address was given, the following call might follow the failure:

```
t_error ("t_connect failed on fd2");
```

The diagnostic message to be printed would look like:

```
t_connect failed on fd2:  Incorrect transport address format
```

where '**Incorrect transport address format**' identifies the specific error that occurred, and '**t_connect failed on fd2**' tells the user which function failed on which transport endpoint.

SEE ALSO

Network Programming

NAME

t_free – free a library structure

SYNOPSIS

```
#include <tiuser.h>

int t_free(ptr, struct_type)
char *ptr;
int struct_type;
```

DESCRIPTION

t_free() frees memory previously allocated by **t_alloc**(3N). This function will free memory for the specified structure, and will also free memory for buffers referenced by the structure.

ptr points to one of the six structure types described for **t_alloc**(3N), and *struct_type* identifies the type of that structure which can be one of the following:

T_BIND	**struct t_bind**
T_CALL	**struct t_call**
T_OPTMGMT	**struct t_optmgmt**
T_DIS	**struct t_discon**
T_UNITDATA	**struct t_unitdata**
T_UDERROR	**struct t_uderr**
T_INFO	**struct t_info**

where each of these structures is used as an argument to one or more transport functions.

t_free() checks the *addr*, *opt*, and *udata* fields of the given structure (as appropriate), and frees the buffers pointed to by the *buf* field of the *netbuf* (see **intro**(3)) structure. The *maxlen*, *len*, and *buf* members of the *netbuf* structure are described in **t_accept**(3N). If *buf* is NULL, **t_free()** will not attempt to free memory. After all buffers are freed, **t_free()** will free the memory associated with the structure pointed to by *ptr*.

Undefined results will occur if *ptr* or any of the *buf* pointers points to a block of memory that was not previously allocated by **t_alloc**(3N).

RETURN VALUES

t_free() returns:

0 on success.

−1 on failure and sets **t_errno** to indicate the error.

ERRORS

TSYSERR The function failed due to a system error and set **errno** to indicate the error.

SEE ALSO

intro(3), **t_alloc**(3N)

Network Programming

NAME

t_getinfo – get protocol-specific service information

SYNOPSIS

```
#include <tiuser.h>

int t_getinfo(fd, info)
int fd;
struct t_info *info;
```

DESCRIPTION

t_getinfo( ) returns the current characteristics of the underlying transport protocol associated with file descriptor *fd*. The *info* structure is used to return the same information returned by **t_open**(3N). **t_getinfo( )** enables a transport user to access this information during any phase of communication.

This argument points to a **t_info** structure which contains the following members:

```
long addr;       /* max size of the transport protocol address */
long options;    /* max number of bytes of protocol-specific options */
long tsdu;       /* max size of a transport service data unit (TSDU) */
long etsdu;      /* max size of an expedited transport service data unit (ETSDU) */
long connect;    /* max amount of data allowed on connection establishment
                    functions */
long discon;     /* max amount of data allowed on t_snddis and t_rcvdis functions */
long servtype;   /* service type supported by the transport provider */
```

FIELDS

The values of the fields have the following meanings:

addr A value greater than or equal to zero indicates the maximum size of a transport protocol address; a value of –1 specifies that there is no limit on the address size; and a value of –2 specifies that the transport provider does not provide user access to transport protocol addresses.

options A value greater than or equal to zero indicates the maximum number of bytes of protocol-specific options supported by the provider; a value of –1 specifies that there is no limit on the option size; and a value of –2 specifies that the transport provider does not support user-settable options.

tsdu A value greater than zero specifies the maximum size of a transport service data unit (TSDU); a value of zero specifies that the transport provider does not support the concept of TSDU, although it does support the sending of a data stream with no logical boundaries preserved across a connection; a value of –1 specifies that there is no limit on the size of a TSDU; and a value of –2 specifies that the transfer of normal data is not supported by the transport provider.

etsdu A value greater than zero specifies the maximum size of an expedited transport service data unit (ETSDU); a value of zero specifies that the transport provider does not support the concept of ETSDU, although it does support the sending of an expedited data stream with no logical boundaries preserved across a connection; a value of –1 specifies that there is no limit on the size of an ETSDU; and a value of –2 specifies that the transfer of expedited data is not supported by the transport provider.

connect A value greater than or equal to zero specifies the maximum amount of data that may be associated with connection establishment functions; a value of –1 specifies that there is no limit on the amount of data sent during connection establishment; and a value of –2 specifies that the transport provider does not allow data to be sent with connection establishment functions.

discon A value greater than or equal to zero specifies the maximum amount of data that may be associated with the **t_snddis**(3N) and **t_rcvdis**(3N) functions; a value of −1 specifies that there is no limit on the amount of data sent with these abortive release functions; and a value of −2 specifies that the transport provider does not allow data to be sent with the abortive release functions.

servtype This field specifies the service type supported by the transport provider, as described below.

If a transport user is concerned with protocol independence, the above sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the **t_alloc**(3N) function may be used to allocate these buffers. An error will result if a transport user exceeds the allowed data size on any function. The value of each field may change as a result of option negotiation, and **t_getinfo**( ) enables a user to retrieve the current characteristics.

RETURN VALUES

The *servtype* field of *info* may specify one of the following values on return:

T_COTS The transport provider supports a connection-mode service but does not support the optional orderly release facility.

T_COTS_ORD The transport provider supports a connection-mode service with the optional orderly release facility.

T_CLTS The transport provider supports a connectionless-mode service. For this service type, **t_open**(3N) will return −2 for the **etsdu**, **connect**, and **discon** fields.

RETURN VALUES

t_getinfo( ) returns 0 on success and −1 on failure.

ERRORS

TBADF The specified file descriptor does not refer to a transport endpoint.

TSYSERR The function failed due to a system error and set **errno** to indicate the error.

SEE ALSO

t_open(3N)

Network Programming

NAME

t_getstate – get the current state

SYNOPSIS

```
#include <tiuser.h>

int t_getstate(fd)
int fd;
```

DESCRIPTION

t_getstate() returns the current state of the provider associated with the transport endpoint specified by *fd*.

If the provider is undergoing a state transition when **t_getstate()** is called, the function will fail. **t_getstate()** returns the current state on successful completion and −1 on failure and **t_errno** is set to indicate the error. The current state may be one of the following:

T_UNBND	unbound
T_IDLE	idle
T_OUTCON	outgoing connection pending
T_INCON	incoming connection pending
T_DATAXFER	data transfer
T_OUTREL	outgoing orderly release (waiting for an orderly release indication)
T_INREL	incoming orderly release (waiting for an orderly release request)

RETURN VALUES

t_getstate() returns:

0	on success.
−1	on failure and sets **t_errno** to indicate the error.

ERRORS

TBADF	The specified file descriptor does not refer to a transport endpoint.
TSTATECHNG	The transport provider is undergoing a state change.
TSYSERR	The function failed due to a system error and set **errno** to indicate the error.

SEE ALSO

t_open(3N)

Network Programming

NAME

t_listen – listen for a connect request

SYNOPSIS

```
#include <tiuser.h>

int t_listen(fd, call)
int fd;
struct t_call *call;
```

DESCRIPTION

t_listen( ) listens for a connect request from a calling transport user. *fd* identifies the local transport endpoint where connect indications arrive, and on return, *call* contains information describing the connect indication. *call* points to a **t_call**( ) structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

The *maxlen*, *len*, and *buf* members of the *netbuf* structure are described in **t_accept**(3N). In *call*, *addr* returns the protocol address of the calling transport user, *opt* returns protocol-specific parameters associated with the connect request, *udata* returns any user data sent by the caller on the connect request, and *sequence* is a number that uniquely identifies the returned connect indication. The value of *sequence* enables the user to listen for multiple connect indications before responding to any of them.

Since this function returns values for the *addr*, *opt*, and *udata* fields of *call*, the *maxlen* field of each must be set before issuing the **t_listen**( ) to indicate the maximum size of the buffer for each.

By default, **t_listen**( ) executes in synchronous mode and waits for a connect indication to arrive before returning to the user. However, if **T_NDELAY** is set (using **t_open**(3N) or **fcntl**( )), **t_listen**( ) executes asynchronously, reducing to a **poll**(2) for existing connect indications. If none are available, it returns –1 and sets **t_errno** to TNODATA.

RETURN VALUES

t_listen( ) returns:

0	on success.
–1	on failure and sets **t_errno** to indicate the error.

ERRORS

TBADF	The specified file descriptor does not refer to a transport endpoint.
TBUFOVFLW	The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. The provider's state, as seen by the user, changes to **T_INCON** and the connect indication information to be returned in *call* is discarded.
TLOOK	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
TNODATA	**T_NDELAY** was set, but no connect indications had been queued.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TSYSERR	The function failed due to a system error and set **errno** to indicate the error.

SEE ALSO

intro(3), **t_accept**(3N), **t_bind**(3N), **t_connect**(3N), **t_open**(3N), **t_rcvconnect**(3N)

Network Programming

NAME

t_look – look at the current event on a transport endpoint

SYNOPSIS

```
#include <tiuser.h>

int t_look(fd)
int fd;
```

DESCRIPTION

t_look() returns the current event on the transport endpoint specified by *fd*. This function enables a transport provider to notify a transport user of an asynchronous event when the user is issuing functions in synchronous mode. Certain events require immediate notification of the user and are indicated by a specific error, TLOOK, on the current or next function to be executed.

This function also enables a transport user to **poll**(2) a transport endpoint periodically for asynchronous events.

RETURN VALUES

Upon success, **t_look()** returns a value that indicates which of the allowable events has occurred, or returns zero if no event exists. One of the following events is returned:

T_LISTEN	Connection indication received
T_CONNECT	Connect confirmation received
T_DATA	Normal data received
T_EXDATA	Expedited data received
T_DISCONNECT	Disconnect received
T_ERROR	Fatal error indication
T_UDERR	Datagram error indication
T_ORDREL	Orderly release indication

On failure, –1 is returned and **t_errno** is set to indicate the error.

ERRORS

TBADF	The specified file descriptor does not refer to a transport endpoint.
TSYSERR	The function failed due to a system error and set **errno** to indicate the error.

SEE ALSO

t_open(3N)

Network Programming

NAME

t_open – establish a transport endpoint

SYNOPSIS

```
#include <tiuser.h>

int t_open(path, oflag, info)
char *path;
int oflag;
struct t_info *info;
```

DESCRIPTION

t_open() must be called as the first step in the initialization of a transport endpoint. It establishes a transport endpoint by opening a file that identifies a particular transport provider (such as a transport protocol) and returning a file descriptor that identifies that endpoint. For example, opening the file **/dev/tcp** identifies an OSI connection-oriented transport layer protocol as the transport provider. Currently, **/dev/tcp** is the only transport protocol available to **t_open()**.

path points to the pathname of the file to open, and *oflag* identifies any open flags (as in **open**(2V)). **t_open()** returns a file descriptor that will be used by all subsequent functions to identify the particular local transport endpoint.

This function also returns various default characteristics of the underlying transport protocol by setting fields in the **t_info** structure pointed to by *info*. **t_info** is defined in **<nettli/tiuser.h>** as:

```
struct t_info {
        long addr;      /* size of protocol address */
        long options;   /* size of protocol options */
        long tsdu;      /* size of max transport service data unit */
        long etsdu;     /* size of max expedited tsdu */
        long connect;   /* max data for connection primitives */
        long discon;    /* max data for disconnect primitives */
        long servtype;  /* provider service type */
};
```

The fields of this structure have the following values:

addr A value greater than or equal to zero indicates the maximum size of a transport protocol address; a value of –1 specifies that there is no limit on the address size; and a value of –2 specifies that the transport provider does not provide user access to transport protocol addresses.

options A value greater than or equal to zero indicates the maximum number of bytes of protocol-specific options supported by the provider; a value of –1 specifies that there is no limit on the option size; and a value of –2 specifies that the transport provider does not support user-settable options.

tsdu A value greater than zero specifies the maximum size of a transport service data unit (TSDU); a value of zero specifies that the transport provider does not support the concept of TSDU, although it does support the sending of a data stream with no logical boundaries preserved across a connection; a value of –1 specifies that there is no limit on the size of a TSDU; and a value of –2 specifies that the transfer of normal data is not supported by the transport provider.

etdsu A value greater than zero specifies the maximum size of an expedited transport service data unit (ETSDU); a value of zero specifies that the transport provider does not support the concept of ETSDU, although it does support the sending of an expedited data stream with no logical boundaries preserved across a connection; a value of –1 specifies that there is no limit on the size of an ETSDU; and a value of –2 specifies that the transfer of expedited data is not supported by the transport provider.

connect A value greater than or equal to zero specifies the maximum amount of data that may be associated with connection establishment functions; a value of −1 specifies that there is no limit on the amount of data sent during connection establishment; and a value of −2 specifies that the transport provider does not allow data to be sent with connection establishment functions.

discon A value greater than or equal to zero specifies the maximum amount of data that may be associated with the **t_snddis**(3N) and **t_rcvdis**(3N) functions; a value of −1 specifies that there is no limit on the amount of data sent with these abortive release functions; and a value of −2 specifies that the transport provider does not allow data to be sent with the abortive release functions.

servtype This field specifies the service type supported by the transport provider.

The *servtype* field of *info* may specify one of the following values on return:

T_COTS The transport provider supports a connection-mode service but does not support the optional orderly release facility.

T_COTS_ORD The transport provider supports a connection-mode service with the optional orderly release facility.

T_CLTS The transport provider supports a connectionless-mode service. For this service type, **t_open**() will return −2 for *etsdu*, *connect*, and *discon*.

A single transport endpoint may support only one of the above services at one time.

If *info* is set to NULL by the transport user, no protocol information is returned by **t_open**().

If a transport user is concerned with protocol independence, the above sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the **t_alloc**(3N) function may be used to allocate these buffers. An error will result if a transport user exceeds the allowed data size on any function.

RETURN VALUES

t_open() returns a non-negative file descriptor on success. On failure, it returns −1 and sets **t_errno** to indicate the error.

ERRORS

TSYSERR The function failed due to a system error and set **errno** to indicate the error.

SEE ALSO

open(2V), **tcp**(4P)

Network Programming

NAME

t_optmgmt – manage options for a transport endpoint

SYNOPSIS

```
#include <tiuser.h>

int t_optmgmt(fd, req, ret)
int fd;
struct t_optmgmt *req;
struct t_optmgmt *ret;
```

DESCRIPTION

t_optmgmt() enables a transport user to retrieve, verify, or negotiate protocol options with the transport provider. *fd* identifies a bound transport endpoint.

The *req* and *ret* arguments point to a **t_optmgmt()** structure containing the following members:

```
    struct netbuf opt;
    long     flags;
```

The *opt* field identifies protocol options and the *flags* field is used to specify the action to take with those options.

The options are represented by a *netbuff* structure in a manner similar to the address in **t_bind**(3N). The *maxlen*, *len*, and *buf* members of the *netbuf* structure are described in **t_accept**(3N). *req* is used to request a specific action of the provider and to send options to the provider. *len* specifies the number of bytes in the options, *buf* points to the options buffer, and *maxlen* has no meaning for the *req* argument. The transport provider may return options and flag values to the user through *ret*. For *ret*, *maxlen* specifies the maximum size of the options buffer and *buf* points to the buffer where the options are to be placed. On return, *len* specifies the number of bytes of options returned. *maxlen* has no meaning for the *req* argument, but must be set in the *ret* argument to specify the maximum number of bytes the options buffer can hold. The actual structure and content of the options is imposed by the transport provider.

The flags field of *req* can specify one of the following actions:

T_NEGOTIATE Enables the user to negotiate the values of the options specified in *req* with the transport provider. The provider will evaluate the requested options and negotiate the values, returning the negotiated values through *ret*.

T_CHECK Enables the user to verify whether the options specified in *req* are supported by the transport provider. On return, the flags field of *ret* will have either **T_SUCCESS** or **T_FAILURE** set to indicate to the user whether the options are supported. These flags are only meaningful for the T_CHECK request.

T_DEFAULT Enables a user to retrieve the default options supported by the transport provider into the *opt* field of *ret*. In *req*, the *len* field of *opt* must be zero and the *buf* field may be NULL.

If issued as part of the connectionless-mode service, **t_optmgmt()** may block due to flow control constraints. **t_optmgmt()** will not complete until the transport provider has processed all previously sent data units.

RETURN VALUES

t_optmgmt() returns:

0 on success.

–1 on failure and sets **t_errno** to indicate the error.

ERRORS

TACCES	The user does not have permission to negotiate the specified options.
TBADF	The specified file descriptor does not refer to a transport endpoint.
TBADFLAG	An invalid flag was specified.
TBADOPT	The specified protocol options were in an incorrect format or contained illegal information.
TBUFOVFLW	The number of bytes allowed for an incoming argument is not sufficient to store the value of that argument. The information to be returned in *ret* will be discarded.
TOUTSTATE	The function was issued in the wrong sequence.
TSYSERR	The function failed due to a system error and set **errno** to indicate the error.

SEE ALSO

intro(3), **t_getinfo**(3N), **t_open**(3N)

Network Programming

NAME

t_rcv – receive normal or expedited data sent over a connection

SYNOPSIS

```
int t_rcv(fd, buf, nbytes, flags)

int fd;
char *buf;
unsigned nbytes;
int *flags;
```

DESCRIPTION

t_rcv() receives either normal or expedited data. *fd* identifies the local transport endpoint through which data will arrive, *buf* points to a receive buffer where user data will be placed, and *nbytes* specifies the size of the receive buffer. *flags* may be set on return from **t_rcv()** and specifies optional flags as described below.

By default, **t_rcv()** operates in synchronous mode and will wait for data to arrive if none is currently available. However, if T_NDELAY is set (using **t_open**(3N) or **fcntl()**), **t_rcv()** will execute in asynchronous mode and will fail if no data is available. See TNODATA below.

On return from the call, if **T_MORE** is set in *flags* this indicates that there is more data and the current transport service data unit (TSDU) or expedited transport service data unit (ETSDU) must be received in multiple **t_rcv()** calls. Each **t_rcv()** with the **T_MORE** flag set indicates that another **t_rcv()** must follow immediately to get more data for the current TSDU. The end of the TSDU is identified by the return of a **t_rcv()** call with the **T_MORE** flag not set. If the transport provider does not support the concept of a TSDU as indicated in the *info* argument on return from **t_open**(3N) or **t_getinfo**(3N), the **T_MORE** flag is not meaningful and should be ignored.

On return, the data returned is expedited data if **T_EXPEDITED** is set in *flags*. If the number of bytes of expedited data exceeds *nbytes*, **t_rcv()** will set **T_EXPEDITED** and **T_MORE** on return from the initial call. Subsequent calls to retrieve the remaining ETSDU will not have **T_EXPEDITED** set on return. The end of the ETSDU is identified by the return of a **t_rcv()** call with the **T_MORE** flag not set.

If expedited data arrives after part of a TSDU has been retrieved, receipt of the remainder of the TSDU will be suspended until the ETSDU has been processed. Only after the full ETSDU has been retrieved (T_MORE not set) will the remainder of the TSDU be available to the user.

RETURN VALUES

On success, **t_rcv()** returns the number of bytes received. On failure, it returns −1.

ERRORS

TBADF	The specified file descriptor does not refer to a transport endpoint.
TLOOK	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
TNODATA	**T_NDELAY** was set, but no data is currently available from the transport provider.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TSYSERR	The function failed due to a system error and set **errno** to indicate the error.

SEE ALSO

t_open(3N), **t_snd**(3N)

Network Programming

NAME

t_rcvconnect – receive the confirmation from a connect request

SYNOPSIS

```
#include <tiuser.h>

int t_rcvconnect(fd, call)
int fd;
struct t_call *call;
```

DESCRIPTION

t_rcvconnect allows a calling transport user to get the status of a previous connect request. It can be used in conjunction with **t_connect**(3N) to establish a connection in asynchronous mode.

fd identifies the local transport endpoint where communication is established. *call* contains information associated with the newly established connection *call* points to a *t_call* structure that contains information associated with the new connection, and is defined in **<nettli/tiuser.h>** as:

```
struct t_call {
        struct netbuf addr;
        struct netbuf opt;
        struct netbuf udata;
        int sequence;
};
```

The *maxlen*, *len*, and *buf* members of the *netbuf* structure are described in **t_accept**(3N). In the *t_call* structure, *addr* returns the protocol address associated with the responding transport endpoint, *opt* presents protocol-specific information associated with the connection, *udata* points to optional user data that may be returned by the destination transport user during connection establishment, and *sequence* has no meaning for this function.

The *maxlen* field of each argument must be set before issuing this function to indicate the maximum buffer size. However, *call* may be NULL, in which case no information is given to the user on return from **t_rcvconnect**( ). By default, **t_rcvconnect**( ) executes synchronously and waits for the connection before returning. On return, the *addr*, *opt*, and *udata* fields reflect values associated with the connection.

If **O_NDELAY** is set (using **t_open**(3N) or **fcntl**( )), **t_rcvconnect**( ) executes asynchronously, reducing to a **poll**(2) request for existing connect confirmations. If none are available, **t_rcvconnect**( ) fails and returns immediately without waiting for the connection to be established. See **TNODATA** below. **t_rcvconnect**( ) must be re-issued at a later time to complete the connection establishment phase and retrieve the information returned in *call*.

RETURN VALUES

t_rcvconnect( ) returns:

0	on success.
–1	on failure and sets **t_errno** to indicate the error.

ERRORS

TBADF	The specified file descriptor does not refer to a transport endpoint.
TBUFOVFLW	The bytes allocated for an incoming argument is sufficient to store the value of that argument and the connect information to be returned in *call* is discarded. The transport provider's state, as seen by the user, will be changed to DATAXFER.
TNODATA	**O_NDELAY** was set, but a connect confirmation has not yet arrived.
TLOOK	An asynchronous event has occurred on this transport connection and requires immediate attention.

TNOTSUPPORT This function is not supported by the underlying transport provider.

TSYSERR The function failed due to a system error and set **errno** to indicate the error.

SEE ALSO

poll(2), **intro**(3), **t_accept**(3N), **t_bind**(3N), **t_connect**(3N), **t_listen**(3N), **t_open**(3N)

Network Programming

NAME

t_rcvdis – retrieve information from disconnect

SYNOPSIS

```
#include <tiuser.h>

t_rcvdis(fd, discon)
int fd;
struct t_discon *discon;
```

DESCRIPTION

t_rcvdis( ) is used to identify the cause of a disconnect, and to retrieve any user data sent with the disconnect. *fd* identifies the local transport endpoint where the connection existed, and *discon* points to a **t_discon** structure defined in **<nettli/tiuser.>** as:

```
struct t_discon {
        struct netbuf udata;            /* user data */
        int reason;                     /* reason code */
        int sequence;                   /* sequence number */
};
```

The *maxlen*, *len*, and *buf* members of the *netbuf* structure are described in **t_accept**(3N). *reason* specifies the reason for the disconnect through a protocol-dependent reason code, *udata* identifies any user data that was sent with the disconnect, and *sequence* may identify an outstanding connect indication with which the disconnect is associated. *sequence* is only meaningful when **t_rcvdis**( ) is issued by a passive transport user who has executed one or more **t_listen(3N)** functions and is processing the resulting connect indications. If a disconnect indication occurs, *sequence* can be used to identify which of the outstanding connect indications is associated with the disconnect.

If a user does not care if there is incoming data and does not need to know the value of *reason* or *sequence*, *discon* may be NULL and any user data associated with the disconnect will be discarded. However, if a user has retrieved more than one outstanding connect indication (using **t_listen**(3N)) and *discon* is NULL, the user will be unable to identify with which connect indication the disconnect is associated.

RETURN VALUES

t_rcvdis( ) returns:

0 on success.

–1 on failure and sets **t_errno** to indicate the error.

ERRORS

TBADF The specified file descriptor does not refer to a transport endpoint.

TBUFOVFLW The number of bytes allocated for incoming data is not sufficient to store the data. The provider's state, as seen by the user, will change to **T_IDLE** and the disconnect indication information to be returned in *discon* will be discarded.

TNODIS No disconnect indication currently exists on the specified transport endpoint.

TNOTSUPPORT This function is not supported by the underlying transport provider.

TSYSERR The function failed due to a system error and set **errno** to indicate the error.

SEE ALSO

intro(3), **t_connect**(3N), **t_listen**(3N), **t_open**(3N), **t_snddis**(3N)

Network Programming

NAME

t_rcvrel – acknowledge receipt of an orderly release indication

SYNOPSIS

```
#include <tiuser.h>

int t_rcvrel(fd)
int fd;
```

DESCRIPTION

t_rcvel( ) acknowledges receipt of an orderly release indication. *fd* identifies the local transport endpoint where the connection exists. After receipt of this indication, the user may not attempt to receive more data because such an attempt will block forever. However, the user may continue to send data over the connection if **t_sndrel**(3N) has not been issued by the user.

t_rcvrel( ) is an optional service of the transport provider, and is only supported if the transport provider returned service type **T_COTS_ORD** on **t_open**(3N) or **t_getinfo**(3N).

RETURN VALUES

t_rcvrel( ) returns:

0	on success.
–1	on failure and sets **t_errno** to indicate the error.

ERRORS

TBADF	The specified file descriptor does not refer to a transport endpoint.
TLOOK	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
TNOREL	No orderly release indication currently exists on the specified transport endpoint.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TSYSERR	The function failed due to a system error and set **errno** to indicate the error.

SEE ALSO

t_open(3N), **t_sndrel**(3N)

Network Programming

NAME

t_rcvudata – receive a data unit

SYNOPSIS

```
#include <tiuser.h>

int t_rcvudata(fd, unitdata, flags)
int fd;
struct t_unitdata *unitdata;
int *flags;
```

DESCRIPTION

t_rcvudata() is used in connectionless mode to receive a data unit from another transport user. *fd* identifies the local transport endpoint through which data will be received, *unitdata* holds information associated with the received data unit, and *flags* is set on return to indicate that the complete data unit was not received. *unitdata* points to a *t_unitdata* structure defined in **<nettli/tiuser.h>** as:

```
struct t_unitdata {
        struct netbuf addr;          /* address        */
        struct netbuf opt;           /* options        */
        struct netbuf udata;         /* user data              */
};
```

The *maxlen*, *len*, and *buf* members of the *netbuf* structure are described in **t_accept**(3N). The *maxlen* field of *addr*, *opt*, and *udata* must be set before issuing **t_rcvudata()** to indicate the maximum size of the buffer for each.

On return from this call, *addr* specifies the protocol address of the sending user, *opt* identifies protocol-specific options that were associated with this data unit, and *udata* specifies the user data that was received.

By default, **t_rcvudata()** operates in synchronous mode and will wait for a data unit to arrive if none is currently available. However, if **O_NDELAY** is set (using **t_open**(3N) or **fcntl()**), **t_rcvudata()** will execute in asynchronous mode and will fail if no data units are available.

If the buffer defined in the *udata* field of *unitdata* is not large enough to hold the current data unit, the buffer will be filled and **T_MORE** will be set in *flags* on return to indicate that another **t_rcvudata()** should be issued to retrieve the rest of the data unit. Subsequent **t_rcvudata()** call(s) will return zero for the length of the address and options until the full data unit has been received.

RETURN VALUES

t_rcvudata() returns:

0 on success.

–1 on failure and sets **t_errno** to indicate the error.

ERRORS

TBADF The specified file descriptor does not refer to a transport endpoint.

TBUFOVFLW The number of bytes allocated for the incoming protocol address or options is not sufficient to store the information. The unit data information to be returned in *unitdata* will be discarded.

TLOOK An asynchronous event has occurred on this transport endpoint and requires immediate attention.

TNODATA **T_NDELAY** was set, but no data units are currently available from the transport provider.

TNOTSUPPORT This function is not supported by the underlying transport provider.

TSYSERR The function failed due to a system error and set **errno** to indicate the error.

SEE ALSO

intro(3), **t_rcvuderr**(3N), **t_sndudata**(3N)

NAME

t_rcvuderr – receive a unit data error indication

SYNOPSIS

```
#include <tiuser.h>

int t_rcvuderr(fd, uderr)
int fd;
struct t_uderr *uderr;
```

DESCRIPTION

t_rcvuderr( ) is used in connectionless mode to receive information concerning an error on a previously sent data unit, and should only be issued following a unit data error indication. It informs the transport user that a data unit with a specific destination address and protocol options produced an error. *fd* identifies the local transport endpoint through which the error report will be received, and *uderr* points to a **t_uderr**( ) structure defined in **<nettli/tiuser.h>** as:

```
        struct t_uderr {
                struct netbuf addr;                     /* address      */
                struct netbuf opt;                      /* options      */
                long  error;                            /* error code   */
        };
```

The *maxlen*, *len*, and *buf* members of the *netbuf* structure are described in **t_accept**(3N). The *maxlen* field of *addr* and *opt* must be set before issuing this function to indicate the maximum size of the buffer for each.

On return from this call, the *addr* structure specifies the destination protocol address of the erroneous data unit, the *opt* structure identifies protocol-specific options that were associated with the data unit, and **error** specifies a protocol-dependent error code.

If the user does not care to identify the data unit that produced an error, *uderr* may be set to NULL and **t_rcvuderr**( ) will simply clear the error indication without reporting any information to the user.

RETURN VALUES

t_rcvuderr( ) returns:

0	on success.
−1	on failure and sets **t_errno** to indicate the error.

ERRORS

TBADF	The specified file descriptor does not refer to a transport endpoint.
TBUFOVFLW	The number of bytes allocated for the incoming protocol address or options is not sufficient to store the information. The unit data error information to be returned in *uderr* will be discarded.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TNOUDERR	No unit data error indication currently exists on the specified transport endpoint.
TSYSERR	The function failed due to a system error and set **errno** to indicate the error.

SEE ALSO

intro(3), **t_rcvudata**(3N), **t_sndudata**(3N)

Network Programming

NAME

t_snd – send normal or expedited data over a connection

SYNOPSIS

```
#include <tiuser.h>

int t_snd(fd, buf, nbytes, flags)
int fd;
char *buf;
unsigned nbytes;
int flags;
```

DESCRIPTION

t_snd() sends either normal or expedited data. *fd* identifies the local transport endpoint over which data should be sent, *buf* points to the user data, *nbytes* specifies the number of user data bytes to be sent, and *flags* specifies any optional flags described below.

By default, **t_snd()** operates synchronously and may wait if flow control restrictions prevents data acceptance by the local transport provider when the call is made. However, if **O_NDELAY** is set (using **t_open**(3N) or **fcntl()**), **t_snd()** executes asynchronously, and fails immediately if there are flow control restrictions.

On success, **t_snd()** returns the byte total accepted by the transport provider. This normally equals the bytes total specified in *nbytes*. If **O_NDELAY** is set, it is possible that the transport provider will accept only part of the data. In this case, **t_snd()** will set **T_MORE** for the data that was sent (see below) and returns a value less than *nbytes*. If *nbytes* is zero, no data is passed to the provider; **t_snd()** returns zero.

If **T_EXPEDITED** is set in *flags*, the data is sent as expedited data, subject to the interpretations of the transport provider.

T_MORE indicates to the transport provider that the transport service data unit (TSDU), or expedited transport service data unit (ETSDU), is being sent through multiple **t_snd()** calls. In these calls, the **T_MORE** flag indicates another **t_snd()** is to follow; the end of TSDU (or ETSDU) is identified by a **t_snd()** call without the **T_MORE** flag. **T_MORE** allows the sender to break up large logical data units, while preserving their boundaries at the other end. The flag does not imply how the data is packaged for transfer below the transport interface. If the transport provider does not support the concept of a TSDU as indicated in the *info* argument on return from **t_open**(3N) or **t_getinfo**(3N), the **T_MORE** flag is meaningless.

The size of each TSDU or ETSDU must not exceed the transport provider limits as returned by **t_open**(3N) or **t_getinfo**(3N). Failure to comply results in protocol error EPROTO. See TSYSERR below.

If **t_snd()** is issued from the **T_IDLE** state, the provider may silently discard the data. If **t_snd()** is issued from any state other than **T_DATAXFER** or **T_IDLE** the provider generates a EPROTO error.

RETURN VALUES

On success, **t_snd()** returns the number of bytes accepted by the transport provider. On failure, it returns –1 and sets **t_errno** to indicate the error.

ERRORS

TBADF	The specified file descriptor does not refer to a transport endpoint.
TFLOW	**O_NDELAY** was set, but the flow control mechanism prevented the transport provider from accepting data at this time.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TSYSERR	The function failed due to a system error and set **errno** to indicate the error.

SEE ALSO

t_open(3N), **t_rcv**(3N)

Network Programming

NAME

t_snddis – send user-initiated disconnect request

SYNOPSIS

#include <tiuser.h>

```
int t_snddis(fd, call)
int fd;
struct t_call *call;
```

DESCRIPTION

t_snddis( ) is used to initiate an abortive release on an already established connection or to reject a connect request. *fd* identifies the local transport endpoint of the connection, and *call* specifies information associated with the abortive release. *call* points to a **t_call**( ) structure which is defined in **<nettlie/tiuser.h>** as:

```
struct t_call {
        struct netbuf addr;             /* address          */
        struct netbuf opt;              /* options          */
        struct netbuf udata;            /* user data        */
        int sequence;                   /* sequence number  */
};
```

The *maxlen*, *len*, and *buf* members of the *netbuf* structure are described in **t_accept**(3N). The values in *call* have different semantics, depending on the context of the call to **t_snddis**( ). When rejecting a connect request, *call* must be non-NULL and contain a valid value of *sequence* to uniquely identify the rejected connect indication to the transport provider. The *addr* and *opt* fields of *call* are ignored. In all other cases, *call* need only be used when data is being sent with the disconnect request. The *addr*, *opt*, and *sequence* fields of the **t_call**( ) structure are ignored. If the user does not wish to send data to the remote user, the value of *call* may be NULL. *udata* specifies the user data to be sent to the remote user. The amount of user data must not exceed the limits supported by the transport provider as returned by **t_open**(3N) or **t_getinfo**(3N). If the *len* field of *udata* is zero, no data will be sent to the remote user.

RETURN VALUES

t_snddis( ) returns:

0	on success.
–1	on failure and sets **t_errno** to indicate the error.

ERRORS

TBADDATA	The amount of user data specified was not within the bounds allowed by the transport provider. The transport provider's outgoing queue will be flushed, so data may be lost.
TBADF	The specified file descriptor does not refer to a transport endpoint.
TBADSEQ	An invalid sequence number was specified. The transport provider's outgoing queue will be flushed, so data may be lost. A NULL call structure was specified when rejecting a connect request. The transport provider's outgoing queue will be flushed, so data may be lost.
TLOOK	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TOUTSTATE	The function was issued in the wrong sequence. The transport provider's outgoing queue may be flushed, so data may be lost.
TSYSERR	The function failed due to a system error and set **errno** to indicate the error.

SEE ALSO

intro(3), **t_connect**(3N), **t_getinfo**(3N), **t_listen**(3N), **t_open**(3N)

Network Programming

NAME

t_sndrel – initiate an orderly release

SYNOPSIS

```
#include <tiuser.h>

int t_sndrel(fd)
int fd;
```

DESCRIPTION

t_sndrel( ) initiates an orderly release of a transport connection and indicates to the transport provider that the transport user has no more data to send. *fd* identifies the local transport endpoint where the connection exists. After issuing **t_sndrel**( ), the user may not send any more data over the connection. However, a user may continue to receive data if an orderly release indication has been received.

t_sndrel( ) is an optional service of the transport provider, and is only supported if the transport provider returned service type **T_COTS_ORD** on **t_open**(3N) or **t_getinfo**(3N).

RETURN VALUES

t_sndrel( ) returns:

0	on success.
–1	on failure and sets **t_errno** to indicate the error.

ERRORS

TBADF	The specified file descriptor does not refer to a transport endpoint.
TFLOW	**O_NDELAY** was set, but the flow control mechanism prevented the transport provider from accepting the function at this time.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TSYSERR	The function failed due to a system error and set **errno** to indicate the error.

SEE ALSO

t_open(3N), **t_rcvrel**(3N)

Network Programming

NAME

t_sndudata – send a data unit

SYNOPSIS

```
#include <tiuser.h>

int t_sndudata(fd, unitdata)
int fd;
struct t_unitdata *unitdata;
```

DESCRIPTION

t_sndudata( ) is used in connectionless mode to send a data unit to another transport user. *fd* identifies the local transport endpoint through which data will be sent, and *unitdata* points to a **t_unitdata** structure defined in **<nettli/tiuser.h>** as:

```
struct t_unitdata {
        struct netbuf addr;              /* address        */
        struct netbuf opt;               /* options        */
        struct netbuf udata;             /* user data      */
};
```

The *maxlen*, *len*, and *buf* members of the *netbuf* structure are described in **t_accept**(3N). In *unitdata*, *addr* specifies the protocol address of the destination user, *opt* identifies protocol-specific options that the user wants associated with this request, and *udata* specifies the user data to be sent. The user may choose not to specify what protocol options are associated with the transfer by setting the *len* field of *opt* to 0. In this case, the provider may use default options.

If the *len* field of *udata* is 0, no data unit will be passed to the transport provider; **t_sndudata**( ) will not send zero-length data units.

By default, **t_sndudata**( ) operates in synchronous mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if T_NDELAY is set (using **t_open**(3N) or **fcntl**( )), **t_sndudata**( ) will execute in asynchronous mode and will fail under such conditions.

If **t_sndudata**( ) is issued from an invalid state, or if the amount of data specified in *udata* exceeds the TSDU size as returned by **t_open**( ) or **t_getinfo**(3N), the provider will generate an EPROTO protocol error. See TSYSERR below.

RETURN VALUES

t_sndudata( ) returns:

0	on success.
−1	on failure and sets **t_errno** to indicate the error.

ERRORS

TBADF	The specified file descriptor does not refer to a transport endpoint.
TFLOW	T_NDELAY was set, but the flow control mechanism prevented the transport provider from accepting data at this time.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TSYSERR	The function failed due to a system error and set **errno** to indicate the error.

SEE ALSO

intro(3), **t_rcvudata**(3N), **t_rcvuderr**(3N)

Network Programming

NAME

t_sync – synchronize transport library

SYNOPSIS

#include <tiuser.h>

int t_sync(fd)
int fd;

DESCRIPTION

For the transport endpoint specified by *fd*, **t_sync**( ) synchronizes the data structures managed by the transport library with information from the underlying transport provider. In doing so, it can convert a raw file descriptor (obtained using **open**(2V), **dup**(2V), or as a result of a **fork**(2V) and **execve**(2V)) to an initialized transport endpoint, assuming that file descriptor referenced a transport provider. **t_sync**( ) also allows two cooperating processes to synchronize their interaction with a transport provider.

For example, if a process *forks* a new process and issues an *exec*, the new process must issue a **t_sync**( ) to build the private library data structure associated with a transport endpoint and to synchronize the data structure with the relevant provider information.

It is important to remember that the transport provider treats all users of a transport endpoint as a single user. If multiple processes are using the same endpoint, they should coordinate their activities so as not to violate the state of the provider. **t_sync**( ) returns the current state of the provider to the user, thereby enabling the user to verify the state before taking further action. This coordination is only valid among cooperating processes; it is possible that a process or an incoming event could change the provider's state *after* a **t_sync**( ) is issued.

If the provider is undergoing a state transition when **t_sync**( ) is called, the function will fail.

RETURN VALUES

t_sync( ) returns –1 on failure. Upon success, the state of the transport provider is returned; it may be one of the following:

T_IDLE	idle
T_OUTCON	outgoing connection pending
T_INCON	incoming connection pending
T_DATAXFER	data transfer
T_OUTREL	outgoing orderly release (waiting for an orderly release indication)
T_INREL	incoming orderly release (waiting for an orderly release request)
T_UNBND	unbound

ERRORS

TBADF	The specified file descriptor is a valid open file descriptor but does not refer to a transport endpoint.
TSTATECHNG	The transport provider is undergoing a state change.
TSYSERR	The function failed due to a system error and set **errno** to indicate the error.

SEE ALSO

dup(2V), **execve**(2V), **fork**(2V), **open**(2V)

Network Programming

NAME

t_unbind – disable a transport endpoint

SYNOPSIS

```
#include <tiuser.h>

int t_unbind(fd)
int fd;
```

DESCRIPTION

t_unbind() disables the transport endpoint specified by *fd* which was previously bound by **t_bind**(3N). On completion of this call, no further data or events destined for this transport endpoint will be accepted by the transport provider.

RETURN VALUES

t_unbind() returns:

0	on success.
−1	on failure and sets **t_errno** to indicate the error.

ERRORS

TBADF	The specified file descriptor does not refer to a transport endpoint.
TLOOK	An asynchronous event has occurred on this transport endpoint.
TOUTSTATE	The function was issued in the wrong sequence.
TSYSERR	The function failed due to a system error and set **errno** to indicate the error.

SEE ALSO

t_bind(3N)

Network Programming

NAME

tcgetpgrp, tcsetpgrp – get, set foreground process group ID

SYNOPSIS

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

pid_t tcgetpgrp(fd)
int fd;

int tcsetpgrp(fd, pgrp_id)
int fd;
pid_t pgrp_id;
```

DESCRIPTION

tcgetpgrp( ) returns the value of the process group ID of the foreground process group associated with the terminal (see NOTES). **tcgetpgrp( )** is allowed from a process that is a member of a background process group; however, the information may be subsequently changed by a process that is a member of a foreground process group.

If the process has a controlling terminal, **tcsetpgrp( )** sets the foreground process group ID associated with the terminal to *pgrp_id*. The file associated with *fd* must be the controlling terminal and must be currently associated with the session of the calling process. The value of *pgrp_id* must match a process group ID of a process in the same session as the calling process.

RETURN VALUES

On success, **tcgetpgrp( )** returns the process group ID of the foreground process group associated with the terminal. On failure, it returns –1 and sets **errno** to indicate the error.

tcsetpgrp( ) returns:

0 on success.

–1 on failure and sets **errno** to indicate the error.

ERRORS

If any of the following conditions occur, **tcgetpgrp( )** sets **errno** to:

EBADF *fd* is not a valid file descriptor.

ENOSYS **tcgetpgrp( )** is not supported in this implementation.

ENOTTY The calling process does not have a controlling terminal.

The file is not the controlling terminal.

If any of the following conditions occur, **tcsetpgrp( )** sets **errno** to:

EBADF *fd* is not a valid file descriptor.

EINVAL The value of *pgrp_id* is not a valid process group ID.

ENOTTY The calling process does not have a controlling terminal.

The file is not the controlling terminal.

The controlling terminal is no longer associated with the session of the calling process.

EPERM The value of *pgrp_id* is a valid process group ID, but does not match the process group ID of a process in the same session as the calling process.

SEE ALSO

setpgid(2V), **setsid**(2V)

NOTES

For **tcgetpgrp( )** and **tcsetpgrp( )** to behave as described above, {_POSIX_JOB_CONTROL} must be in effect (see **sysconf**(2V)). {_POSIX_JOB_CONTROL} is always in effect on SunOS systems, but for portability, applications should call **sysconf( )** to determine whether {_POSIX_JOB_CONTROL} is in effect for the current system.

If {_POSIX_JOB_CONTROL} is not defined on a system conforming to *IEEE Std 1003.1-1988* either **tcgetpgrp( )** and **tcsetpgrp( )** behave as described above, or **tcgetpgrp( )** and **tcsetpgrp( )** fail.

NAME

termcap, tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs – terminal independent operation routines

SYNOPSIS

```
char PC;
char *BC;
char *UP;
short ospeed;

tgetent(bp, name)
char *bp, *name;

tgetnum (id)
char *id;

tgetflag (id)
char *id;

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

char *
tgoto(cm, destcol, destline)
char *cm;

tputs(cp, affcnt, outc)
register char *cp;
int affcnt;
int (*outc)( );
```

DESCRIPTION

These functions extract and use capabilities from the terminal capability data base **termcap**(5). These are low level routines; see **curses**(3V) for a higher level package.

tgetent( ) extracts the entry for terminal *name* into the *bp* buffer, with the current size of the tty (usually a window). This allows pre-SunWindows programs to run in a window of arbitrary size. *bp* should be a character buffer of size 1024 and must be retained through all subsequent calls to **tgetnum**( ), **tgetflag**( ), and **tgetstr**( ). **tgetent**( ) returns –1 if it cannot open the **termcap**( ) file, 0 if the terminal name given does not have an entry, and 1 if all goes well. It will look in the environment for a **TERMCAP** variable. If found, and the value does not begin with a slash, and the terminal type *name* is the same as the environment string **TERM**, the **TERMCAP** string is used instead of reading the termcap file. If it does begin with a slash, the string is used as a path name rather than **/etc/termcap**. This can speed up entry into programs that call **tgetent**, as well as to help debug new terminal descriptions or to make one for your terminal if you cannot write the file **/etc/termcap**. Note: if the window size changes, the "lines" and "columns" entries in *bp* are no longer correct. See the *SunView Programmer's Guide* for details regarding [how to handle] this.

tgetnum( ) gets the numeric value of capability **ID**, returning –1 if is not given for the terminal. **tgetflag**( ) returns 1 if the specified capability is present in the terminal's entry, 0 if it is not. **tgetstr**( ) gets the string value of capability **ID**, placing it in the buffer at *area*, advancing the *area* pointer. It decodes the abbreviations for this field described in **termcap**(5), except for cursor addressing and padding information. **tgetstr**( ) returns the string pointer if successful. Otherwise it returns zero.

tgoto() returns a cursor addressing string decoded from *cm* to go to column *destcol* in line *destline*. It uses the external variables **UP** (from the **up** capability) and **BC** (if **bc** is given rather than **bs**) if necessary to avoid placing **\n**, **^D** or **^@** in the returned string. (Programs which call **tgoto**() should be sure to turn off the **XTABS bit(s),since tgoto**() may now output a tab. Note: programs using **termcap**() should in general turn off **XTABS** anyway since some terminals use **^I** (CTRL-I) for other functions, such as nondestructive space.) If a **%** sequence is given which is not understood, then **tgoto**() returns **OOPS**.

tputs() decodes the leading padding information of the string *cp*; *affcnt* gives the number of lines affected by the operation, or 1 if this is not applicable, *outc* is a routine which is called with each character in turn. The external variable *ospeed* should contain the encoded output speed of the terminal as described in **tty**(4). The external variable PC should contain a pad character to be used (from the **pc** capability) if a NULL (^@) is inappropriate.

FILES

/usr/lib/libtermcap.a	–ltermcap library
/etc/termcap	data base

SEE ALSO

ex(1), **curses**(3V), **tty**(4), **termcap**(5)

NAME

termios, tcgetattr, tcsetattr, tcsendbreak, tcdrain, tcflush, tcflow, cfgetospeed, cfgetispeed, cfsetispeed, cfsetospeed – get and set terminal attributes, line control, get and set baud rate, get and set terminal foreground process group ID

SYNOPSIS

```
#include <termios.h>
#include <unistd.h>

int tcgetattr(fd, termios_p)
int fd;
struct termios *termios_p;

int tcsetattr(fd, optional_actions, termios_p)
int fd;
int optional_actions;
struct termios *termios_p;

int tcsendbreak(fd, duration)
int fd;
int duration;

int tcdrain(fd)
int fd;

int tcflush(fd, queue_selector)
int fd;
int queue_selector;

int tcflow(fd, action)
int fd;
int action;

speed_t cfgetospeed(termios_p)
struct termios *termios_p;

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

speed_t cfgetispeed(termios_p)
struct termios *termios_p;

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

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

DESCRIPTION

The termios functions describe a general terminal interface that is provided to control asynchronous communications ports. A more detailed overview of the terminal interface can be found in **termio**(4). That section also describes an **ioctl**( ) interface that can be used to access the same functionality. However, the function interface described here is the preferred user interface.

Many of the functions described here have a *termios_p* argument that is a pointer to a **termios** structure. This structure contains the following members:

```
tcflag_t    c_iflag;              /* input modes */
tcflag_t    c_oflag;              /* output modes */
tcflag_t    c_cflag;              /* control modes */
tcflag_t    c_lflag;              /* local modes */
cc_t        c_cc[NCCS];           /* control chars */
```

These structure members are described in detail in **termio**(4).

tcgetattr( ) gets the parameters associated with the object referred by *fd* and stores them in the **termios** structure referenced by *termios_p*. This function may be invoked from a background process; however, the terminal attributes may be subsequently changed by a foreground process.

tcsetattr( ) sets the parameters associated with the terminal (unless support is required from the underlying hardware that is not available) from the **termios** structure referred to by *termios_p* as follows:

- If *optional_actions* is **TCSANOW**, the change occurs immediately.
- If *optional_actions* is **TCSADRAIN**, the change occurs after all output written to *fd* has been transmitted. This function should be used when changing parameters that affect output.
- If *optional_actions* is **TCSAFLUSH**, the change occurs after all output written to the object referred by *fd* has been transmitted, and all input that has been received but not read will be discarded before the change is made.

The symbolic constants for the values of *optional_actions* are defined in **<sys/termios.h>**.

If the terminal is using asynchronous serial data transmission, **tcsendbreak**( ) transmits a continuous stream of zero-valued bits for a specific duration. If *duration* is zero, it transmits zero-valued bits for at least 0.25 seconds, and not more that 0.5 seconds. If *duration* is not zero, it sends zero-valued bits for *duration***N* seconds, where *N* is at least 0.25, and not more than 0.5.

If the terminal is not using asynchronous serial data transmission, **tcsendbreak**( ) returns without taking any action.

tcdrain( ) waits until all output written to the object referred to by *fd* has been transmitted.

tcflush( ) discards data written to the object referred to by *fd* but not transmitted, or data received but not read, depending on the value of *queue_selector*:

- If *queue_selector* is **TCIFLUSH,** it flushes data received but not read.
- If *queue_selector* is **TCOFLUSH,** it flushes data written but not transmitted.
- If *queue_selector* is **TCIOFLUSH,** it flushes both data received but not read, and data written but not transmitted.

The symbolic constants for the values of *queue_selector* and *action* are defined in **termios.h**.

The default on open of a terminal file is that neither its input nor its output is suspended.

tcflow( ) suspends transmission or reception of data on the object referred to by *fd*, depending on the value of *actions*:

- If action is **TCOOFF,** it suspends output.
- If action is **TCOON,** it restarts suspended output.
- If action is **TCIOFF,** the system transmits a STOP character, which stops the terminal device from transmitting data to the system. (See **termio**(4).)
- If action is **TCION,** the system transmits a START character, which starts the terminal device transmitting data to the system. (See **termio**(4).)

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

The input and output baud rates are stored in the **termios** structure. The values shown in the table are supported. The names in this table are defined in **termios.h**

Name	Description	Name	Description
B0	Hang up	**B600**	600 baud
B50	50 baud	**B1200**	1200 baud
B75	75 baud	**B1800**	1800 baud
B110	110 baud	**B2400**	2400 baud
B134	134.5 baud	**B4800**	4800 baud
B150	150 baud	**B9600**	9600 baud
B200	200 baud	**B19200**	19200 baud
B300	300 baud	**B38400**	38400 baud

cfgetospeed( ) returns the output baud rate stored in the **termios** structure pointed to by *termios_p*.

cfsetospeed( ) sets the output baud rate stored in the **termios** structure pointed to by *termios_p* to *speed*. The zero baud rate, **B0**, is used to terminate the connection. If **B0** is specified, the modem control lines shall no longer be asserted. Normally, this will disconnect the line.

If the input baud rate is set to zero, the input baud rate will be specified by the value of the output baud rate.

cfgetispeed( ) returns the input baud rate stored in the **termios** structure.

cfsetispeed( ) sets the input baud rate stored in the **termios** structure to *speed*.

RETURN VALUES

cfgetispeed( ) returns the input baud rate stored in the **termios** structure.

cfgetospeed( ) returns the output baud rate stored in the **termios** structure.

cfsetispeed( ) and **cfsetospeed( )** return:

0 on success.

−1 on failure and sets **errno** to indicate the error.

All other functions return:

0 on success.

−1 on failure and set **errno** to indicate the error.

ERRORS

EBADF The *fd* argument is not a valid file descriptor.

ENOTTY The file associated with *fd* is not a terminal.

tcsetattr( ) may set **errno** to:

EINVAL The *optional_actions* argument is not a proper value.

An attempt was made to change an attribute represented in the **termios** structure to an unsupported value.

tcsendbreak( ) may set **errno** to:

EINVAL The device does not support **tcsendbreak( )**.

tcdrain( ) may set **errno** to:

EINTR A signal interrupted **tcdrain( )**.

EINVAL The device does not support **tcdrain( )**.

tcflush( ) may set **errno** to:

EINVAL The device does not support **tcflush( )**.

The *queue_selector* argument is not a proper value.

tcflow( ) may set **errno** to:

EINVAL   The device does not support **tcflow( )**.

The *action* argument is not a proper value.

tcsetattr( ) may set **errno** to:

EAGAIN   There is insufficient memory available to copy in the arguments.

EBADF   *fd* is not a valid descriptor.

EFAULT   Some part of the structure pointed to by *termios_p* is outside the process's allocated address space.

EINVAL   *optional_actions* is not valid.

EIO   The calling process is a background process.

ENOTTY   *fd* does not refer to a terminal device.

ENXIO   The terminal referred to by *fd* is hung up.

cfsetispeed( ) and **cfsetospeed( )** may set **errno** to:

EINVAL   *speed* is greater than B38400 or less than 0.

SEE ALSO

setpgid(2V), **setsid**(2V), **termio**(4)

NAME

time, ftime – get date and time

SYNOPSIS

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

time_t time(tloc)
time_t *tloc;

#include <sys/timeb.h>

int ftime(tp)
struct timeb *tp;
```

DESCRIPTION

time( ) returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds.

If *tloc* is non-NULL, the return value is also stored in the location to which *tloc* points.

ftime( ) fills in a structure pointed to by *tp*, as defined in **<sys/timeb.h>**:

```
struct timeb
{
        time_t   time;
        unsigned short millitm;
        short    timezone;
        short    dstflag;
};
```

The structure contains the time since the epoch in seconds, up to 1000 milliseconds of more-precise interval, the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

RETURN VALUES

time( ) returns the value of time on success. On failure, it returns **(time_t)** –1.

On success, **ftime**( ) returns no useful value. On failure, it returns –1.

SEE ALSO

date(1V), **gettimeofday**(2), **ctime**(3V)

NAME

times – get process times

SYNOPSIS

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

int times(buffer)
struct tms *buffer;
```

SYSTEM V SYNOPSIS

```
clock_t times(buffer)
struct tms *buffer;
```

DESCRIPTION

This interface is obsoleted by **getrusage**(2).

times( ) returns time-accounting information for the current process and for the terminated child processes of the current process. All times are in 1/HZ seconds, where HZ is 60.

buffer points to the following structure:

```
struct tms {
        clock_t tms_utime;            /* user time */
        clock_t tms_stime;            /* system time */
        clock_t tms_cutime;           /* user time, children */
        clock_t tms_cstime;           /* system time, children */
};
```

This information comes from the calling process and each of its terminated child processes for which it has executed a **wait**(2V).

tms_utime is the CPU time used while executing instructions in the user space of the calling process.

tms_stime is the CPU time used by the system on behalf of the calling process.

tms_cutime is the sum of the **tms_utimes** and **tms_cutimes** of the child processes.

tms_cstime is the sum of the **tms_stimes** and **tms_cstimes** of the child processes.

RETURN VALUES

times( ) returns:

0 on success.

–1 on failure.

SYSTEM V RETURN VALUES

Upon successful completion, **times**( ) returns the elapsed real time, in 60ths of a second, since an arbitrary point in the past. This point does not change from one invocation of **times**( ) to another within the same process. On failure, **times**( ) returns **(clock_t)** –1.

SEE ALSO

time(1V), **getrusage**(2), **wait**(2V), **time**(3V)

NAME

timezone – get time zone name given offset from GMT

SYNOPSIS

char *timezone(zone, dst)

DESCRIPTION

timezone( ) attempts to return the name of the time zone associated with its first argument, which is measured in minutes westward from Greenwich. If the second argument is 0, the standard name is used, otherwise the Daylight Savings Time version. If the required name does not appear in a table built into the routine, the difference from GMT is produced; for instance, in Afghanistan '**timezone(–(60*4+30), 0)**' is appropriate because it is 4:30 ahead of GMT and the string **GMT+4:30** is produced.

Note: the offset westward from Greenwich and an indication of whether Daylight Savings Time is in effect may not be sufficient to determine the name of the time zone, as the name may differ between different locations in the same time zone. Instead of using **timezone**( ) to determine the name of the time zone for a given time, that time should be converted to a '**struct tm**' using **localtime**( ) (see **ctime**(3V)) and the *tm_zone* field of that structure should be used. **timezone**( ) is retained for compatibility with existing programs.

SEE ALSO

ctime(3V)

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**(3), and a NULL pointer is returned. The file will automatically be deleted when the process using it terminates. The file is opened for update ("w+").

SEE ALSO

creat(2V), **unlink**(2V), **fopen**(3V), **mktemp**(3), **perror**(3), **tmpnam**(3S)

NAME

tmpnam, tempnam – create a name for a temporary file

SYNOPSIS

```
#include <stdio.h>

char *tmpnam (s)
char *s;

char *tempnam (dir, pfx)
char *dir, *pfx;
```

DESCRIPTION

These functions generate file names that can safely be used for a temporary file.

tmpnam() always generates a file name using the path-prefix defined as **P_tmpdir** in the **<stdio.h>** header file. If *s* is NULL, **tmpnam()** leaves its result in an internal static area and returns a pointer to that area. The next call to **tmpnam()** will destroy the contents of the area. If *s* is not NULL, it is assumed to be the address of an array of at least **L_tmpnam** bytes, where **L_tmpnam** is a constant defined in **<stdio.h>**; **tmpnam()** places its result in that array and returns *s*.

tempnam() allows the user to control the choice of a directory. The argument **dir** points to the name of the directory in which the file is to be created. If **dir** is NULL or points to a string which is not a name for an appropriate directory, the path-prefix defined as **P_tmpdir** in the **<stdio.h>** header file is used. If that directory is not accessible, **/tmp** will be used as a last resort. This entire sequence can be up-staged by providing an environment variable **TMPDIR** in the user's environment, whose value is the name of the desired temporary-file directory.

Many applications prefer their temporary files to have certain favorite initial letter sequences in their names. Use the *pfx* argument for this. This argument may be NULL or point to a string of up to five characters to be used as the first few characters of the temporary-file name.

tempnam() uses **malloc()** 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**(3V)). If **tempnam()** cannot return the expected result for any reason, that is, **malloc()** failed, or none of the above mentioned attempts to find an appropriate directory was successful, a NULL pointer will be returned.

NOTES

These functions generate a different file name each time they are called.

Files created using these functions and either **fopen()** or **creat()** are temporary only in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to use **unlink**(2V) to remove the file when its use is ended.

SEE ALSO

creat(2V), **unlink**(2V), **fopen**(3V), **malloc**(3V), **mktemp**(3), **tmpfile**(3S)

BUGS

If called more than 17,576 times in a single process, these functions will start recycling previously used names.

Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using these functions or **mktemp()**, and the file names are chosen so as to render duplication by other means unlikely.

NAME

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 will be set to point to the datum which will be at the root of the new tree.

Like **tsearch()**, **tfind()** will search for a datum in the tree, returning a pointer to it if found. However, if it is not found, **tfind()** will return a NULL pointer. The arguments for **tfind()** are the same as for **tsearch()**.

tdelete() deletes a node from a binary search tree. The arguments are the same as for **tsearch()**. The variable pointed to by *rootp* will be changed if the deleted node was the root of the tree. **tdelete()** returns a pointer to the parent of the deleted node, or a NULL pointer if the node is not found.

twalk() traverses a binary search tree. *root* is the root of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) *action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The second argument is a value from an enumeration data type **typedef enum { preorder, postorder, endorder, leaf } VISIT**; (defined in the **<search.h>** header file), depending on whether this is the first, second or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level zero.

The pointers to the key and the root of the tree should be of type pointer-to-element, and cast to type pointer-to-pointer-to-character. Similarly, although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

EXAMPLES

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>

void twalk();
char *tsearch();

struct node {            /* pointers to these are stored in the tree */
        char *string;
        int count;
};

#define MAXNODES      12
#define MAXSTRING     100
#define MINSTRING     3               /* char, newline, eos */

char string_space[MAXSTRING];         /* space to store strings */
struct node node_space[MAXNODES];     /* nodes to store */
struct node *root = NULL;             /* this points to the root */

main()
{
        char *strptr = string_space;
        int maxstrlen = MAXSTRING;
        struct node *nodeptr = node_space;
        int node_compare();
        void print_node();
        struct node **found;
        int length;

        while (fgets(strptr, maxstrlen, stdin) != NULL) {
                /* remove the trailing newline */
                length = strlen(strptr);
                strptr[length-1] = 0;
                /* set node */
                nodeptr->string = strptr;
                /* locate node into the tree */
                found = (struct node **)
                    tsearch((char *) nodeptr, (char **) &root, node_compare);
                /* bump the count */
                (*found)->count++;

                if (*found == nodeptr) {
                        /* node was inserted, so get a new one */
                        strptr += length;
                        maxstrlen -= length;
                        if (maxstrlen < MINSTRING)
                                break;
                        if (++nodeptr >= &node_space[MAXNODES])
                                break;
                }
        }
        twalk((char *)root, print_node);
}
```

```
/*
     This routine compares two nodes, based on an
     alphabetical ordering of the string field.
*/

int node_compare(node1, node2)
        struct node *node1, *node2;
{
        return strcmp(node1->string, node2->string);
}

/* Print out nodes in alphabetical order */
/*ARGSUSED2*/
void
print_node(node, order, level)
        struct node **node;
        VISIT order;
        int level;
{
        if (order == postorder || order == leaf) {
                (void) printf("string = %20s,  count = %d0,
                    (*node)->string, (*node)->count);
        }
}
```

SEE ALSO

bsearch(3), **hsearch(3)**, **lsearch(3)**

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 **tsearch( )**, **tfind( )** and **tdelete( )** if *rootp* is NULL on entry.

If the datum is found, both **tsearch( )** and **tfind( )** return a pointer to it. If not, **tfind( )** returns NULL, and **tsearch( )** returns a pointer to the inserted item.

WARNINGS

The *root* argument to **twalk( )** is one level of indirection less than the *rootp* arguments to **tsearch( )** and **tdelete( )**.

There are two nomenclatures used to refer to the order in which tree nodes are visited. **tsearch( )** uses preorder, postorder and endorder to respectively refer to 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.

BUGS

If the calling function alters the pointer to the root, results are unpredictable.

NAME

ttyname, isatty – find name of a terminal

SYNOPSIS

```
char *ttyname(fd)
int fd;

int isatty(fd)
int fd;
```

DESCRIPTION

ttyname() returns a pointer to the null-terminated path name of the terminal device associated with file descriptor *fd*.

isatty() returns 1 if *fd* is associated with a terminal device, 0 otherwise.

FILES

/dev/*

SEE ALSO

ctermid(3V), **ioctl**(2), **ttytab**(5)

RETURN VALUES

On success, **ttyname()** returns a pointer to the terminal device. If *fd* does not describe a terminal device in directory **/dev**, **ttyname()** returns NULL.

isatty() returns 1 if *fd* is associated with a terminal device. It returns 0 otherwise.

BUGS

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

NAME

ttyslot – find the slot in the utmp file of the current process

SYNOPSIS

int ttyslot( )

DESCRIPTION

ttyslot( ) returns the index of the current user's entry in **/etc/utmp**. This is accomplished by actually scanning the file **/etc/ttytab** for the name of the terminal associated with the standard input, the standard output, or the error output (0, 1 or 2).

RETURN VALUES

On success, **ttyslot**() returns the index of the current user's entry in **/etc/utmp**. 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, **ttyslot**( ) returns 0.

SYSTEM V RETURN VALUES

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, **ttyslot**( ) returns –1.

FILES

/etc/ttytab
/etc/utmp

NAME

ualarm – schedule signal after interval in microseconds

SYNOPSIS

```
unsigned ualarm(value, interval)
unsigned value;
unsigned interval;
```

DESCRIPTION

This is a simplified interface to setitimer() (see **getitimer**(2)).

ualarm() sends signal **SIGALRM**, see **signal**(3V), to the invoking process in a number of microseconds given by the *value* argument. Unless caught or ignored, the signal terminates the process.

If the *interval* argument is non-zero, the **SIGALRM** signal will be sent to the process every *interval* microseconds after the timer expires (for instance, after *value* microseconds have passed).

Because of scheduling delays, resumption of execution of when the signal is caught may be delayed an arbitrary amount. The longest specifiable delay time is 2147483647 microseconds.

The return value is the amount of time previously remaining in the alarm clock.

SEE ALSO

getitimer(2), **sigpause**(2V), **sigvec**(2), **alarm**(3V), **signal**(3V), **sleep**(3V), **usleep**(3)

NAME

ulimit – get and set user limits

SYNOPSIS

```
long ulimit(cmd, newlimit)
int cmd;
long newlimit;
```

DESCRIPTION

This function is included for System V compatibility.

This routine provides for control over process limits. The *cmd* values available are:

1 Get the process's file size limit. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.

2 Set the process's file size limit to the value of *newlimit*. Any process may decrease this limit, but only a process with an effective user ID of super-user may increase the limit. **ulimit**() will fail and the limit will be unchanged if a process with an effective user ID other than the super-user attempts to increase its file size limit.

3 Get the maximum possible break value. See **brk**(2).

4 Get the size of the process' file descriptor table, as returned by **getdtablesize**(2).

RETURN VALUE

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

ERRORS

EPERM A user other than the super-user attempted to increase the file size limit.

SEE ALSO

brk(2), **getdtablesize**(2), **getrlimit**(2), **write**(2V)

NAME

ungetc – push character back into input stream

SYNOPSIS

#include <stdio.h>

ungetc(c, stream)
FILE *stream;

DESCRIPTION

ungetc( ) pushes the character *c* back onto an input stream. That character will be returned by the next **getc( )** call on that stream. **ungetc( )** returns *c*, and leaves the file stream unchanged.

One character of pushback is guaranteed provided something has been read from the stream and the stream is actually buffered. In the case that stream is **stdin**, one character may be pushed back onto the buffer without a previous read statement.

If *c* equals EOF, **ungetc( )** does nothing to the buffer and returns EOF.

An **fseek**(3S) erases all memory of pushed back characters.

SEE ALSO

fseek(3S), **getc**(3V), **setbuf**(3V)

DIAGNOSTICS

ungetc( ) returns EOF if it cannot push a character back.

NAME

usleep – suspend execution for interval in microseconds

SYNOPSIS

```
usleep(useconds)
unsigned useconds;
```

DESCRIPTION

Suspend the current process for the number of microseconds specified by the argument. The actual suspension time may be an arbitrary amount longer because of other activity in the system, or because of the time spent in processing the call.

The routine is implemented by setting an interval timer and pausing until it occurs. The previous state of this timer is saved and restored. If the sleep time exceeds the time to the expiration of the previous timer, the process sleeps only until the signal would have occurred, and the signal is sent a short time later.

This routine is implemented using **setitimer**( ) (see **getitimer**(2)); it requires eight system calls each time it is invoked. A similar but less compatible function can be obtained with a single **select**(2); it would not restart after signals, but would not interfere with other uses of **setitimer**.

SEE ALSO

getitimer(2), **sigpause**(2V), **alarm**(3V), **sleep**(3V), **ualarm**(3)

NAME

utime – set file times

SYNOPSIS

```
#include <utime.h>

int utime(path, times)
char *path;
struct utimbuf *times;
```

DESCRIPTION

utime() sets the access and modification times of the file named by *path*.

If *times* is NULL, the access and modification times are set to the current time. The effective user ID (UID) of the calling process must match the owner of the file or the process must have write permission for the file to use **utime()** in this manner.

If *times* is not NULL, it is assumed to point to a **utimbuf** structure, defined in **<utime.h>** as:

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

The access time is set to the value of the first member, and the modification time is set to the value of the second member. The times contained in this structure are measured in seconds since 00:00:00 GMT Jan 1, 1970. Only the owner of the file or the super-user may use **utime()** in this manner.

Upon successful completion, **utime()** marks for update the *st_ctime* field of the file.

RETURN VALUES

utime() returns:

0 on success.

–1 on failure and sets **errno** to indicate the error.

ERRORS

EACCES Search permission is denied for a component of the path prefix of *path*.

EACCES The effective user ID is not super-user and not the owner of the file, write permission is denied for the file, and *times* is NULL.

EFAULT *path* or *times* points outside the process's allocated address space.

EIO An I/O error occurred while reading from or writing to the file system.

ELOOP Too many symbolic links were encountered in translating *path*.

ENAMETOOLONG The length of *path* exceeds {PATH_MAX}.

A pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).

ENOENT The file referred to by *path* does not exist.

ENOTDIR A component of the path prefix of *path* is not a directory.

EPERM The effective user ID of the process is not super-user and not the owner of the file, and *times* is not NULL.

EROFS The file system containing the file is mounted read-only.

SYSTEM V ERRORS

In addition to the above, the following may also occur:

ENOENT *path* points to an empty string.

SEE ALSO

pathconf(2V), **stat**(2V), **utimes**(2)

NAME

values – machine-dependent values

SYNOPSIS

#include <values.h>

DESCRIPTION

This file contains a set of manifest constants, conditionally defined for particular processor architectures.

The model assumed for integers is binary representation (one's or two's complement), where the sign is represented by the value of the high-order bit.

BITS(*type*) The number of bits in a specified type (for instance, int).

HIBITS The value of a short integer with only the high-order bit set (in most implementations, 0x8000).

HIBITL The value of a long integer with only the high-order bit set (in most implementations, 0x80000000).

HIBITI The value of a regular integer with only the high-order bit set (usually the same as HIBITS or HIBITL).

MAXSHORT The maximum value of a signed short integer (in most implementations, 0x7FFF ≡ 32767).

MAXLONG The maximum value of a signed long integer (in most implementations, 0x7FFFFFFF ≡ 2147483647).

MAXINT The maximum value of a signed regular integer (usually the same as MAXSHORT or MAXLONG).

MAXFLOAT

LN_MAXFLOAT The maximum value of a single-precision floating-point number, and its natural logarithm.

MAXDOUBLE

LN_MAXDOUBLE The maximum value of a double-precision floating-point number, and its natural logarithm.

MINFLOAT

LN_MINFLOAT The minimum positive value of a single-precision floating-point number, and its natural logarithm.

MINDOUBLE

LN_MINDOUBLE The minimum positive value of a double-precision floating-point number, and its natural logarithm.

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.

SEE ALSO

intro(3), **intro**(3M)

NAME

varargs – handle variable argument list

SYNOPSIS

#include <varargs.h>

function(va_alist) va_dcl

va_list pvar;

va_start(pvar);

f = va_arg(pvar, type);

va_end(pvar);

DESCRIPTION

This set of macros provides a means of writing portable procedures that accept variable argument lists. Routines having variable argument lists (such as **printf**(3V)) but do not use **varargs**( ) are inherently nonportable, since different machines use different argument passing conventions. Routines with variable arguments lists *must* use **varargs**( ) functions in order to run correctly on Sun-4 systems.

va_alist( ) is used in a function header to declare a variable argument list.

va_dcl( ) is a declaration for **va_alist**( ). No semicolon should follow **va_dcl**( ).

va_list( ) is a type defined for the variable used to traverse the list. One such variable must always be declared.

va_start(*pvar*) is called to initialize *pvar* to the beginning of the list.

va_arg(*pvar*, *type*) will return the next argument in the list pointed to by *pvar*. The parameter *type* is a type name such that the type of a pointer to an object that has the specified type can be obtained simply by appending a * to *type*. If *type* disagrees with the type of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined.

In standard C, arguments that are **char** or **short** are converted to **int** and should be accessed as **int**, arguments that are **unsigned char** or **unsigned short** are converted to **unsigned int** and should be accessed as **unsigned int**, and arguments that are **float** are converted to **double** and should be accessed as **double**. Different types can be mixed, but it is up to the routine to know what type of argument is expected, since it cannot be determined at runtime.

va_end(*pvar*) is used to finish up.

Multiple traversals, each bracketed by **va_start**( ) ... **va_end**( ), are possible.

va_alist( ) must encompass the entire arguments list. This insures that a **#define** statement can be used to redefine or expand its value.

The argument list (or its remainder) can be passed to another function using a pointer to a variable of type **va_list**( ) — in which case a call to **va_arg**( ) in the subroutine advances the argument-list pointer with respect to the caller as well.

EXAMPLE

This example is a possible implementation of **execl**(3V).

```
#include <varargs.h>
#define MAXARGS         100

/*      execl is called by
 *      execl(file, arg1, arg2, ..., (char *)0);
 */
execl (va_alist)
va_dcl
{
        va_list ap;
        char *file;
        char *args[MAXARGS];
        int argno = 0;

        va_start (ap);
        file = va_arg(ap, char *);
        while ((args[argno++] = va_arg(ap, char *)) != (char *)0)
                ;
        va_end (ap);
        return execv(file, args);
}
```

SEE ALSO

execl(3V), **printf**(3V)

BUGS

It is up to the calling routine to specify how many arguments there are, since it is not possible to determine this from the stack frame. For example, **execl**() is passed a zero pointer to signal the end of the list. **printf**() can tell how many arguments are supposed to be there by the format.

The macros **va_start**() and **va_end**() may be arbitrarily complex; for example, **va_start**() might contain an opening brace, which is closed by a matching brace in **va_end**(). Thus, they should only be used where they could be placed within a single complex statement.

NAME

vlimit – control maximum system resource consumption

SYNOPSIS

#include <sys/vlimit.h>

vlimit(resource, value) int resource, value;

DESCRIPTION

This facility is superseded by getrlimit(2).

Limits the consumption by the current process and each process it creates to not individually exceed *value* on the specified resource. If *value* is specified as –1, then the current limit is returned and the limit is unchanged. The resources which are currently controllable are:

LIM_NORAISE A pseudo-limit; if set non-zero then the limits may not be raised. Only the super-user may remove the *noraise* restriction.

LIM_CPU the maximum number of CPU-seconds to be used by each process

LIM_FSIZE the largest single file which can be created

LIM_DATA the maximum growth of the data+stack region using **sbrk**() (see **brk**(2)) beyond the end of the program text

LIM_STACK the maximum size of the automatically-extended stack region

LIM_CORE the size of the largest core dump that will be created.

LIM_MAXRSS a soft limit for the amount of physical memory (in bytes) to be given to the program. If memory is tight, the system will prefer to take memory from processes which are exceeding their declared **LIM_MAXRSS.**

Because this information is stored in the per-process information this system call must be executed directly by the shell if it is to affect all future processes created by the shell; *limit* is thus a built-in command to **csh**(1).

The system refuses to extend the data or stack space when the limits would be exceeded in the normal way; a *break* call fails if the data space limit is reached, or the process is killed when the stack limit is reached (since the stack cannot be extended, there is no way to send a signal!).

A file I/O operation which would create a file which is too large will cause a signal **SIGXFSZ** to be generated, this normally terminates the process, but may be caught. When the cpu time limit is exceeded, a signal **SIGXCPU** is sent to the offending process; to allow it time to process the signal it is given 5 seconds grace by raising the CPU time limit.

SEE ALSO

csh(1), **sh**(1), **brk**(2)

BUGS

If **LIM_NORAISE** is set, then no grace should be given when the CPU time limit is exceeded.

There should be *limit* and *unlimit* commands in **sh**(1) as well as in **csh**(1).

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;

char *vsprintf(s, format, ap)
char *s, *format;
va_list ap;
```

SYSTEM V SYNOPSIS

```
int vsprintf(s, format, ap)
char *s, *format;
va_list ap;
```

DESCRIPTION

vprintf(), **vfprintf()**, and **vsprintf()** are the same as **printf**(3V), **fprintf()**, and **sprintf()** (see **printf**(3V)) respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by **varargs**(3).

RETURN VALUES

On success, **vprintf()** and **vfprintf()** return the number of characters transmitted, excluding the null character. On failure, they return EOF.

vsprintf() returns *s*.

SYSTEM V RETURN VALUES

vsprintf() returns the number of characters transmitted, excluding the null character.

EXAMPLES

The following demonstrates how **vfprintf()** could be used to write an error routine.

```
#include <stdio.h>
#include <varargs.h>
...
        /*  error should be called like:
         *       error(function_name, format, arg1, arg2...);
         *  Note: function_name and format cannot be declared
         *  separately because of the definition of varargs.
         */

/*VARARGS0*/
void
error (va_alist)
        va_dcl
{
        va_list args;
        char *fmt;

        va_start(args);
                /* print 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(3V), varargs(3)

NAME

vsyslog – log message with a varargs argument list

SYNOPSIS

```
#include <syslog.h>
#include <varargs.h>

int vsyslog(priority, message, ap)
char *message;
va_list ap;
```

DESCRIPTION

vsyslog() is the same as **syslog**(3) except that instead of being called with a variable number of arguments, it is called with an argument list as defined by **varargs**(3).

EXAMPLE

The following demonstrates how **vsyslog()** could be used to write an error routine.

```
#include <syslog.h>
#include <varargs.h>
...
        /*  error should be called like:
         *      error(pri, function_name, format, arg1, arg2...);
         *  Note that pri, function_name, and format cannot be declared
         *  separately because of the definition of varargs.
         */

/*VARARGS0*/
void
error(va_alist)
        va_dcl;
{
        va_list args;
        int pri;
        char *message;

        va_start(args);
        pri = va_arg(args, int);
                /* log name of function causing error */
        (void) syslog(pri, "ERROR in %s", va_arg(args, char *));
        message = va_arg(args, char *);
                /* log remainder of message */
        (void) vsyslog(pri, fmt, args);
        va_end(args);
        (void) abort();
}
```

SEE ALSO

syslog(3), **varargs**(3)

NAME

vtimes – get information about resource utilization

SYNOPSIS

```
vtimes(par_vm, ch_vm)
struct vtimes *par_vm, *ch_vm;
```

DESCRIPTION

Note: this facility is superseded by **getrusage(2)**.

vtimes() returns accounting information for the current process and for the terminated child processes of the current process. Either *par_vm* or *ch_vm* or both may be 0, in which case only the information for the pointers which are non-zero is returned.

After the call, each buffer contains information as defined by the contents of the include file **<sys/vtimes.h>**:

```
struct vtimes {
        int      vm_utime;              /* user time (*HZ) */
        int      vm_stime;              /* system time (*HZ) */
        /* divide next two by utime+stime to get averages */
        unsigned vm_idsrss;             /* integral of d+s rss */
        unsigned vm_ixrss;              /* integral of text rss */
        int      vm_maxrss;             /* maximum rss */
        int      vm_majflt;             /* major page faults */
        int      vm_minflt;             /* minor page faults */
        int      vm_nswap;              /* number of swaps */
        int      vm_inblk;              /* block reads */
        int      vm_oublk;              /* block writes */
};
```

The **vm_utime** and **vm_stime** fields give the user and system time respectively in 60ths of a second (or 50ths if that is the frequency of wall current in your locality.) The **vm_idrss** and **vm_ixrss** measure memory usage. They are computed by integrating the number of memory pages in use each over cpu time. They are reported as though computed discretely, adding the current memory usage (in 512 byte pages) each time the clock ticks. If a process used 5 core pages over 1 cpu-second for its data and stack, then **vm_idsrss** would have the value 5*60, where **vm_utime+vm_stime** would be the 60. **vm_idsrss** integrates data and stack segment usage, while **vm_ixrss** integrates text segment usage. **vm_maxrss** reports the maximum instantaneous sum of the text+data+stack core-resident page count.

The **vm_majflt** field gives the number of page faults which resulted in disk activity; the **vm_minflt** field gives the number of page faults incurred in simulation of reference bits; **vm_nswap** is the number of swaps which occurred. The number of file system input/output events are reported in **vm_inblk** and **vm_oublk** These numbers account only for real I/O; data supplied by the caching mechanism is charged only to the first process to read or write the data.

SEE ALSO

getrusage(2), **wait(2V)**

NAME

xdr – library routines for external data representation

SYNOPSIS AND DESCRIPTION

XDR routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Data for remote procedure calls (RPC) are encoded and decoded using these routines. See **rpc**(3N).

All XDR routines require the header **<rpc/xdr.h>** to be included.

The XDR routines have been grouped by usage on the following man pages.

xdr_admin(3N) Library routines for managing the XDR stream. The routines documented on this page include:

xdr_getpos( )
xdr_inline( )
xdrrec_endofrecord( )
xdrrec_eof( )
xdrrec_readbytes( )
xdrrec_skiprecord( )
xdr_setpos( )

xdr_complex(3N) Library routines for translating complex data types into their external data representation. The routines documented on this page include:

xdr_array( )
xdr_bytes( )
xdr_opaque( )
xdr_pointer( )
xdr_reference( )
xdr_string( )
xdr_union( )
xdr_vector( )
xdr_wrapstring( )

xdr_create(3N) Library routines for creating XDR streams. The routines documented on this page include:

xdr_destroy( )
xdrmem_create( )
xdrrec_create( )
xdrstdio_create( )

xdr_simple(3N) Library routines for translating simple data types into their external data representation. The routines documented on this page include:

xdr_bool( )
xdr_char( )
xdr_double( )
xdr_enum( )
xdr_float( )
xdr_free( )
xdr_int( )
xdr_long( )
xdr_short( )
xdr_u_char( )
xdr_u_int( )
xdr_u_long( )
xdr_u_short( )
xdr_void( )

SEE ALSO

rpc(3N), **xdr_admin**(3N), **xdr_complex**(3N), **xdr_create**(3N), **xdr_simple**(3N)

Network Programming

NAME

xdr_getpos, xdr_inline, xdrrec_endofrecord, xdrrec_eof, xdrrec_readbytes, xdrrec_skiprecord, xdr_setpos – library routines for management of the XDR stream

DESCRIPTION

XDR library routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Protocols such as remote procedure calls (RPC) use these routines to describe the format of the data.

These routines deal specifically with the management of the XDR stream.

Routines

The **XDR** data structure is defined in the RPC/XDR Library Definitions of the *Network Programming*.

#include <rpc/xdr.h>

u_int xdr_getpos(xdrs)
XDR *xdrs;

Invoke the get-position routine associated with the XDR stream, *xdrs*. The routine returns an unsigned integer, which indicates the position of the XDR byte stream. A desirable feature of XDR streams is that simple arithmetic works with this number, although the XDR stream instances need not guarantee this.

long * xdr_inline(xdrs, len)
XDR *xdrs;
int len;

Invoke the in-line routine associated with the XDR stream, *xdrs*. The routine returns a pointer to a contiguous piece of the stream's buffer; *len* is the byte length of the desired buffer. Note: A pointer is cast to **long ***.

Warning: **xdr_inline()** may return NULL if it cannot allocate a contiguous piece of a buffer. Therefore the behavior may vary among stream instances; it exists for the sake of efficiency.

bool_t xdrrec_endofrecord(xdrs, sendnow)
XDR *xdrs;
int sendnow;

This routine can be invoked only on streams created by **xdrrec_create()** (see **xdr_create**(3N)). The data in the output buffer is marked as a completed record, and the output buffer is optionally written out if *sendnow* is non-zero. This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdrrec_eof(xdrs)
XDR *xdrs;
int empty;

This routine can be invoked only on streams created by **xdrrec_create()** (see **xdr_create**(3N)). After consuming the rest of the current record in the stream, this routine returns TRUE if the stream has no more input, FALSE otherwise.

int xdrrec_readbytes(xdrs, addr, nbytes)
XDR *xdrs;
caddr_t addr;
u_int nbytes;

This routine can be invoked only on streams created by **xdrrec_create()** (see **xdr_create**(3N)). It attempts to read *nbytes* bytes from the XDR stream into the buffer pointed to by *addr*. On success it returns the number of bytes read. Returns –1 on failure. A return value of 0 indicates an end of record.

bool_t xdrrec_skiprecord(xdrs)
XDR *xdrs;

This routine can be invoked only on streams created by **xdrrec_create**() (see **xdr_create**(3N)). It tells the XDR implementation that the rest of the current record in the stream's input buffer should be discarded. This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_setpos(xdrs, pos)
XDR *xdrs;
u_int pos;

Invoke the set position routine associated with the XDR stream *xdrs*. The parameter *pos* is a position value obtained from **xdr_getpos**(). This routine returns 1 if the XDR stream could be repositioned, and 0 otherwise.

Warning: It is difficult to reposition some types of XDR streams, so this routine may fail with one type of stream and succeed with another.

SEE ALSO

xdr(3N), **xdr_complex**(3N), **xdr_create**(3N), **xdr_simple**(3N)

NAME

xdr_array, xdr_bytes, xdr_opaque, xdr_pointer, xdr_reference, xdr_string, xdr_union, xdr_vector, xdr_wrapstring – library routines for translating complex data types

DESCRIPTION

XDR library routines allow C programmers to describe complex data structures in a machine-independent fashion. Protocols such as remote procedure calls (RPC) use these routines to describe the format of the data.

Routines

The **XDR** data structure is defined in the RPC/XDR Library Definitions of the *Network Programming*.

#include <rpc/xdr.h>

```
bool_t xdr_array(xdrs, arrp, sizep, maxsize, elsize, elproc)
XDR *xdrs;
char **arrp;
u_int *sizep, maxsize, elsize;
xdrproc_t elproc;
```

A filter primitive that translates between a variable-length array and its corresponding external representations. The parameter *arrp* is the address of the pointer to the array, while *sizep* is the address of the element count of the array. This value is used by the filter while encoding and is set by it while decoding; the routine fails if the element count exceeds *maxsize*. The parameter *elsize* is the *sizeof* each of the array's elements, and *elproc* is an XDR filter that translates between the array elements' C form, and their external representation. This routine returns TRUE if it succeeds, FALSE otherwise.

```
bool_t xdr_bytes(xdrs, arrp, sizep, maxsize)
XDR *xdrs;
char **arrp;
u_int *sizep, maxsize;
```

A filter primitive that translates between an array of bytes and its external representation. It treats the array of bytes as opaque data. The parameter *arrp* is the address of the array of bytes. While decoding if **arrp* is NULL, then the necessary storage is allocated to hold the array. This storage can be freed by using **xdr_free( )** (see **xdr_simple**(3N)). *sizep* is the pointer to the actual length specifier for the array. This value is used by the filter while encoding and is set by it when decoding. *maxsize* is the maximum length of the array. The routine fails if the actual length of the array is greater than *maxsize* This routine returns TRUE if it succeeds, FALSE otherwise.

```
bool_t xdr_opaque(xdrs, cp, cnt)
XDR *xdrs;
char *cp;
u_int cnt;
```

A filter primitive that translates between fixed size opaque data and its external representation. The parameter *cp* is the address of the opaque object, and *cnt* is its size in bytes. This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_pointer(xdrs, objpp, objsize, objproc)
XDR *xdrs;
char **objpp;
u_int objsize;
xdrproc_t objproc;

Like **xdr_reference()** except that it serializes NULL pointers, whereas **xdr_reference()** does not. Thus, **xdr_pointer()** can represent recursive data structures, such as binary trees or linked lists. The parameter *objpp* is the address of the pointer; *objsize* is the *sizeof* the structure that **objpp* points to; and *objproc* is an XDR procedure that filters the structure between its C form and its external representation. This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_reference(xdrs, pp, size, proc)
XDR *xdrs;
char **pp;
u_int size;
xdrproc_t proc;

A primitive that provides pointer chasing within structures. The parameter *pp* is the address of the pointer; *size* is the *sizeof* the structure that **pp* points to; and *proc* is an XDR procedure that filters the structure between its C form and its external representation. This routine returns TRUE if it succeeds, FALSE otherwise.

Warning: This routine does not understand NULL pointers. Use **xdr_pointer()** instead.

bool_t xdr_string(xdrs, strp, maxsize)
XDR *xdrs;
char **strp;
u_int maxsize;

A filter primitive that translates between C strings and their corresponding external representations. The routine fails if the string being translated is longer than *maxsize*. *strp* is the address of the pointer to the string. While decoding if **strp* is NULL, then the necessary storage is allocated to hold this null-terminated string and **strp* is set to point to this. This storage can be freed by using **xdr_free()** (see **xdr_simple(3N)**). This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_union(xdrs, dscmp, unp, choices, defaultarm)
XDR *xdrs;
int *dscmp;
char *unp;
struct xdr_discrim *choices;
bool_t (*defaultarm) (); /* may be NULL */

A filter primitive that translates between a discriminated C **union** and its corresponding external representation. It first translates the discriminant of the union located at *dscmp*. This discriminant is always an **enum_t**. Next the union located at *unp* is translated. The parameter *choices* is a pointer to an array of **xdr_discrim** structures. Each structure contains an ordered pair of [*value*,*proc*]. If the union's discriminant is equal to any of the *value*s, then the associated *proc* is called to translate the union. The end of the **xdr_discrim** structure array is denoted by a NULL pointer. If the discriminant is not found in the *choices* array, then the *defaultarm* procedure is called (if it is not NULL). This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_vector(xdrs, arrp, size, elsize, elproc)
XDR *xdrs;
char *arrp;
u_int size, elsize;
xdrproc_t elproc;

A filter primitive that translates between fixed-length arrays and their corresponding external representations. The parameter *arrp* is the address of the array, while *size* is the element count of the array. The parameter *elsize* is the *sizeof* each of the array's elements, and *elproc* is an XDR filter that translates between the array elements' C form, and their external representation. This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_wrapstring(xdrs, strp)
XDR *xdrs;
char **strp;

A primitive that calls **xdr_string(xdrs, strp, MAXUNSIGNED);** where **MAXUNSIGNED** is the maximum value of an unsigned integer. **xdr_wrapstring()** is handy because the RPC package passes a maximum of two XDR routines as parameters, and **xdr_string()**, one of the most frequently used primitives, requires three. *strp* is the address of the pointer to the string. While decoding if **strp* is NULL, then the necessary storage is allocated to hold the null-terminated string and **strp* is set to point to this. This storage can be freed by using **xdr_free()** (see **xdr_simple**(3N)). This routine returns TRUE if it succeeds, FALSE otherwise.

SEE ALSO

xdr(3N), **xdr_admin**(3N), **xdr_create**(3N), **xdr_simple**(3N)

NAME

xdr_destroy, xdrmem_create, xdrrec_create, xdrstdio_create – library routines for external data representation stream creation

DESCRIPTION

XDR library routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Protocols such as remote procedure calls (RPC) use these routines to describe the format of the data.

These routines deal with the creation of XDR streams. XDR streams have to be created before any data can be translated into XDR format.

Routines

The **XDR**, **CLIENT**, and **SVCXPRT** data structures are defined in the RPC/XDR Library Definitions of the *Network Programming*.

#include <rpc/xdr.h>

void xdr_destroy(xdrs)
XDR *xdrs;

> Invoke the destroy routine associated with the XDR stream, *xdrs*. Destruction usually involves freeing private data structures associated with the stream. Using *xdrs* after invoking **xdr_destroy( )** is undefined.

void xdrmem_create(xdrs, addr, size, op)
XDR *xdrs;
char *addr;
u_int size;
enum xdr_op op;

> This routine initializes the XDR stream object pointed to by *xdrs*. The stream's data is written to, or read from, a chunk of memory at location *addr* whose length is no more than *size* bytes long. *size* should be a multiple of 4. The *op* determines the direction of the XDR stream (either **XDR_ENCODE**, **XDR_DECODE**, or **XDR_FREE**).

void xdrrec_create(xdrs, sendsz, recvsz, handle, readit, writeit)
XDR *xdrs;
u_int sendsz, recvsz;
char *handle;
int (*readit) (), (*writeit) ();

> This routine initializes the XDR stream object pointed to by *xdrs*. The stream's data is written to a buffer of size *sendsz*; a value of zero indicates the system should use a suitable default. The stream's data is read from a buffer of size *recvsz*; it too can be set to a suitable default by passing a zero value. When a stream's output buffer is full, *writeit* is called. Similarly, when a stream's input buffer is empty, *readit* is called. The behavior of these two routines is similar to **read**(2V) and **write**(2V), except that *handle* is passed to the former routines as the first parameter. Note: The XDR stream's *op* field must be set by the caller. *sendsz* and *recvsz* should be multiples of 4.
>
> Warning: This XDR stream implements an intermediate record stream. Therefore there are additional bytes in the stream to provide record boundary information.

void xdrstdio_create(xdrs, filep, op)
XDR *xdrs;
FILE *filep;
enum xdr_op op;

This routine initializes the XDR stream object pointed to by *xdrs*. The XDR stream data is written to, or read from, the Standard I/O stream *filep*. The parameter *op* determines the direction of the XDR stream (either **XDR_ENCODE**, **XDR_DECODE**, or **XDR_FREE**).

Warning: The destroy routine associated with such XDR streams calls **fflush()** on the *file* stream, but never **fclose**(3V).

SEE ALSO

read(2V), **write**(2V), **fclose**(3V), **xdr**(3N), **xdr_admin**(3N), **xdr_complex**(3N), **xdr_simple**(3N)

NAME

xdr_bool, xdr_char, xdr_double, xdr_enum, xdr_float, xdr_free, xdr_int, xdr_long, xdr_short, xdr_u_char, xdr_u_int, xdr_u_long, xdr_u_short, xdr_void – library routines for translating simple data types

DESCRIPTION

XDR library routines allow C programmers to describe simple data structures in a machine-independent fashion. Protocols such as remote procedure calls (RPC) use these routines to describe the format of the data.

These routines require the creation of XDR streams (see **xdr_create**(3N)).

Routines

The **XDR** data structure is defined in the RPC/XDR Library Definitions of the *Network Programming*.

#include <rpc/xdr.h>

bool_t xdr_bool(xdrs, bp)
XDR *xdrs;
bool_t *bp;

A filter primitive that translates between a boolean (C integer) and its external representation. When encoding data, this filter produces values of either one or zero. This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_char(xdrs, cp)
XDR *xdrs;
char *cp;

A filter primitive that translates between a C character and its external representation. This routine returns TRUE if it succeeds, FALSE otherwise.

Note: Encoded characters are not packed, and occupy 4 bytes each. For arrays of characters, it is worthwhile to consider **xdr_bytes**(), **xdr_opaque**() or **xdr_string**() , see **xdr_complex**(3N).

bool_t xdr_double(xdrs, dp)
XDR *xdrs;
double *dp;

A filter primitive that translates between a C **double** precision number and its external representation. This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_enum(xdrs, ep)
XDR *xdrs;
enum_t *ep;

A filter primitive that translates between a C **enum** (actually integer) and its external representation. This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_float(xdrs, fp)
XDR *xdrs;
float *fp;

A filter primitive that translates between a C **float** and its external representation. This routine returns TRUE if it succeeds, FALSE otherwise.

void xdr_free(proc, objp)
xdrproc_t proc;
char *objp;

Generic freeing routine. The first argument is the XDR routine for the object being freed. The second argument is a pointer to the object itself. Note: The pointer passed to this routine is *not* freed, but what it points to *is* freed, recursively such that objects pointed to are also freed for example, linked lists.

bool_t xdr_int(xdrs, ip)
XDR *xdrs;
int *ip;

A filter primitive that translates between a C **integer** and its external representation. This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_long(xdrs, lp)
XDR *xdrs;
long *lp;

A filter primitive that translates between a C **long** integer and its external representation. This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_short(xdrs, sp)
XDR *xdrs;
short *sp;

A filter primitive that translates between a C **short** integer and its external representation. This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_u_char(xdrs, ucp)
XDR *xdrs;
unsigned char *ucp;

A filter primitive that translates between an **unsigned** C character and its external representation. This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_u_int(xdrs, up)
XDR *xdrs;
unsigned *up;

A filter primitive that translates between a C **unsigned** integer and its external representation. This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_u_long(xdrs, ulp)
XDR *xdrs;
unsigned long *ulp;

A filter primitive that translates between a C **unsigned long** integer and its external representation. This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_u_short(xdrs, usp)
XDR *xdrs;
unsigned short *usp;

A filter primitive that translates between a C **unsigned short** integer and its external representation. This routine returns TRUE if it succeeds, FALSE otherwise.

bool_t xdr_void( )

This routine always returns TRUE. It may be passed to RPC routines that require a function parameter, where nothing is to be done.

SEE ALSO

xdr(3N), **xdr_admin**(3N), **xdr_complex**(3N), **xdr_create**(3N)

NAME

ypclnt, yp_get_default_domain, yp_bind, yp_unbind, yp_match, yp_first, yp_next, yp_all, yp_order, yp_master, yperr_string, ypprot_err – NIS client interface

SYNOPSIS AND DESCRIPTION

This package of functions provides an interface to the Network Information Service (NIS). The package can be loaded from the standard library, **/usr/lib/libc.a**. Refer to **ypfiles**(5) and **ypserv**(8) for an overview of the NIS name service, including the definitions of *map* and *domain*, and a description of the various servers, databases, and commands that comprise the NIS services.

All input parameters names begin with *in*. Output parameters begin with *out*. Output parameters of type **char **** should be addresses of uninitialized character pointers. Memory is allocated by the NIS client package using **malloc**(3V), and may be freed if the user code has no continuing need for it. For each *outkey* and *outval*, two extra bytes of memory are allocated at the end that contain NEWLINE and the **null** character, respectively, but these two bytes are not reflected in *outkeylen* or *outvallen*. *indomain* and *inmap* strings must not be empty and must be null-terminated. String parameters which are accompanied by a count parameter may not be NULL, but may point to null strings, with the count parameter indicating this. Counted strings need not be null-terminated.

All functions in this package of type *int* return 0 if they succeed, and a failure code (**YPERR**_*xxxx*) otherwise. Failure codes are described under DIAGNOSTICS below.

```
yp_bind (indomain);
char *indomain;
```

To use the NIS services, the client process must be "bound" to a NIS server that serves the appropriate domain using **yp_bind**(). Binding need not be done explicitly by user code; this is done automatically whenever a NIS lookup function is called. **yp_bind**() can be called directly for processes that make use of a backup strategy (for example, a local file) in cases when NIS services are not available.

```
void
yp_unbind (indomain)
char *indomain;
```

Each binding allocates (uses up) one client process socket descriptor; each bound domain costs one socket descriptor. However, multiple requests to the same domain use that same descriptor. **yp_unbind**() is available at the client interface for processes that explicitly manage their socket descriptors while accessing multiple domains. The call to **yp_unbind**() make the domain *unbound*, and free all per-process and per-node resources used to bind it.

If an RPC failure results upon use of a binding, that domain will be unbound automatically. At that point, the ypclnt layer will retry forever or until the operation succeeds, provided that **ypbind** is running, and either

a) the client process cannot bind a server for the proper domain, or

b) RPC requests to the server fail.

If an error is not RPC-related, or if **ypbind** is not running, or if a bound **ypserv** process returns any answer (success or failure), the ypclnt layer will return control to the user code, either with an error code, or a success code and any results.

yp_get_default_domain (outdomain);
char **outdomain;

The NIS lookup calls require a map name and a domain name, at minimum. It is assumed that the client process knows the name of the map of interest. Client processes should fetch the node's default domain by calling **yp_get_default_domain**( ), and use the returned *outdomain* as the *indomain* parameter to successive NIS calls.

yp_match(indomain, inmap, inkey, inkeylen, outval, outvallen)
char *indomain;
char *inmap;
char *inkey;
int inkeylen;
char **outval;
int *outvallen;

yp_match( ) returns the value associated with a passed key. This key must be exact; no pattern matching is available.

yp_first(indomain, inmap, outkey, outkeylen, outval, outvallen)
char *indomain;
char *inmap;
char **outkey;
int *outkeylen;
char **outval;
int *outvallen;

yp_first( ) returns the first key-value pair from the named map in the named domain.

yp_next(indomain, inmap, inkey, inkeylen, outkey, outkeylen, outval, outvallen);
char *indomain;
char *inmap;
char *inkey;
int inkeylen;
char **outkey;
int *outkeylen;
char **outval;
int *outvallen;

yp_next( ) returns the next key-value pair in a named map. The *inkey* parameter should be the *outkey* returned from an initial call to **yp_first**( ) (to get the second key-value pair) or the one returned from the nth call to **yp_next**( ) (to get the nth + second key-value pair).

The concept of first (and, for that matter, of next) is particular to the structure of the NIS map being processing; there is no relation in retrieval order to either the lexical order within any original (non-NIS) data base, or to any obvious numerical sorting order on the keys, values, or key-value pairs. The only ordering guarantee made is that if the **yp_first**( ) function is called on a particular map, and then the **yp_next**( ) function is repeatedly called on the same map at the same server until the call fails with a reason of **YPERR_NOMORE**, every entry in the data base will be seen exactly once. Further, if the same sequence of operations is performed on the same map at the same server, the entries will be seen in the same order.

Under conditions of heavy server load or server failure, it is possible for the domain to become unbound, then bound once again (perhaps to a different server) while a client is running. This can cause a break in one of the enumeration rules; specific entries may be seen twice by the client, or not at all. This approach protects the client from error messages that would otherwise be returned in the midst of the enumeration. The next paragraph describes a better solution to enumerating all entries in a map.

```
yp_all(indomain, inmap, incallback);
char *indomain;
char *inmap;
struct ypall_callback *incallback;
```

yp_all() provides a way to transfer an entire map from server to client in a single request using TCP (rather than UDP as with other functions in this package). The entire transaction take place as a single RPC request and response. You can use **yp_all()** just like any other NIS procedure, identify the map in the normal manner, and supply the name of a function which will be called to process each key-value pair within the map. You return from the call to **yp_all()** only when the transaction is completed (successfully or unsuccessfully), or your foreach function decides that it does not want to see any more key-value pairs.

The third parameter to **yp_all()** is

```
struct ypall_callback *incallback {
int (*foreach)();
char *data;
};
```

The function **foreach** is called

```
foreach(instatus, inkey, inkeylen, inval, invallen, indata);
int instatus;
char *inkey;
int inkeylen;
char *inval;
int invallen;
char *indata;
```

The *instatus* parameter will hold one of the return status values defined in **<rpcsvc/yp_prot.h>** — either **YP_TRUE** or an error code. See **ypprot_err()**, below, for a function which converts a NIS protocol error code to a ypclnt layer error code.

The key and value parameters are somewhat different than defined in the synopsis section above. First, the memory pointed to by the *inkey* and *inval* parameters is private to the **yp_all()** function, and is overwritten with the arrival of each new key-value pair. It is the responsibility of the **foreach** function to do something useful with the contents of that memory, but it does not own the memory itself. Key and value objects presented to the **foreach** function look exactly as they do in the server's map — if they were not NEWLINE-terminated or null-terminated in the map, they will not be here either.

The *indata* parameter is the contents of the **incallback->data** element passed to **yp_all()**. The **data** element of the callback structure may be used to share state information between the **foreach** function and the mainline code. Its use is optional, and no part of the NIS client package inspects its contents — cast it to something useful, or ignore it as you see fit.

The **foreach** function is a Boolean. It should return zero to indicate that it wants to be called again for further received key-value pairs, or non-zero to stop the flow of key-value pairs. If **foreach** returns a non-zero value, it is not called again; the functional value of **yp_all()** is then 0.

yp_order(indomain, inmap, outorder);
char *indomain;
char *inmap;
int *outorder;

yp_order( ) returns the order number for a map.

yp_master(indomain, inmap, outname);
char *indomain;
char *inmap;
char **outname;

yp_master( ) returns the machine name of the master NIS server for a map.

char *yperr_string(incode)
int incode;

yperr_string( ) returns a pointer to an error message string that is null-terminated but contains no period or NEWLINE.

ypprot_err (incode)
unsigned int incode;

ypprot_err( ) takes a NIS protocol error code as input, and returns a ypclnt layer error code, which may be used in turn as an input to **yperr_string( )**.

FILES

<rpcsvc/ypclnt.h>
<rpcsvc/yp_prot.h>
/usr/lib/libc.a

SEE ALSO

malloc(3V), **ypupdate(3N)**, **ypfiles(5)**, **ypserv(8)**

DIAGNOSTICS

All integer functions return 0 if the requested operation is successful, or one of the following errors if the operation fails.

```
#define YPERR_BADARGS
        1       /* args to function are bad */

#define YPERR_RPC
        2       /* RPC failure - domain has been unbound */

#define YPERR_DOMAIN
        3       /* can't bind to server on this domain */

#define YPERR_MAP
        4       /* no such map in server's domain */

#define YPERR_KEY
        5       /* no such key in map */

#define YPERR_YPERR
        6       /* internal yp server or client error */

#define YPERR_RESRC
        7       /* resource allocation failure */

#define YPERR_NOMORE
        8       /* no more records in map database */

#define YPERR_PMAP
        9       /* can't communicate with portmapper */

#define YPERR_YPBIND
```

```
               10      /* can't communicate with ypbind */
        #define YPERR_YPSERV
               11      /* can't communicate with ypserv */
        #define YPERR_NODOM
               12      /* local domain name not set */
        #define YPERR_BADDBfR
               13      /* yp database is bad */
        #define YPERR_VERSfR
               14      /* yp version mismatch */
        #define YPERR_ACCESS
               15      /* access violation */
        #define YPERR_BUSY
               16      /* database busy */
```

NOTES

The Network Information Service (NIS) was formerly known as Sun Yellow Pages (YP). The functionality of the two remains the same; only the name has changed. The name Yellow Pages is a registered trademark in the United Kingdom of British Telecommunications plc, and may not be used without permission.

NAME

yp_update – changes NIS information

SYNOPSIS

```
#include <rpcsvc/ypclnt.h>

yp_update(domain, map, ypop, key, keylen, data, datalen)
char *domain;
char *map;
unsigned ypop
char *key;
int keylen;
char *data;
int datalen;
```

DESCRIPTION

yp_update() is used to make changes to the Network Information Service (NIS) database. The syntax is the same as that of **yp_match()** (see **ypclnt**(3N)) except for the extra parameter *ypop* which may take on one of four values. If it is **YPOP_CHANGE** then the data associated with the key will be changed to the new value. If the key is not found in the database, then **yp_update()** returns **YPERR_KEY**. If *ypop* has the value **YPOP_INSERT** then the key-value pair will be inserted into the database. The error **YPERR_KEY** is returned if the key already exists in the database. To store an item into the database without concern for whether it exists already or not, pass *ypop* as **YPOP_STORE** and no error will be returned if the key already or does not exist. To delete an entry, the value of *ypop* should be **YPOP_DELETE**.

This routine depends upon secure RPC, and will not work unless the network is running secure RPC.

SEE ALSO

ypclnt(3N)

System and Network Administration

NOTES

The Network Information Service (NIS) was formerly known as Sun Yellow Pages (YP). The functionality of the two remains the same; only the name has changed. The name Yellow Pages is a registered trademark in the United Kingdom of British Telecommunications plc, and may not be used without permission.

NAME

intro – introduction to the lightweight process library (LWP)

DESCRIPTION

The lightweight process library (LWP) provides a mechanism to support multiple threads of control that share a single address space. Under SunOS, the address space is derived from a single *forked* ("heavy-weight") process. Each thread has its own stack segment (specified when the thread is created) so that it can access local variables and make procedure calls independently of other threads. The collection of threads sharing an address space is called a *pod.* Under SunOS, threads share all of the resources of the heavyweight process that contains the pod, including descriptors and signal handlers.

The LWP provides a means for creating and destroying threads, message exchange between threads, manipulating condition variables and monitors, handling synchronous exceptions, mapping asynchronous events into messages, mapping synchronous events into exceptions, arranging for special per-thread context, multiplexing the clock for timeouts, and scheduling threads both preemptively and non–preemptively.

The LWP system exists as a library of routines (**/usr/lib/liblwp.a**) linked in (**–llwp**) with a client program which should **#include** the file **<lwp/lwp.h>**. **main** is transparently converted into a lightweight process as soon as it attempts to use any LWP primitives.

When an object created by a LWP primitive is destroyed, every attempt is made to clean up after it. For example, if a thread dies, all threads blocked on sends to or receives from that thread are unblocked, and all monitor locks held by the dead thread are released.

Because there is no kernel support for threads at present, system calls effectively block the entire pod. By linking in the non-blocking I/O library (**–lnbio**) ahead of the LWP library, you can alleviate this problem for those system calls that can issue a signal when a system call would be profitable to try. This library (which redefines some system calls) uses asynchronous I/O and events (for example, **SIGCHLD** and **SIGIO**) to make blocking less painful. The system calls remapped by the nbio library are: **open**(2V), **socket**(2), **pipe**(2V), **close**(2V), **read**(2V), **write**(2V), **send**(2), **recv**(2), **accept**(2), **connect**(2), **select** (2) and **wait**(2V).

RETURN VALUES

LWP primitives return non-negative integers on success. On errors, they return –1. See **lwp_perror**(3L) for details on error handling.

FILES

/usr/lib/liblwp.a
/usr/lib/libnbio.a

SEE ALSO

accept(2), **close**(2V), **connect**(2), **open**(2V), **pipe**(2V), **read**(2V), **recv**(2), **select**(2), **send**(2), **socket**(2), **wait**(2V) **write**(2V)
Lightweight Processes in the *System Services Overview*

INDEX

The following are the primitives currently supported, grouped roughly by function.

Thread Creation

lwp_self(tid)
lwp_getstate(tid, statvec)
lwp_setregs(tid, machstate)
lwp_getregs(tid, machstate)
lwp_ping(tid)
lwp_create(tid, pc, prio, flags, stack, nargs, arg1, . . . , argn)
lwp_destroy(tid)
lwp_enumerate(vec, maxsize)
pod_setexit(status)
pod_getexit( )
pod_exit(status)
SAMETHREAD(t1, t2)

- Thread Scheduling
  - **pod_setmaxpri(maxprio)**
  - **pod_getmaxpri( )**
  - **pod_getmaxsize( )**
  - **lwp_resched(prio)**
  - **lwp_setpri(tid, prio)**
  - **lwp_sleep(timeout)**
  - **lwp_suspend(tid)**
  - **lwp_resume(tid)**
  - **lwp_yield(tid)**
  - **lwp_join(tid)**
- Error Handling
  - **lwp_geterr( )**
  - **lwp_perror(s)**
  - **lwp_errstr( )**
- Messages
  - **msg_send(tid, argbuf, argsize, resbuf, ressize)**
  - **msg_recv(tid, argbuf, argsize, resbuf, ressize, timeout)**
  - **MSG_RECVALL(tid, argbuf, argsize, resbuf, ressize, timeout)**
  - **msg_reply(tid)**
  - **msg_enumsend(vec, maxsize)**
  - **msg_enumrecv(vec, maxsize)**
- Event Mapping (Agents)
  - **agt_create(agt, event, memory)**
  - **agt_enumerate(vec, maxsize)**
  - **agt_trap(event)**
- Thread Synchronization: Monitors
  - **mon_create(mid)**
  - **mon_destroy(mid)**
  - **mon_enter(mid)**
  - **mon_exit(mid)**
  - **mon_enumerate(vec, maxsize)**
  - **mon_waiters (mid, owner, vec, maxsize)**
  - **mon_cond_enter(mid)**
  - **mon_break(mid)**
  - **MONITOR(mid)**
  - **SAMEMON(m1, m2)**
- Thread Synchronization: Condition Variables
  - **cv_create(cv, mid)**
  - **cv_destroy(cv)**
  - **cv_wait(cv)**
  - **cv_notify(cv)**
  - **cv_send(cv, tid)**
  - **cv_broadcast(cv)**
  - **cv_enumerate(vec, maxsize)**
  - **cv_waiters(cv, vec, maxsize)**
  - **SAMECV(c1, c2)**
- Exception Handling
  - **exc_handle(pattern, func, arg)**
  - **exc_unhandle( )**
  - **(*exc_bound(pattern, arg))( )**
  - **exc_notify(pattern)**
  - **exc_raise(pattern)**

exc_on_exit(func, arg)
exc_uniqpatt( )

Special Context Handling

lwp_ctxinit(tid, cookie)
lwp_ctxremove(tid, cookie)
lwp_ctxset(save, restore, ctxsize, optimise)
lwp_ctxmemget(mem, tid, ctx)
lwp_ctxmemset(mem, tid, ctx)
lwp_fpset(tid)
lwp_libcset(tid)

Stack Management

CHECK(location, result)
lwp_setstkcache(minsize, numstks)
lwp_newstk( )
lwp_datastk(data, size, addr)
lwp_stkcswset(tid, limit)
lwp_checkstkset(tid, limit)
STKTOP(s)

BUGS

There is no language support available from C.
There is no kernel support yet. Thus system calls in different threads cannot execute in parallel.
Killing a process that uses the non-blocking I/O library may leave objects (such as its standard input) in a non-blocking state. This could cause confusion to the shell.

LIST OF LWP LIBRARY FUNCTIONS

Name	**Appears on Page**	**Description**
agt_create	**agt_create(3L)**	map LWP events into messages
agt_enumerate	**agt_create(3L)**	map LWP events into messages
agt_trap	**agt_create(3L)**	map LWP events into messages
CHECK	**lwp_newstk(3L)**	LWP stack management
cv_broadcast	**cv_create(3L)**	manage LWP condition variables
cv_create	**cv_create(3L)**	manage LWP condition variables
cv_destroy	**cv_create(3L)**	manage LWP condition variables
cv_enumerate	**cv_create(3L)**	manage LWP condition variables
cv_notify	**cv_create(3L)**	manage LWP condition variables
cv_send	**cv_create(3L)**	manage LWP condition variables
cv_wait	**cv_create(3L)**	manage LWP condition variables
cv_waiters	**cv_create(3L)**	manage LWP condition variables
exc_bound	**exc_handle(3L)**	LWP exception handling
exc_handle	**exc_handle(3L)**	LWP exception handling
exc_notify	**exc_handle(3L)**	LWP exception handling
exc_on_exit	**exc_handle(3L)**	LWP exception handling
exc_raise	**exc_handle(3L)**	LWP exception handling
exc_unhandle	**exc_handle(3L)**	LWP exception handling
exc_uniqpatt	**exc_handle(3L)**	LWP exception handling
lwp_checkstkset	**lwp_newstk(3L)**	LWP stack management
lwp_create	**lwp_create(3L)**	LWP thread creation and destruction primitives
lwp_ctxinit	**lwp_ctxinit(3L)**	special LWP context operations
lwp_ctxmemget	**lwp_ctxinit(3L)**	special LWP context operations
lwp_ctxmemset	**lwp_ctxinit(3L)**	special LWP context operations
lwp_ctxremove	**lwp_ctxinit(3L)**	special LWP context operations
lwp_ctxset	**lwp_ctxinit(3L)**	special LWP context operations
lwp_datastk	**lwp_newstk(3L)**	LWP stack management

lwp_destroy	**lwp_create**(3L)	LWP thread creation and destruction primitives
lwp_enumerate	**lwp_status**(3L)	LWP status information
lwp_errstr	**lwp_perror**(3L)	LWP error handling
lwp_fpset	**lwp_ctxinit**(3L)	special LWP context operations
lwp_geterr	**lwp_perror**(3L)	LWP error handling
lwp_getregs	**lwp_status**(3L)	LWP status information
lwp_getstate	**lwp_status**(3L)	LWP status information
lwp_join	**lwp_yield**(3L)	control LWP scheduling
lwp_libcset	**lwp_ctxinit**(3L)	special LWP context operations
lwp_newstk	**lwp_newstk**(3L)	LWP stack management
lwp_perror	**lwp_perror**(3L)	LWP error handling
lwp_ping	**lwp_status**(3L)	LWP status information
lwp_resched	**lwp_yield**(3L)	control LWP scheduling
lwp_resume	**lwp_yield**(3L)	control LWP scheduling
lwp_self	**lwp_status**(3L)	LWP status information
lwp_setpri	**lwp_yield**(3L)	control LWP scheduling
lwp_setregs	**lwp_status**(3L)	LWP status information
lwp_setstkcache	**lwp_newstk**(3L)	LWP stack management
lwp_sleep	**lwp_yield**(3L)	control LWP scheduling
lwp_stkcswset	**lwp_newstk**(3L)	LWP stack management
lwp_suspend	**lwp_yield**(3L)	control LWP scheduling
lwp_yield	**lwp_yield**(3L)	control LWP scheduling
MINSTACKSZ	**lwp_newstk**(3L)	LWP stack management
mon_break	**mon_create**(3L)	LWP routines to manage critical sections
mon_cond_enter	**mon_create**(3L)	LWP routines to manage critical sections
mon_create	**mon_create**(3L)	LWP routines to manage critical sections
mon_destroy	**mon_create**(3L)	LWP routines to manage critical sections
mon_enter	**mon_create**(3L)	LWP routines to manage critical sections
mon_enumerate	**mon_create**(3L)	LWP routines to manage critical sections
mon_exit	**mon_create**(3L)	LWP routines to manage critical sections
mon_waiters	**mon_create**(3L)	LWP routines to manage critical sections
MONITOR	**mon_create**(3L)	LWP routines to manage critical sections
msg_enumrecv	**msg_send**(3L)	LWP send and receive messages
msg_enumsend	**msg_send**(3L)	LWP send and receive messages
msg_recv	**msg_send**(3L)	LWP send and receive messages
MSG_RECVALL	**msg_send**(3L)	LWP send and receive messages
msg_reply	**msg_send**(3L)	LWP send and receive messages
msg_send	**msg_send**(3L)	LWP send and receive messages
pod_exit	**lwp_create**(3L)	LWP thread creation and destruction primitives
pod_getexit	**lwp_create**(3L)	LWP thread creation and destruction primitives
pod_getmaxpri	**pod_getmaxpri**(3L)	control LWP scheduling priority
pod_getmaxsize	**pod_getmaxpri**(3L)	control LWP scheduling priority
pod_setexit	**lwp_create**(3L)	LWP thread creation and destruction primitives
pod_setmaxpri	**pod_getmaxpri**(3L)	control LWP scheduling priority
SAMECV	**cv_create**(3L)	manage LWP condition variables
SAMEMON	**mon_create**(3L)	LWP routines to manage critical sections
SAMETHREAD	**lwp_create**(3L)	LWP thread creation and destruction primitives
STKTOP	**lwp_newstk**(3L)	LWP stack management

NAME

agt_create, agt_enumerate, agt_trap – map LWP events into messages

SYNOPSIS

```
#include <lwp/lwp.h>

thread_t agt_create(agt, event, memory)
thread_t *agt;
int event;
caddr_t memory;

int agt_enumerate(vec, maxsize)
thread_t vec[ ];
int maxsize;

int agt_trap(event)
int event;
```

DESCRIPTION

Agents are entities that act like threads sending messages when an asynchronous event occurs. **agt_create()** creates an object called an *agent* which maps the asynchronous event *event* into messages that can be received with **msg_recv()** (see **msg_send**(3L)). *agt* stores the handle on this object. *event* is a UNIX signal number.

agt_trap() causes the event, *event*, to generate an exception (see **exc_handle**(3L)). Once initialized using **agt_create()** or **agt_trap()**, an event can not be remapped to a different style of handling. If traps are enabled, an event will cause the termination of the *thread* running at the time of the trap if the trap exception is not handled. If an exception handler is in place, an exception will be raised. If an agent exists for the event, the event is mapped into a message for the agent. If neither agent nor trap mapping is enabled, the default signal action (SIG_DFL) is applied to the *pod*. Use of standard UNIX signal handling facilities will defeat the event mapping mechanism.

The message sent by the agent (in the argument buffer) will look like any other message with the sender being the agent. The receive buffer is NULL. A message is always sent by an agent to the thread which created the agent.

All messages sent by an agent contain an **eventinfo_t**. This structure indicates the thread running at the time the interrupt happened, and the particular event that occurred. Some agent messages contain more information if the particular event warrants it. In this case, a struct containing an **eventinfo_t** as its first element is passed as the argument buffer. Definitions of these structures are contained in **<lwp/lwp.h>**.

An agent appears to the owning thread just like another thread. It must therefore have some memory for holding its message, as the sender and receiver must belong to the same address space. *memory* is the space an agent will use to store its message. Typically, this is on the stack of the thread that created the agent. It must be of the correct size for the kind of event being created (most events need something to store an **eventinfo_t**. **SIGCHLD** events need room for a **sigchldev_t**.)

You should reply to an agent (using **msg_reply()** (see **msg_send**(3L)) as you would reply to a thread. Although agents do not ordinarily lose events, the next agent message will not be delivered until a reply is sent to the agent. Thus, an agent appears to the client as an ordinary thread sending messages. An agent will only lose events if the total number of unreplied-to events in a pod exceeds AGENTMEMORY.

lwp_destroy() is used to destroy an agent. All agents created by a thread automatically disappear when that thread dies. **agt_enumerate()** fills in a list with the ID's of all existing agents and returns the total number of agents. This primitive uses *maxsize* to avoid exceeding the capacity of the list. If the number of agents is greater than *maxsize*, only *maxsize* agents ID's are filled in *vec*. If *maxsize* is zero, **agt_enumerate()** returns the total number of agents.

The special event **LASTRITES** is caused by the termination of a thread. An agent for **LASTRITES** will be informed about every thread that terminates, regardless of cause. The **eventinfo_code** element of this agent will contain the stack argument that the dead thread was created with. Note: by allocating adjacent space above the thread stack, this argument can be used to point to private information about a thread. The **eventinfo_victimid** element will contain the id of the dead thread.

RETURN VALUES

agt_create( ) and **agt_trap( )** return:

0	on success.
−1	on failure.

agt_enumerate( ) returns the total number of agents.

ERRORS

agt_trap( ) will fail if one or more of the following are true:

LE_INUSE	Agent in use for this event.
LE_INVALIDARG	Event specified does not exist.

agt_create( ) will fail if one or more of the following are true:

LE_INUSE	Trap mapping in use for this event.
LE_INVALIDARG	Attempt to create agent for non-existent event.

SEE ALSO

exc_handle(3L), **msg_send**(3L)

BUGS

Signal handlers always take the **SIG_DFL** action when no agent manages the event.

If a descriptor used by a parent of the pod (such as its standard input) is marked non-blocking by a thread, it should be reset when the pod terminates to prevent the parent from receiving EWOULDBLOCK errors on the descriptor. There is no way to prevent this from happening if a pod is terminated with extreme prejudice (for instance, using **SIGKILL**).

If an agent reports that a descriptor has I/O available, there may be more than one occurrence of I/O available from that descriptor. Thus, being informed that **SIGIO** has occurred on socket *s* may mean that there are several messages waiting to be received from *s*. Clients should be careful to clean out all I/O from a descriptor before going back to sleep.

All system calls should be protected with loops testing for EINTR (and monitors if multiple threads can try to use system calls concurrently). An **lwp_sleep( )** could result in a hidden clock interrupt for example.

WARNINGS

agt_trap( ) should not be used for asynchronous events. If an unsuspecting thread which has no exception handler is running at the time of a trapped event, it will be terminated.

Clients should not normally handle signals themselves since the agent mechanism assumes it is the only entity handling signals.

NAME

cv_create, cv_destroy, cv_wait, cv_notify, cv_broadcast, cv_send, cv_enumerate, cv_waiters, SAMECV – manage LWP condition variables

SYNOPSIS

```
#include <lwp/lwp.h>

cv_t cv_create(cv, mid)
cv_t *cv;
mon_t mid;

int cv_destroy(cv)
cv_t cv;

int cv_wait(cv)
cv_t cv;

int cv_notify(cv)
cv_t cv;

int cv_send(cv, tid)
cv_t cv;
lwp_t tid

int cv_broadcast(cv)
cv_t cv;

int cv_enumerate(vec, maxsize)
cv_t vec[ ];        /* will contain list of all conditions */
int maxsize;        /* maximum size of vec */

int cv_waiters(cv, vec, maxsize)
cv_t cv;            /* condition variable being interrogated */
thread_t vec[ ];    /* which threads are blocked on cv */
int maxsize;        /* maximum size of vec */

SAMECV(c1, c2)
```

DESCRIPTION

Condition variables are useful for synchronization within monitors. By waiting on a condition variable, the currently-held monitor (a condition variable must *always* be used within a monitor) is released atomically and the invoking thread is suspended. When monitors are nested, monitor locks other than the current one are retained by the thread. At some later point, a different thread may awaken the waiting thread by issuing a notification on the condition variable. When the notification occurs, the waiting thread will queue to reacquire the monitor it gave up. It is possible to have different condition variables operating within the same monitor to allow selectivity in waking up threads.

cv_create( ) creates a new condition variable (returned in *cv*) which is bound to the monitor specified by *mid*. It is illegal to access (using **cv_wait( )**, **cv_notify( )**, **cv_send( )** or **cv_broadcast( )**) a condition variable from a monitor other than the one it is bound to. **cv_destroy( )** removes a condition variable.

cv_wait( ) blocks the current thread and releases the monitor lock associated with the condition (which must also be the monitor lock most recently acquired by the thread). Other monitor locks held by the thread are not affected. The blocked thread is enqueued by its scheduling priority on the condition.

cv_notify( ) awakens at most one thread blocked on the condition variable and causes the awakened thread to queue for access to the monitor released at the time it waited on the condition. It can be dangerous to use **cv_notify( )** if there is a possibility that the thread being awakened is one of several threads that are waiting on a condition variable and the awakened thread may not be the one intended. In this case, use of **cv_broadcast( )** is recommended.

cv_broadcast( ) is the same as **cv_notify( )** except that *all* threads blocked on the condition variable are awakened. **cv_notify( )** and **cv_broadcast( )** do nothing if no thread is waiting on the condition. For both **cv_notify( )** and **cv_broadcast( )**, the currently held monitor must agree with the one bound to the condition by **cv_create( )**.

cv_send( ) is like **cv_notify( )** except that the particular thread **tid** is awakened. If this thread is not currently blocked on the condition, **cv_send( )** reports an error.

cv_enumerate( ) lists the ID of all of the condition variables. The value returned is the total number of condition variables. The vector supplied is filled in with the ID's of condition variables. **cv_waiters( )** lists the ID's of the threads blocked on the condition variable *cv* and returns the number of threads blocked on *cv*. For both **cv_enumerate( )** and **cv_waiters( )**, *maxsize* is used to avoid exceeding the capacity of the list *vec*. If the number of entries to be filled is greater than *maxsize*, only *maxsize* entries are filled in *vec*. It is legal in both of these primitives to specify a *maxsize* of 0.

SAMECV is a convenient predicate used to compare two condition variables for equality.

RETURN VALUES

cv_create( ), **cv_destroy( )**, **cv_send( )**, **cv_wait( )**, **cv_notify( )** and **cv_broadcast( )** return:

0	on success.
−1	on failure and set **errno** to indicate the error.

cv_enumerate( ) returns the total number of condition variables.

cv_waiters( ) returns the number of threads blocked on a condition variable.

ERRORS

cv_destroy( ) will fail if one or more of the following is true:

LE_INUSE	Attempt to destroy condition variable being waited on by a thread.
LE_NONEXIST	Attempt to destroy non-existent condition variable.

cv_wait( ) will fail if one or more of the following is true:

LE_NONEXIST	Attempt to wait on non-existent condition variable.
LE_NOTOWNED	Attempt to wait on a condition without possessing the correct monitor lock.

cv_notify( ) will fail if one or more of the following is true:

LE_NONEXIST	Attempt to notify non-existent condition variable.
LE_NOTOWNED	Attempt to notify condition variable without possessing the correct monitor.

cv_send( ) will fail if one or more of the following is true:

LE_NONEXIST	Attempt to awaken non-existent condition variable.
LE_NOTOWNED	Attempt to awaken condition variable without possessing the correct monitor lock.
LE_NOWAIT	The specified thread is not currently blocked on the condition.

cv_broadcast( ) will fail if one or more of the following is true:

LE_NONEXIST	Attempt to broadcast non-existent condition variable.
LE_NOTOWNED	Attempt to broadcast condition without possessing the correct monitor lock.

SEE ALSO

mon_create(3L)

NAME

exc_handle, exc_unhandle, exc_bound, exc_notify, exc_raise, exc_on_exit, exc_uniqpatt – LWP exception handling

SYNOPSIS

```
#include <lwp/lwp.h>

int exc_handle(pattern, func, arg)
int pattern;
caddr_t (*func)();
caddr_t arg;

int exc_raise(pattern)
int pattern;

int exc_unhandle()

caddr_t (*exc_bound(pattern, arg))()
int pattern;
caddr_t *arg;

int exc_notify(pattern)
int pattern;

int exc_on_exit(func, arg)
void (*func)();
caddr_t arg;

int exc_uniqpatt()
```

DESCRIPTION

These primitives can be used to manage exceptional conditions in a thread. Basically, raising an exception is a more general form of non-local goto or *longjmp*, but the invocation is pattern-based. It is also possible to *notify* an exception handler whereby a function supplied by the exception handler is invoked and control is returned to the raiser of the exception. Finally, one can establish a handler which is always invoked upon procedure exit, regardless of whether the procedure exits using a *return* or an exception raised to a handler established prior to the invocation of the exiting procedure.

exc_handle() is used to establish an exception handler. **exc_handle()** returns 0 to indicate that a handler has been established. A return of −1 indicates an error in trying to establish the exception handler. If it returns something else, an exception has occurred and any procedure calls deeper than the one containing the handler have disappeared. All exception handlers established by a procedure are automatically discarded when the procedure terminates.

exc_handle() binds a *pattern* to the handler, where a pattern is an integer, and two patterns *match* if their values are equal. When an exception is raised with **exc_raise()**, the most recent handler that has established a matching pattern will catch the exception. A special pattern (CATCHALL) is provided which matches any **exc_raise()** pattern. This is useful for handlers which know that there is no chance the resources allocated in a routine can be reclaimed by previous routines in the call chain.

The other two arguments to **exc_handle()** are a function and an argument to that function. **exc_bound()** retrieves these arguments from an **exc_hanc'le()** call made by the specified thread. By using **exc_bound()** to retrieve and call a function bound by the exception handler, a procedure can raise a *notification exception* which allows control to return to the raiser of the exception after the exception is handled.

exc_raise( ) allows the caller to transfer control (do a non-local goto) to the matching **exc_handle**( ). This matching exception handler is destroyed after the control transfer. At this time, it behaves as if **exc_handle**( ) returns with the *pattern* from **exc_raise**( ) as the return value. Note: *func* of **exc_handle**( ) is not called using **exc_raise**( ) — it is only there for notification exceptions. Because the exception handler returns the pattern that invoked it, it is possible for a handler that matches the CATCHALL pattern to *reraise* the exact exception it caught by using **exc_raise**( ) on the caught pattern. It is illegal to handle or raise the pattern 0 or the pattern −1. Handlers are searched for pattern matches in the reverse execution order that they are set (i.e., the most recently established handler is searched first).

exc_unhandle( ) destroys the most recently established exception handler set by the current thread. It is an error to destroy an exit-handler set up by **exc_on_exit**( ). When a procedure exits, all handlers and exit handlers set in the procedure are automatically deallocated.

exc_notify( ) is a convenient way to use **exc_bound**. The function which is bound to *pattern* is retrieved. If the function is not NULL, the function is called with the associated argument and the result is returned. If the function is NULL, **exc_raise(***pattern***)** is returned.

exc_on_exit( ) specifies an exit procedure and argument to be passed to the exit procedure, which is called when the procedure which sets an exit handler using **exc_on_exit**( ) exits. The exit procedures (more than one may be set) will be called regardless if the setting procedure is exited using a *return* or an **exc_raise**( ). Because the exit procedure is called as if the handling procedure had returned, the argument passed to it should not contain addresses on the handler's stack. However, any value returned by the procedure which established the exit procedure is preserved no matter what the exit procedure returns. This primitive is used in the MONITOR macro to enforce the monitor discipline on procedures.

Some signals can be considered to be synchronous traps. They are usually the starred (*) signals in the **signal**(3V) man pages. These are: **SIGSYS, SIGBUS, SIGEMT, SIGFPE, SIGILL, SIGTRAP, SIGSEGV.** If an event is marked as a trap using **agt_trap**() (see **agt_create**(3L)) the event will generate exceptions instead of agent messages. This mapping is per-pod, not per-thread. A thread which handles the signal number of one of these as the pattern for **exc_handle**( ) will catch such a signal as an exception. The exception will be raised as an **exc_notify**( ) so either escape or notification style exceptions can be used, depending on what the matching **exc_handle**( ) provides. If the exception is not handled, the thread will terminate. Note: it can be dangerous to supply an exception handler to treat stack overflow since the client's stack is used in raising the exception.

exc_uniqpatt( ) returns an exception pattern that is not any of the pre-defined patterns (any of the synchronous exceptions or −1 or CATCHALL). Each call to **exc_uniqpatt**() results in a different pattern. If **exc_uniqpatt**( ) cannot guarantee uniqueness, −1 is returned instead the *first* time this happens. Subsequent calls after this error result in patterns which may be duplicates.

RETURN VALUES

exc_uniqpatt( ) returns a unique pattern on success. The *first* time it fails, **exc_uniqpatt**( ) returns −1.

exc_handle( ) returns:

0 on success.

−1 on failure. When **exc_handle**( ) returns because of a matching call to **exc_raise**( ), it returns the *pattern* raised by **exc_raise**( ).

On success, **exc_raise**( ) transfers control to the matching **exc_handle**() and does not return. On failure, it returns −1.

exc_unhandle( ) returns:

0 on success.

−1 on failure.

exc_bound( ) returns a pointer to a function on success. On failure, it returns NULL.

On success, **exc_notify**( ) returns the return value of a function, or transfers control to a matching **exc_handle**( ) and does not return. On failure, it returns −1.

exc_on_exit( ) returns 0.

ERRORS

exc_unhandle( ) will fail if one or more of the following is true:

LE_NONEXIST	Attempt to remove a non-existent handler.
	Attempt to remove an exit handler.

exc_raise( ) will fail if one or more of the following is true:

LE_INVALIDARG	Attempt to raise an illegal pattern (−1 or 0).
LE_NONEXIST	No context found to raise an exception to.

exc_handle( ) will fail if one or more of the following is true:

LE_INVALIDARG	Attempt to handle an illegal pattern (−1 or 0).

exc_uniqpatt( ) will fail if one or more of the following is true:

LE_REUSE	Possible reuse of existing object. **agt_create**(3L), **signal**(3V)

BUGS

The stack may not contain useful information after an exception has been caught so post-exception debugging can be difficult. The reason for this is that a given handler may call procedures that trash the stack before reraising an exception.

The distinction between traps and interrupts can be problematical.

The environment restored on **exc_raise**( ) consists of the registers at the time of the **exc_handle**( ). As a result, modifications to register variables between the times of **exc_handle**() and **exc_raise**( ) will not be seen. This problem does not occur in the sun4 implementation.

WARNINGS

exc_on_exit( ) passes a simple type as an argument to the exit routine. If you need to pass a complex type, such as **thread_t**, **mon_t**, or **cv_t**, pass a pointer to the object instead.

NAME

lwp_create, lwp_destroy, SAMETHREAD, pod_setexit, pod_getexit, pod_exit – LWP thread creation and destruction primitives

SYNOPSIS

```
#include <lwp/lwp.h>
#include <lwp/stackdep.h>

int lwp_create(tid, func, prio, flags, stack, nargs, arg1, . . ., argn)
thread_t *tid;
void (*func)( );
int prio;
int flags;
stkalign_t *stack;
int nargs;
int arg1, . . . , argn;

int lwp_destroy(tid)
thread_t tid;

void pod_setexit(status)
int status;

int pod_getexit(status)
int status;

void pod_exit(status)
int status

SAMETHREAD(t1, t2)
```

DESCRIPTION

lwp_create( ) creates a lightweight process which starts at address *func* and has stack segment *stack*. If *stack* is NULL, the thread is created in a suspended state (see below) and no stack or pc is bound to the thread. *prio* is the scheduling priority of the thread (higher priorities are favored by the scheduler). The identity of the new thread is filled in the reference parameter *tid*. *flags* describes some options on the new thread. **LWPSUSPEND** creates the thread in suspended state (see **lwp_yield**(3L)). **LWPNOLASTRITES** will disable the **LASTRITES** agent message when the thread dies. The default (0) is to create the thread in running state with **LASTRITES** reporting enabled. **LWPSERVER** indicates that a thread is only viable as long as non-LWPSERVER threads are alive. The pod will terminate if the only living threads are marked **LWPSERVER** and blocked on a lwp resource (for instance, waiting for a message to be sent). *nargs* is the number (0 or more) of simple-type (int) arguments supplied to the thread.

The first time a lwp primitive is used, the lwp library automatically converts the caller (i.e., **main**) into a thread with the highest available scheduling priority (see **pod_getmaxpri**(3L)). The identity of this thread can be retrieved using **lwp_self** (see **lwp_status**(3L)). This thread has the normal SunOS stack given to any *forked* process.

Scheduling is, by default, non-preemptive within a priority, and within a priority, threads enter the run queue on a FIFO basis (that is, whenever a thread becomes eligible to run, it goes to the end of the run queue of its particular priority). Thus, a thread continues to run until it voluntarily relinquishes control or an event (including thread creation) occurs to enable a higher priority thread. Some primitives may cause the current thread to block, in which case the unblocked thread with the highest priority runs next. When several threads are created with the same priority, they are queued for execution in the order of creation. This order may not be preserved as threads yield and block within a priority. If an agent owned by a thread with a higher priority is invoked, that thread will preempt the currently running one.

There is no concept of ancestry in threads: the creator of a thread has no special relation to the thread it created. When all threads have died, the pod terminates.

lwp_destroy( ) is a way to explicitly terminate a thread or agent (instead of having an executing thread "fall though", which also terminates the thread). *tid* specifies the id of the thread or agent to be terminated. If *tid* is **SELF**, the invoking thread is destroyed. Upon termination, the resources (messages, monitor locks, agents) owned by the thread are released, in some cases resulting in another thread being notified of the death of its peer (by having a blocking primitive become unblocked with an error indication). A thread may terminate itself explicitly, although self-destruction is automatic when it returns from the procedure specified in the **lwp_create**( ) primitive.

pod_setexit( ) sets the exit status for a pod. This value will be returned to the parent process of the pod when the pod dies (default is 0). **exit**(3) terminates the current *thread*, using the argument supplied to *exit* to set the current value of the exit status. **on_exit**(3) establishes an action that will be taken when the entire pod terminates. **pod_exit**( ) is available to terminate the pod immediately with the final actions established by **on_exit**. If you wish to terminate the pod immediately, **pod_exit**( ) or **exit**(2V) should be used.

pod_getexit( ) returns the current value of the pod's exit status.

SAMETHREAD( ) is a convenient predicate used to compare two threads for equality.

RETURN VALUES

lwp_create( ), and **lwp_destroy**( ) return:

0 on success.

−1 on failure.

pod_getexit( ) returns the current exit status of the pod.

ERRORS

lwp_create( ) will fail if one or more of the following are true:

LE_ILLPRIO Illegal priority.

LE_INVALIDARG Too many arguments (> 512).

LE_NOROOM Unable to allocate memory for thread context.

lwp_destroy( ) will fail if one or more of the following are true:

LE_NONEXIST Attempt to destroy a thread or agent that does not exist.

SEE ALSO

exit(2V), **exit**(3), **lwp_yield**(3L), **on_exit**(3), **pod_getmaxpri**(3L)

WARNINGS

Some special threads may be created silently by the lwp library. These include an *idle* thread that runs when no other activity is going on, and a *reaper* thread that frees stacks allocated by **lwp_newstk**. These special threads will show up in status calls. A pod will terminate if these special threads are the only ones extant.

NAME

lwp_ctxinit, lwp_ctxremove, lwp_ctxset, lwp_ctxmemget, lwp_ctxmemset, lwp_fpset, lwp_libcset – special LWP context operations

SYNOPSIS

```
#include <lwp/lwp.h>

int lwp_ctxset(save, restore, ctxsize, optimize)
void (*save)(/* caddr_t ctx, thread_t old, thread_t new */);
void (*restore)(/* caddr_t ctx, thread_t old, thread_t new */);
unsigned int ctxsize;
int optimize;

int lwp_ctxinit(tid, cookie)
thread_t tid;           /* thread with special contexts */
int cookie;             /* type of context */

int lwp_ctxremove(tid, cookie)
thread_t tid;
int cookie;

int lwp_ctxmemget(mem, tid, ctx)
caddr_t mem;
thread_t tid;
int ctx;

int lwp_ctxmemset(mem, tid, ctx)
caddr_t mem;
thread_t tid;
int ctx;

int lwp_fpset(tid)
thread_t tid;           /* thread utilizing floating point hardware */

int lwp_libcset(tid)
thread_t tid;           /* thread utilizing errno */
```

DESCRIPTION

Normally on a context switch, only machine registers are saved/restored to provide each thread its own virtual machine. However, there are other hardware and software resources which can be multiplexed in this way. For example, floating point registers can be used by several threads in a pod. As another example, the global value **errno** in the standard C library may be used by all threads making system calls.

To accommodate the variety of contexts that a thread may need without requiring all threads to pay for unneeded switching overhead, **lwp_ctxinit**( ) is provided. This primitive allows a client to specify that a given thread requires certain context to be saved and restored across context switches (by default just the machine registers are switched). More than one special context may be given to a thread.

To use **lwp_ctxinit**(), it is first necessary to define a special context. **lwp_ctxset**( ) specifies save and restore routines, as well as the size of the context that will be used to hold the switchable state. The *save* routine will automatically be invoked when an active thread is blocked and the *restore* routine will be invoked when a blocked thread is restarted. These routines will be passed a pointer to a buffer (initialized to all 0's) of size *ctxsize* which is allocated by the LWP library and used to hold the volatile state. In addition, the identity of the thread whose special context is being saved (old) and the identity of the thread being restarted (new) are passed in to the *save* and *restore* routines. **lwp_ctxset**( ) returns a cookie used by subsequent **lwp_ctxinit**( ) calls to refer to the kind of context just defined. If the *optimize* flag is TRUE, a special context switch action will not be invoked unless the thread resuming execution differs from the last thread to use the special context and also uses the special context. If the *optimize* flag is FALSE, the *save* routine will always be invoked immediately when the thread using this context is scheduled out and the *restore* routine will be invoked immediately when a new thread using this context is scheduled in. Note that an unoptimized special context is protected from threads which do not use the special context but which do affect the context state. **lwp_ctxremove( )** can be used to remove a special context installed by **lwp_ctxinit( )**.

Because context switching is done by the scheduler on behalf of a thread, it is an error to use an LWP primitive in an action done at context switch time. Also, the stack used by the save and restore routines belongs to the scheduler, so care should be taken not to use lots of stack space. As a result of these restrictions, only knowledgeable users should write their own special context switching routines.

lwp_ctxmemget( ) and **lwp_ctxmemset( )** are used to retrieve and set (respectively) the memory associated with a given special context (*ctx*) and a given thread (*tid*). *mem* is the address of client memory that will hold the context information being retrieved or set. Note that the special context *save* and *restore* routines may be NULL, so pure data may be associated with a given thread using these primitives.

Several kinds of special contexts are predefined. To allow a thread to share floating point hardware with other threads, the **lwp_fpset( )** primitive is available. The floating-point hardware bound at compile-time is selected automatically. To multiplex the global variable **errno**, **lwp_libcset( )** is used to have **errno** become part of the context of thread *tid*.

Special contexts can be used to assist in managing stacks. See **lwp_newstk**(3L) for details.

RETURN VALUES

On success, **lwp_ctxset( )** returns a cookie to be used by subsequent calls to **lwp_ctxinit( )**. If unable to define the context, it returns −1.

ERRORS

lwp_ctxinit( ) will fail if one or more of the following are true:

LE_INUSE   This special context already set for this thread.

lwp_ctxremove( ) will fail if one or more of the following are true:

LE_NONEXIST   The specified context is not set for this thread.

lwp_ctxset( ) will fail if one or more of the following are true:

LE_NOROOM   Unable to allocate memory to define special context.

SEE ALSO

lwp_newstk(3L)

BUGS

The floating point contexts should be initialized implicitly for those threads that use floating point.

NAME

lwp_checkstkset, lwp_stkcswset, CHECK, lwp_setstkcache, lwp_newstk, lwp_datastk, STKTOP – LWP stack management

SYNOPSIS

```
#include <lwp/lwp.h>
#include <lwp/check.h>
#include <lwp/lwpmachdep.h>
#include <lwp/stackdep.h>

CHECK(location, result)

int lwp_checkstkset(tid, limit)
thread_t tid;
caddr_t limit;

int lwp_stkcswset(tid, limit)
thread_t tid;
caddr_t limit;

int lwp_setstkcache(minstksz, numstks)
int minstksz;
int numstks;

stkalign_t *lwp_newstk()

stkalign_t *lwp_datastk(data, size, addr)
caddr_t data;
int size;
caddr_t *addr;

STKTOP(s)
```

DESCRIPTION

Stacks are problematical with lightweight processes. What is desired is that stacks for each thread are redzone protected so that one thread's stack does not unexpectedly grow into the stack of another. In addition, stacks should be of infinite length, grown as needed. The process stack is a maximum-sized segment (see **getrlimit**(2).) This stack is redzone protected, and you can even try to extend it beyond its initial maximum size in some cases. With SunOS 4.*x*, it is possible to efficiently allocate large stacks that have red zone protection, and the LWP library provides some support for this. For those systems that do not have flexible memory management, the LWP library provides assistance in dealing with the problems of maintaining multiple stacks.

The stack used by **main**() is the same stack that the system allocates for a process on **fork**(2V). For allocating other thread stacks, the client is free to use any statically or dynamically allocated memory (using memory from **main**()'s stack is subject to the stack resource limit for any process created by **fork**()). In addition, the **LASTRITES** agent message is available to free allocated resources when a thread dies. The size of any stack should be at least **MINSTACKSZ * sizeof (stkalign_t)**, because the LWP library will use the client stack to execute primitives. For very fast dynamically allocated stacks, a stack cacheing mechanism is available. **lwp_setstkcache**() allocates a cache of stacks. Each time the cache is empty, it is filled with *numstks* new stacks, each containing at least *minstksz* bytes. *minstksz* will automatically be augmented to take into account the stack needs of the LWP library. **lwp_newstk**() returns a cached stack that is suitable for use in an **lwp_create**() call. **lwp_setstkcache**() must be called (once) prior to any use of **lwp_newstk**. If running under SunOS 4.*x*, the stacks allocated by **lwp_newstk**() will be red-zone protected (an attempt to reference below the stack bottom will result in a **SIGSEGV** event).

Threads created with stacks from **lwp_newstk**() should not use the **NOLASTRITES** flag. If they do, cached stacks will not be returned to the cache when a thread dies.

lwp_datastk( ) also returns a red-zone protected stack like **lwp_newstk**( ) does. It copies any amount of data (subject to the size limitations imposed by **lwp_setstkcache**) onto the stack *above* the stack top that it returns. *data* points to information of *size* bytes to be copied. The exact location where the data is stored is returned in the reference parameter *addr*. Because **lwp_create**( ) only passes simple types to the newly-created thread, **lwp_datastk**( ) is useful to pass a more complex argument: Call **lwp_datastk**( ) to get an initialized stack, and pass the address of the data structure (*addr*) as an argument to the new thread.

A *reaper* thread running at the maximum pod priority is created by **lwp_setstkcache**. It's action may be delayed by other threads running at that priority, so it is suggested that the maximum pod priority not be used for client-created threads when **lwp_newstk**( ) is being used. Altering the maximum pod priority with **pod_setmaxpri**( ) will have the side effect of increasing the reaper thread priority as well.

The stack address passed to **lwp_create**( ) represents the top of the stack: the LWP library will not use any addresses at or above it. Thus, it is safe to store information above the stack top if there is room there.

For stacks that are not protected with hardware redzones, some protection is still possible. For any thread *tid* with stack boundary *limit* made part of a special context with **lwp_checkstkset**( ), the CHECK macro may be used. This macro, if used at the beginning of each procedure (and before local storage is initialized (it is all right to *declare* locals though)), will check that the stack limit has not been violated. If it has, the non-local *location* will be set to *result* and the procedure will return. CHECK is not perfect, as it is possible to call a procedure with many arguments after CHECK validates the stack, only to have these arguments clobber the stack before the new procedure is entered.

lwp_stkcswset( ) checks at context-switch time the stack belonging to thread *tid* for passing stack boundary *limit*. In addition, a checksum at the bottom of the stack is validated to ensure that the stack did not temporarily grow beyond its limit. This is automated and more efficient than using CHECK, but by the time a context switch occurs, it's too late to do much but **abort**(3) if the stack was clobbered.

To portably use statically allocated stacks, the macros in **<lwp/stackdep.h>** should be used. Declare a stack *s* to be an array of **stkalign_t**, and pass the stack to **lwp_create**( ) as STKTOP(s).

RETURN VALUES

lwp_checkstkset( ) and **lwp_stkcswset**( ) return 0.

lwp_setstkcache( ) returns the actual size of the stacks allocated in the cache.

lwp_newstk( ) and **lwp_datastk**( ) return a valid new stack address on success. On failure, they return 0.

SEE ALSO

getrlimit(2), **abort**(3)

WARNINGS

lwp_datastk( ) should not be directly used in a **lwp_create**( ) call since C does not guarantee the order in which arguments to a function are evaluated.

BUGS

C should provide support for heap-allocated stacks at procedure entry time. The hardware should be segment-based to eliminate the problem altogether.

NAME

lwp_geterr, lwp_perror, lwp_errstr – LWP error handling

SYNOPSIS

```
#include <lwp/lwp.h>
#include <lwp/lwperror.h>

lwp_err_t lwp_geterr( );

void
lwp_perror(s)
char *s;

char **lwp_errstr( );
```

DESCRIPTION

When a primitive fails (returns –1), **lwp_geterr( )** can be used to obtain the identity of the error (which is part of the context for each lwp). **lwp_perror( )** can be used to print an error message on the standard error file (analogous to **perror**(3)) when a lwp primitive returns an error indication. **lwp_perror( )** uses the same mechanism as **lwp_geterr( )** to obtain the last error. **lwp_errstr** returns a pointer to the (NULL-terminated) list of error messages.

lwp_libcset (see **lwp_ctxinit**(3L)) allows **errno** from the standard C library reflect a per-thread value rather than a per-pod value.

SEE ALSO

lwp_ctxinit(3L), **perror**(3)

NAME

lwp_self, lwp_ping, lwp_enumerate, lwp_getstate, lwp_setregs, lwp_getregs – LWP status information

SYNOPSIS

```
#include <lwp/lwp.h>
#include <lwp/lwpmachdep.h>

int
lwp_enumerate(vec, maxsize)
thread_t vec[ ];  /* list of id's to be filled in */
int maxsize;      /* number of elements in vec */

int
lwp_ping(tid)
thread_t tid;

int
lwp_getregs(tid, machstate)
thread_t tid;
machstate_t *machstate;

int
lwp_setregs(tid, machstate)
thread_t tid;
machstate_t *machstate;

int
lwp_getstate(tid, statvec)
thread_t tid;
statvec_t *statvec;

int
lwp_self(tid)
thread_t *tid;
```

DESCRIPTION

lwp_self( ) returns the ID of the current thread in *tid*. This is the *only* way to retrieve the identity of *main*.

lwp_enumerate( ) fills in a list with the ID's of all existing threads and returns the total number of threads. This primitive will use *maxsize* to avoid exceeding the capacity of the list. If the number of threads is greater than *maxsize*, only *maxsize* thread ID's are filled in *vec*. If *maxsize* is zero, **lwp_enumerate( )** just returns the total number of threads.

lwp_getstate( ) is used to retrieve the context of a given thread. It is possible to see what object (thread, monitor, etc.) if any that thread is blocked on, and the scheduling priority of the thread.

lwp_ping returns 0 (no error) if the thread *tid* exists. Otherwise, -1 is returned.

lwp_setregs sets the machine-dependent context (i.e., registers) of a thread. The next time the thread is scheduled in, this context is installed. Consult **lwpmachdep.h** for the details. **lwp_getregs** retrieves the machine-dependent context. Note: the registers may not be meaningful unless the thread in question is blocked or suspended because the state of the registers as of the most recent context switch is returned.

RETURNS

Upon successful completion, **lwp_self** and **lwp_getstate( )** return 0, –1 on error.

lwp_enumerate( ) returns the total number of threads.

lwp_ping returns 0 if the specified thread exists, else -1.

ERRORS

lwp_getstatea( ) , **lwp_ping( )** , and **lwp_setstate( )** will fail if one or more of the following is true:

LE_NONEXIST	Attempt to get the status of a non-existent thread.

NAME

lwp_yield, lwp_suspend, lwp_resume, lwp_join, lwp_setpri, lwp_resched, lwp_sleep – control LWP scheduling

SYNOPSIS

```
#include <lwp/lwp.h>

int lwp_yield(tid)
thread_t tid;

int lwp_sleep(timeout)
struct timeval *timeout;

int lwp_resched(prio)
int prio;

int lwp_setpri(tid, prio)
thread_t tid;
int prio;

int lwp_suspend(tid)
thread_t tid;

int lwp_resume(tid)
thread_t tid;

int lwp_join(tid)
thread_t tid;
```

DESCRIPTION

lwp_yield( ) allows the currently running thread to voluntarily relinquish control to another thread *with the same scheduling priority*. If *tid* is SELF, the next thread in the same priority queue of the yielding thread will run and the current thread will go the end of the scheduling queue. Otherwise, it is the ID of the thread to run next, and the current thread will take second place in the scheduling queue.

lwp_sleep( ) blocks the thread executing this primitive for at least the time specified by *timeout*.

Scheduling of threads is, by default, preemptive (higher priorities preempt lower ones) across priorities and non-preemptive within a priority. **lwp_resched( )** moves the front thread for a given priority to the end of the scheduling queue. Thus, to achieve a preemptive round–robin scheduling discipline, a high priority thread can periodically wake up and shuffle the queue of threads at a lower priority. **lwp_resched( )** does not affect threads which are blocked. If the priority of the rescheduled thread is the same as that of the caller, the effect is the same as **lwp_yield( )**.

lwp_setpri( ) is used to alter (raise or lower) the scheduling priority of the specified thread. If *tid* is SELF, the priority of the invoking thread is set. Note: if the priority of the affected thread becomes greater than that of the caller and the affected thread is not blocked, the caller will not run next. **lwp_setpri( )** can be used on either blocked or unblocked threads.

lwp_join( ) blocks the thread issuing the join until the thread *tid* terminates. More than one thread may join *tid*.

lwp_suspend( ) makes the specified thread ineligible to run. If *tid* is SELF, the caller is itself suspended. **lwp_resume( )** undoes the effect of **lwp_suspend( )**. If a blocked thread is suspended, it will not run until it has been unblocked as well as explicitly made eligible to run using **lwp_resume( )**. By suspending a thread, one can safely examine it without worrying that its execution–time state will change.

NOTES

When scheduling preemptively, be sure to use monitors to protect shared data structures such as those used by the standard I/O library.

RETURN VALUES

lwp_yield( ), **lwp_sleep( )**, **lwp_resched( )**, **lwp_join( )**, **lwp_suspend( )** and **lwp_resume( )** return:

0 on success.

−1 on failure.

lwp_setpri( ) returns the previous priority on success. On failure, it returns −1.

ERRORS

lwp_yield( ) will fail if one or more of the following is true:

LE_ILLPRIO Attempt to yield to thread with different priority.

LE_INVALIDARG Attempt to yield to a blocked thread.

LE_NONEXIST Attempt to yield to a non-existent thread.

lwp_sleep( ) will fail if one or more of the following is true:

LE_INVALIDARG Illegal timeout specified.

lwp_resched( ) will fail if one or more of the following is true:

LE_ILLPRIO The priority queue specified contains no threads to reschedule.

LE_INVALIDARG Attempt to reschedule thread at priority greater than that of the caller.

lwp_setpri( ) will fail if one or more of the following is true:

LE_INVALIDARG The priority specified is beyond the maximum available to the pod.

LE_NONEXIST Attempt to set priority of a non-existent thread.

lwp_join( ) will fail if one or more of the following are true:

LE_NONEXIST Attempt to join a thread that does not exist.

lwp_suspend( ) will fail if one or more of the following is true:

LE_NONEXIST Attempt to suspend a non-existent thread.

lwp_resume( ) will fail if one or more of the following is true:

LE_NONEXIST Attempt to resume a non-existent thread.

NAME

mon_create, mon_destroy, mon_enter, mon_exit, mon_enumerate, mon_waiters, mon_cond_enter, mon_break, MONITOR, SAMEMON – LWP routines to manage critical sections

SYNOPSIS

```
#include <lwp/lwp.h>

int mon_create(mid)
mon_t *mid;

int mon_destroy(mid)
mon_t mid;

int mon_enter(mid)
mon_t mid;

int mon_exit(mid)
mon_t mid;

int mon_enumerate(vec, maxsize)
mon_t vec[ ];       /* list of all monitors */
int maxsize;        /* max size of vec */

int mon_waiters(mid, owner, vec, maxsize)
mon_t mid;                /* monitor in question */
thread_t *owner;          /* which thread owns the monitor */
thread_t vec[ ];          /* list of blocked threads */
int maxsize;              /* max size of vec */

int mon_cond_enter(mid)
mon_t mid;

int mon_break(mid)
mon_t mid;

void MONITOR(mid)
mon_t mid;

int SAMEMON(m1, m2)
mon_t m1;
mon_t m2;
```

DESCRIPTION

Monitors are used to synchronize access to common resources. Although it is possible (on a uniprocessor) to use knowledge of how scheduling priorities work to serialize access to a resource, monitors (and condition variables) provide a general tool to provide the necessary synchronization.

mon_create( ) creates a new monitor and returns its identity in *mid*. **mon_destroy**( ) destroys a monitor, as well as any conditions bound to it (see **cv_create**(3L)). Because the lifetime of a monitor can transcend the lifetime of the LWP that created it, monitor destruction is not automatic upon LWP destruction.

mon_enter( ) blocks the calling thread (if the monitor is in use) until the monitor becomes free by being exited or by waiting on a condition (see **cv_create**(3L)). Threads unable to gain entry into the monitor are queued for monitor service by the priority of the thread requesting monitor access, FCFS within a priority. Monitor calls may nest. If, while holding monitor M1 a request for monitor M2 is made, M1 will be held until M2 can be acquired.

mon_cond_enter( ) will enter the monitor only if the monitor is not busy. Otherwise, an error is returned.

mon_enter( ) and **mon_cond_enter**( ) will allow a thread which already has the monitor to reenter the monitor. In this case, the nesting level of monitor entries is returned. Thus, the first time a monitor is entered, **mon_enter**( ) returns 0. The next time the monitor is entered, **mon_enter**( ) returns 1. **mon_exit**( ) frees the current monitor and allows the next thread blocked on the monitor (if any) to enter the monitor. However, if a monitor is entered more than once, **mon_exit( )** returns the previous monitor nesting level without freeing the monitor to other threads. Thus, if the monitor was not reentered, **mon_exit( )** returns 0.

mon_enumerate( ) lists all the monitors in the system. The vector supplied is filled in with the ID's of the monitors. *maxsize* is used to avoid exceeding the capacity of the list. If the number of monitors is greater than *maxsize*, only *maxsize* monitor ID's are filled in *vec*.

mon_waiters( ) puts the thread that currently owns the monitor in *owner* and all threads blocked on the monitor in *vec* (subject to the *maxsize* limitation), and returns the number of waiting threads.

mon_break( ) forces the release of a monitor lock not necessarily held by the invoking thread. This enables the next thread blocked on the monitor to enter it.

MONITOR is a macro that can be used at the start of a procedure to indicate that the procedure is a monitor. It uses the exception handling mechanism to ensure that the monitor is exited automatically when the procedure exits. Ordinarily, this single macro replaces paired **mon_enter( )**- **mon_exit( )** calls in a monitor procedure.

The SAMEMON macro is a convenient predicate used to compare two monitors for equality.

Monitor locks are released automatically when the LWP holding them dies. This may have implications for the validity of the monitor invariant (a condition that is always true *outside* of the monitor) if a thread unexpectedly terminates.

RETURN VALUES

mon_create( ) returns the ID of a new monitor.

mon_destroy( ) returns:

0	on success.
−1	on failure.

mon_enter( ) returns the nesting level of the monitor.

mon_exit( ) returns the previous nesting level on success. On failure, it returns −1.

mon_enumerate( ) returns the total number of monitors.

mon_waiters( ) returns the number of threads waiting for the monitor.

mon_cond_enter( ) returns the nesting level of the monitor if the monitor is not busy. If the monitor is busy, it returns −1.

mon_break( ) returns:

0	on success.
−1	on failure.

The macro **SAMEMON( )** returns 1 if the monitors specified by *m1* and *m2* are equal. It returns 0 otherwise.

ERRORS

mon_break( ) will fail if one or more of the following are true:

LE_NONEXIST	Attempt to break lock on non-existent monitor.
LE_NOTOWNED	Attempt to break a monitor lock that is not set.

mon_cond_enter( ) will fail if one or more of the following are true:

LE_INUSE	The requested monitor is being used by another thread.
LE_NONEXIST	Attempt to destroy non-existent monitor.

mon_destroy( ) will fail if one or more of the following are true:

LE_INUSE	Attempt to destroy a monitor that has threads blocked on it.
LE_NONEXIST	Attempt to destroy non-existent monitor.

mon_exit( ) will fail if one or more of the following are true:

LE_INVALIDARG	Attempt to exit a monitor that the thread does not own.
LE_NONEXIST	Attempt to exit non-existent monitor.

SEE ALSO

cv_create(3L)

BUGS

There should be language support to enforce the monitor enter-exit discipline.

NAME

msg_send, msg_recv, msg_reply, MSG_RECVALL, msg_enumsend, msg_enumrecv – LWP send and receive messages

SYNOPSIS

```
#include <lwp/lwp.h>

int msg_send(dest, arg, argsize, res, ressize)
thread_t dest;    /* destination thread */
caddr_t arg;      /* argument buffer */
int argsize;      /* size of argument buffer */
caddr_t res;      /* result buffer */
int ressize;      /* size of result buffer */

int msg_recv(sender, arg, argsize, res, ressize, timeout)
thread_t *sender;          /* value-result: sending thread or agent */
caddr_t *arg;              /* argument buffer */
int *argsize;              /* argument size */
caddr_t *res;              /* result buffer */
int *ressize;              /* result size */
struct timeval *timeout;  /* POLL, INFINITY, else timeout */

int msg_reply(sender)
thread_t sender;/* agent id or thread id */

int msg_enumsend(vec, maxsize)
thread_t vec[ ];  /* list of blocked senders */
int maxsize;

int msg_enumrecv(vec, maxsize)
thread_t vec[ ];  /* list of blocked receivers */
int maxsize;

int MSG_RECVALL(sender, arg, argsize, res, ressize, timeout)
thread_t *sender;
caddr_t *arg;
int *argsize;
caddr_t *res;
int *ressize;
struct timeval *timeout;
```

DESCRIPTION

Each thread queues messages addressed to it as they arrive. Threads may either specify that a particular sender's message is to be received next, or that *any* sender's message may be received next.

msg_send() specifies a message buffer and a reply buffer, and initiates one half of a rendezvous with the receiver. The sender will block until the receiver replies using **msg_reply()**. **msg_recv()** initiates the other half of a rendezvous and blocks the invoking thread until a corresponding **msg_send()** is received. When unblocked by **msg_send()**, the receiver may read the message and generate a reply by filling in the reply buffer and issuing **msg_reply()**. **msg_reply()** unblocks the sender. Once a reply is sent, the receiver should no longer access either the message or reply buffer.

In **msg_send()**, *argsize* specifies the size in bytes of the argument buffer *argbuf*, which is intended to be a read-only (to the receiver) buffer. *ressize* specifies the size in bytes of the result buffer *resbuf*, which is intended to be a write-only (to the receiver) buffer. *dest* is the thread that is the target of the send.

msg_recv( ) blocks the receiver until:

- A message from the agent or thread bound to *sender* has been sent to the receiver or,
- *sender* points to a THREADNULL-valued variable and *any* message has been sent to the receiver from a thread or agent, or,
- After the time specified by *timeout* elapses and no message is received.

If *timeout* is **POLL**, **msg_recv**( ) returns immediately, returning success if the message expected has arrived; otherwise an error is returned. If *timeout* is **INFINITY**, **msg_recv**( ) blocks forever or until the expected message arrives. If *timeout* is any other value **msg_recv**( ) blocks for the time specified by *timeout* or until the expected message arrives, whichever comes first. When **msg_recv**( ) returns, *sender* is filled in with the identity of the sending thread or agent, and the buffer addresses and sizes specified by the matching send are stored in *arg*, *argsize*, *res*, and *ressize*.

msg_enumsend( ) and **msg_enumrecv**( ) are used to list all of the threads blocked on sends (awaiting a reply) and receives (awaiting a send), respectively. The value returned is the number of such blocked threads. The vector supplied by the client is filled in (subject to the *maxsize* limitation) with the ID's of the blocked threads. *maxsize* is used to avoid exceeding the capacity of the list. If the number of threads blocked on sends or receives is greater than *maxsize*, only *maxsize* thread ID's are filled in *vec*. If *maxsize* is 0, just the total number of blocked threads is returned.

sender in **msg_recv**( ) is a reference parameter. If you wish to receive from *any* sender, be sure to reinitialize the thread *sender* points to as THREADNULL before each use (do not use the address of THREADNULL for the sender). Alternatively, use the **MSG_RECVALL**() macro. This macro has the same parameters as **msg_recv**( ), but ensures that the sender is properly initialized to allow receipt from any sender. **MSG_RECVALL**() returns the result from **msg_recv**.

RETURN VALUES

msg_send( ), **msg_recv**( ), **MSG_RECVALL**( ) and **msg_reply**( ) return:

0 on success.

−1 on failure.

msg_enumsend( ) returns the number of threads blocked on **msg_send**( ).

msg_enumrecv( ) returns the number of threads blocked on **msg_recv**( ).

ERRORS

msg_recv( ) will fail if one or more of the following is true:

LE_INVALIDARG An illegal timeout was specified.

The sender address is that of THREADNULL.

LE_NONEXIST The specified thread or agent does not exist.

LE_TIMEOUT Timed out before message arrived.

msg_reply( ) will fail if one or more of the following is true:

LE_NONEXIST Attempt to reply to a sender that does not exist or has terminated.

LE_NOWAIT Attempt to reply to a sender that is not expecting a reply.

msg_send( ) will fail if one or more of the following is true:

LE_INVALIDARG Attempt to send a message to yourself.

LE_NONEXIST The specified destination thread does not exist or has terminated.

NAME

pod_getmaxpri, pod_getmaxsize, pod_setmaxpri – control LWP scheduling priority

SYNOPSIS

int pod_getmaxpri( )

int pod_getmaxsize( )

int pod_setmaxpri(maxprio)
int maxprio;

DESCRIPTION

The LWP library is self-initializing: the first time you use a primitive that requires threads to be supported, *main* is automatically converted into a thread. A pod will terminate when all client-created lightweight threads (including the thread bound to *main*) are dead.

By default, only a single priority (MINPRIO) is available. However, by using **pod_setmaxpri( )**, you can make an arbitrary number (up to the limit imposed by the implementation) of priorities available. The *main* thread will receive the highest available scheduling priority at the time of initialization. By using **pod_setmaxpri( )** before any other LWP primitives, you can ensure that main will receive the same priority as the argument to **pod_setmaxpri( )**. **pod_setmaxpri( )** can be called repeatedly, as long as the number of scheduling priorities (*maxprio*) increases with each call.

pod_getmaxpri( ) returns the current number of available priorities. Priorities are numbered from 1 (MINPRIO) to MAXPRIO.

The implementation-dependent maximum number of priorities available can be retrieved using **pod_getmaxsize( )**. This value will never be less than 255.

RETURN VALUES

pod_getmaxpri( ) returns the number of priority levels set by the most recent call to **pod_setmaxpri( )**.

pod_getmaxsize( ) returns the maximum number of priorities your system supports.

pod_setmaxpri( ) returns:

0	on success.
–1	on failure.

ERRORS

pod_setmaxpri( ) will fail if one or more of the following are true:

LE_INVALIDARG	Attempt to allocate more priorities than supported.
LE_NOROOM	No internal memory left to create pod.

NAME

intro – introduction to mathematical library functions and constants

SYNOPSIS

#include <sys/ieeefp.h>

#include <floatingpoint.h>

#include <math.h>

DESCRIPTION

The include file **<math.h>** contains declarations of all the functions described in Section 3M that are implemented in the math library, **libm**. C programs should be linked with the **–lm** option in order to use this library.

<sys/ieeefp.h> and **<floatingpoint.h>** define certain types and constants used for **libm** exception handling, conforming to ANSI/IEEE Std 754-1985, the *IEEE Standard for Binary Floating-Point Arithmetic*.

ACKNOWLEDGEMENT

The Sun version of **libm** is based upon and developed from ideas embodied and codes contained in 4.3 BSD, which may not be compatible with earlier BSD or UNIX implementations.

IEEE ENVIRONMENT

The IEEE Standard specifies modes for rounding direction, precision, and exception trapping, and status reflecting accrued exceptions. These modes and status constitute the IEEE run-time environment. On Sun-2 and Sun-3 systems without 68881 floating-point co-processors, only the default rounding direction to nearest is available, only the default non-stop exception handling is available, and accrued exception bits are not maintained.

IEEE EXCEPTION HANDLING

The IEEE Standard specifies exception handling for **aint**, **ceil**, **floor**, **irint**, **remainder**, **rint**, and **sqrt**, and suggests appropriate exception handling for **fp_class**, **copysign**, **fabs**, **finite**, **fmod**, **isinf**, **isnan**, **ilogb**, **ldexp**, **logb**, **nextafter**, **scalb**, **scalbn** and **signbit**, but does not specify exception handling for the other **libm** functions.

For these other unspecified functions the spirit of the IEEE Standard is generally followed in **libm** by handling invalid operand, singularity (division by zero), overflow, and underflow exceptions, as much as possible, in the same way they are handled for the fundamental floating-point operations such as addition and multiplication.

These unspecified functions are usually not quite correctly rounded, may not observe the optional rounding directions, and may not set the inexact exception correctly.

SYSTEM V EXCEPTION HANDLING

The *System V Interface Definition* (SVID) specifies exception handling for some **libm** functions: **j0**(), **j1**(), **jn**(), **y0**(), **y1**(), **yn**(), **exp**(), **log**(), **log10**(), **pow**(), **sqrt**(), **hypot**(), **lgamma**(), **sinh**(), **cosh**(), **sin**(), **cos**(), **tan**(), **asin**(), **acos**(), and **atan2**(). See **matherr**(3M) for a discussion of the extent to which Sun's implementation of **libm** follows the SVID when it is consistent with the IEEE Standard and with hardware efficiency.

LIST OF MATH LIBRARY FUNCTIONS

Name	**Appears on Page**	**Description**
–	**bessel**(3M)	Bessel functions
–	**frexp**(3M)	floating-point analysis
–	**hyperbolic**(3M)	hyperbolic functions
–	**ieee_functions**(3M)	IEEE classification
–	**ieee_test**(3M)	IEEE tests for compliance
–	**ieee_values**(3M)	returns double-precision IEEE infinity
–	**trig**(3M)	trigonometric functions
acos	**trig**(3M)	trigonometric functions

acosh	**hyperbolic**(3M)	hyperbolic functions
aint	**rint**(3M)	round to integral value in floating-point or integer format
anint	**rint**(3M)	round to integral value in floating-point or integer format
annuity	**exp**(3M)	exponential, logarithm, power
asin	**trig**(3M)	trigonometric functions
asinh	**hyperbolic**(3M)	hyperbolic functions
atan	**trig**(3M)	trigonometric functions
atan2	**trig**(3M)	trigonometric functions
atanh	**hyperbolic**(3M)	hyperbolic functions
cbrt	**sqrt**(3M)	cube root, square root
ceil	**rint**(3M)	round to integral value in floating-point or integer format
compound	**exp**(3M)	exponential, logarithm, power
copysign	**ieee_functions**(3M)	miscellaneous functions for IEEE arithmetic
cos	**trig**(3M)	trigonometric functions
cosh	**hyperbolic**(3M)	hyperbolic functions
erf	**erf**(3M)	error functions
erfc	**erf**(3M)	error functions
exp	**exp**(3M)	exponential, logarithm, power
exp2	**exp**(3M)	exponential, logarithm, power
exp10	**exp**(3M)	exponential, logarithm, power
expm1	**exp**(3M)	exponential, logarithm, power
fabs	**ieee_functions**(3M)	miscellaneous functions for IEEE arithmetic
finite	**ieee_functions**(3M)	miscellaneous functions for IEEE arithmetic
floor	**rint**(3M)	round to integral value in floating-point or integer format
fmod	**ieee_functions**(3M)	miscellaneous functions for IEEE arithmetic
fp_class	**ieee_functions**(3M)	miscellaneous functions for IEEE arithmetic
frexp	**frexp**(3M)	traditional UNIX functions
HUGE	**ieee_values**(3M)	functions that return extreme values of IEEE arithmetic
HUGE_VAL	**ieee_values**(3M)	functions that return extreme values of IEEE arithmetic
hypot	**hypot**(3M)	Euclidean distance
ieee_flags	**ieee_flags**(3M)	mode and status function for IEEE standard arithmetic
ieee_functions	**ieee_functions**(3M)	miscellaneous functions for IEEE arithmetic
ieee_handler	**ieee_handler**(3M)	IEEE exception trap handler function
ieee_test	**ieee_test**(3M)	IEEE test functions for verifying standard compliance
ieee_values	**ieee_values**(3M)	functions that return extreme values of IEEE arithmetic
ilogb	**ieee_functions**(3M)	miscellaneous functions for IEEE arithmetic
infinity	**ieee_values**(3M)	functions that return extreme values of IEEE arithmetic
irint	**rint**(3M)	round to integral value in floating-point or integer format
isinf	**ieee_functions**(3M)	miscellaneous functions for IEEE arithmetic
isnan	**ieee_functions**(3M)	miscellaneous functions for IEEE arithmetic
isnormal	**ieee_functions**(3M)	miscellaneous functions for IEEE arithmetic
issubnormal	**ieee_functions**(3M)	miscellaneous functions for IEEE arithmetic
iszero	**ieee_functions**(3M)	miscellaneous functions for IEEE arithmetic
j0	**bessel**(3M)	Bessel functions
j1	**bessel**(3M)	Bessel functions
jn	**bessel**(3M)	Bessel functions
ldexp	**frexp**(3M)	traditional UNIX functions
lgamma	**lgamma**(3M)	log gamma function
log	**exp**(3M)	exponential, logarithm, power
log2	**exp**(3M)	exponential, logarithm, power
log10	**exp**(3M)	exponential, logarithm, power
log1p	**exp**(3M)	exponential, logarithm, power
logb	**ieee_test**(3M)	IEEE test functions for verifying standard compliance

matherr	**matherr**(3M)	math library exception-handling function
max_normal	**ieee_values**(3M)	functions that return extreme values of IEEE arithmetic
max_subnormal	**ieee_values**(3M)	functions that return extreme values of IEEE arithmetic
min_normal	**ieee_values**(3M)	functions that return extreme values of IEEE arithmetic
min_subnormal	**ieee_values**(3M)	functions that return extreme values of IEEE arithmetic
modf	**frexp**(3M)	traditional UNIX functions
nextafter	**ieee_functions**(3M)	miscellaneous functions for IEEE arithmetic
nint	**rint**(3M)	round to integral value in floating-point or integer format
pow	**exp**(3M)	exponential, logarithm, power
quiet_nan	**ieee_values**(3M)	functions that return extreme values of IEEE arithmetic
remainder	**ieee_functions**(3M)	miscellaneous functions for IEEE arithmetic
rint	**rint**(3M)	round to integral value in floating-point or integer format
scalb	**ieee_test**(3M)	IEEE test functions for verifying standard compliance
scalbn	**ieee_functions**(3M)	miscellaneous functions for IEEE arithmetic
signaling_nan	**ieee_values**(3M)	functions that return extreme values of IEEE arithmetic
signbit	**ieee_functions**(3M)	miscellaneous functions for IEEE arithmetic
significant	**ieee_test**(3M)	IEEE test functions for verifying standard compliance
sin	**trig**(3M)	trigonometric functions
single_precision	**single_precision**(3M)	single-precision access to libm functions
sinh	**hyperbolic**(3M)	hyperbolic functions
sqrt	**sqrt**(3M)	cube root, square root
tan	**trig**(3M)	trigonometric functions
tanh	**hyperbolic**(3M)	hyperbolic functions
y0	**bessel**(3M)	Bessel functions
y1	**bessel**(3M)	Bessel functions
yn	**bessel**(3M)	Bessel functions

NAME

j0, j1, jn, y0, y1, yn – Bessel functions

SYNOPSIS

#include <math.h>

double j0(x)
double x;

double j1(x)
double x;

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

double y0(x)
double x;

double y1(x)
double x;

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

DESCRIPTION

These functions calculate Bessel functions of the first and second kinds for real arguments and integer orders.

SEE ALSO

exp(3M)

DIAGNOSTICS

The functions *y0*, *y1*, and *yn* have logarithmic singularities at the origin, so they treat zero and negative arguments the way *log* does, as described in **exp**(3M). Such arguments are unexceptional for *j0*, *j1*, and *jn*.

NAME

erf, erfc – error functions

SYNOPSIS

```
#include <math.h>

double erf(x)
double x;

double erfc(x)
double x;
```

DESCRIPTION

erf(x) returns the error function of *x*; where **erf** (x):= $(2/\sqrt{\pi}) \int_0^x \exp(-t^2)\,dt$.

erfc(x) returns 1.0–erf (x), computed however by other methods that avoid cancellation for large *x*.

NAME

exp, expm1, exp2, exp10, log, log1p, log2, log10, pow, compound, annuity – exponential, logarithm, power

SYNOPSIS

```
#include <math.h>

double exp(x)
double x;

double expm1(x)
double x;

double exp2(x)
double x;

double exp10(x)
double x;

double log(x)
double x;

double log1p(x)
double x;

double log2(x)
double x;

double log10(x)
double x;

double pow(x, y)
double x, y;

double compound(r, n)
double r, n;

double annuity(r, n)
double r, n;
```

DESCRIPTION

exp( ) returns the exponential function *e**x*.

expm1( ) returns *e**x–1* accurately even for tiny *x*.

exp2( ) and **exp10( )** return *2**x* and *10**x* respectively.

log( ) returns the natural logarithm of *x*.

log1p( ) returns log(1+x) accurately even for tiny *x*.

log2( ) and **log10( )** return the logarithm to base 2 and 10 respectively.

pow( ) returns *x**y*. **pow(***x* **,0.0)** is 1 for all x, in conformance with 4.3BSD, as discussed in the *Numerical Computation Guide*.

compound( ) and **annuity( )** are functions important in financial computations of the effect of interest at periodic rate *r* over *n* periods. **compound(***r*, *n***)** computes *(1+r)**n*, the compound interest factor. Given an initial principal *P0*, its value after *n* periods is just **Pn** = *P0* * **compound(***r*, *n***)**. **annuity(***r*, *n***)** computes *(1 – (1+r)**–n)/r*, the present value of annuity factor. Given an initial principal *P0*, the equivalent periodic payment is just **p** = *P0* / **annuity(***r*, *n***)**. **compound( )** and **annuity( )** are computed using **log1p( )** and **expm1( )** to avoid gratuitous inaccuracy for small-magnitude *r*. **compound( )** and **annuity( )** are not defined for *r* <= –1.

Thus a principal amount *P0* placed at 5% annual interest compounded quarterly for 30 years would yield

P30 = *P0* *** compound(.05/4, 30.0 * 4)**

while a conventional fixed-rate 30-year home loan of amount *P0* at 10% annual interest would be amortized by monthly payments in the amount

p = *P0* **/ annuity(.10/12, 30.0 * 12)**

SEE ALSO

matherr(3M)

DIAGNOSTICS

All these functions handle exceptional arguments in the spirit of ANSI/IEEE Std 754-1985. Thus for x == ±0, **log**(*x*) is −∞ with a division by zero exception; for x < 0, including −∞, **log**(*x*) is a quiet NaN with an invalid operation exception; for x == +∞ or a quiet NaN, **log**(*x*) is x without exception; for x a signaling NaN, **log**(*x*) is a quiet NaN with an invalid operation exception; for x == 1, **log**(*x*) is 0 without exception; for any other positive x, **log**(*x*) is a normalized number with an inexact exception.

In addition, **exp**(), **exp2**(), **exp10**(), **log**(), **log2**(), **log10**() and **pow**() may also set **errno** and call **matherr**(3M).

NAME

frexp, modf, ldexp – traditional UNIX functions

SYNOPSIS

```
#include <math.h>

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

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

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

DESCRIPTION

These functions are provided for compatibility with other UNIX system implementations. They are not used internally in **libm** or **libc**. Better ways to accomplish similar ends may be found in **ieee_functions**(3M) and **rint**(3M).

ldexp(*x*,*n*) returns *x* * *2***n* computed by exponent manipulation rather than by actually performing an exponentiation or a multiplication. Note: **ldexp**(*x*,*n*) differs from **scalbn**(*x*,*n*), defined in **ieee_functions**(3M), only that in the event of IEEE overflow and underflow, **ldexp**(*x*,*n*) sets **errno** to ERANGE.

Every non-zero number can be written uniquely as *x* * *2***n*, where the significant *x* is in the range *0.5 <= |x| < 1.0* and the exponent *n* is an integer. The function **frexp**( ) returns the significant of a double *value* as a double quantity, *x*, and stores the exponent *n*, indirectly through *eptr*. If *value* == 0, both results returned by **frexp**( ) are 0.

modf( ) returns the fractional part of *value* and stores the integral part indirectly through *iptr*. Thus the argument *value* and the returned values **modf**( ) and **iptr* satisfy

(**iptr* + *modf*) == *value*

and both results have the same sign as *value*. The definition of **modf**( ) varies among UNIX system implementations, so avoid **modf**( ) in portable code.

The results of **frexp**( ) and **modf**( ) are not defined when *value* is an IEEE infinity or NaN.

SEE ALSO

ieee_functions(3M), **rint**(3M)

NAME

sinh, cosh, tanh, asinh, acosh, atanh – hyperbolic functions

SYNOPSIS

```
#include <math.h>

double sinh(x)
double x;

double cosh(x)
double x;

double tanh(x)
double x;

double asinh(x)
double x;

double acosh(x)
double x;

double atanh(x)
double x;
```

DESCRIPTION

These functions compute the designated direct and inverse hyperbolic functions for real arguments. They inherit much of their roundoff error from **expm1()** and **log1p**, described in **exp**(3M).

DIAGNOSTICS

These functions handle exceptional arguments in the spirit of ANSI/IEEE Std 754-1985. Thus **sinh()** and **cosh()** return ±∞ on overflow, **acosh()** returns a NaN if its argument is less than 1, and **atanh()** returns a NaN if its argument has absolute value greater than 1. In addition, **sinh**,**cosh**, and **tanh()** may also set **errno** and call **matherr**(3M).

SEE ALSO

exp(3M), **matherr**(3M)

NAME

hypot – Euclidean distance

SYNOPSIS

```
#include <math.h>

double hypot(x, y)
double x, y;
```

DESCRIPTION

hypot( ) returns

sqrt(x*x + y*y) ,

taking precautions against unwarranted IEEE exceptions. On IEEE overflow, **hypot**( ) may also set **errno** and call **matherr**(3M). **hypot(±∞, y)** is +∞ for any y, even a NaN, and is exceptional only for a signaling NaN.

hypot(*x*,*y*) and **atan2**(*y*,*x*) (see **trig**(3M)) convert rectangular coordinates (*x*,*y*) to polar (*r*,θ); **hypot**( ) computes *r*, the modulus or radius.

SEE ALSO

trig(3M), **matherr**(3M)

NAME

ieee_flags – mode and status function for IEEE standard arithmetic

SYNOPSIS

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

int ieee_flags(action, mode, in, out)
char *action, *mode, *in, **out;
```

DESCRIPTION

This function provides easy access to the modes and status required to fully exploit ANSI/IEEE Std 754-1985 arithmetic in a C program. All arguments are pointers to strings. Results arising from invalid arguments and invalid combinations are undefined for efficiency.

There are four types of *action*: **get**, **set**, **clear** and **clearall**. There are three valid settings for *mode*, two corresponding to modes of IEEE arithmetic:

direction	current rounding direction mode
precision	current rounding precision mode

and one corresponding to status of IEEE arithmetic:

exception	accrued exception-occurred status

There are fourteen types of *in* and *out*:

nearest	round toward nearest
tozero	round toward zero
negative	round toward negative infinity
positive	round toward positive infinity
extended	
double	
single	
inexact	
division	division by zero exception
underflow	
overflow	
invalid	
all	all five exceptions above
common	invalid, overflow, and division exceptions

Note: **all** and **common** only make sense with **set** or **clear**.

For **clearall**, **ieee_flags( )** returns 0 and restores all default modes and status. Nothing will be assigned to *out*. Thus

```
char *mode, *out, *in;
ieee_flags("clearall", mode, in, &out);
```

set rounding direction to **nearest**, rounding precision to **extended**, and all accrued exception-occurred status to zero.

For **clear**, **ieee_flags( )** returns 0 and restores the default mode or status. Nothing will be assigned to *out*. Thus

```
char *out, *in;
ieee_flags("clear", "direction", in, &out);      . . . set rounding direction to round to nearest.
```

For **set**, **ieee_flags( )** returns 0 if the action is successful and 1 if the corresponding required status or mode is not available (for instance, not supported in hardware). Nothing will be assigned to *out*. Thus

```
char *out, *in;
ieee_flags ("set", "direction", "tozero", &out);      set rounding direction to round toward zero;
```

For **get**, we have the following cases:

Case 1: *mode* is **direction**. In that case, *out* returns one of the four strings **nearest**, **tozero**, **positive**, **negative**, and **ieee_flags( )** returns a value corresponding to *out* according to the **enum fp_direction_type** defined in **<sys/ieeefp.h>**.

Case 2: *mode* is **precision**. In that case, *out* returns one of the three strings **extended**, **double** and **single**, and **ieee_flags( )** returns a value corresponding to *out* according to the **enum fp_precision_type** defined in **<sys/ieeefp.h>**.

Case 3: *mode* is **exception**. In that case, *out* returns

not available if information on exception is not available.

no exception if no accrued exception.

the accrued exception that has the highest priority according to the following list:

the exception named by *in*
invalid
overflow
division
underflow
inexact

In this case **ieee_flags( )** returns a five or six bit value where each bit (see **enum fp_exception_type** in **<sys/ieeefp.h>**) corresponds to an exception-occurred accrued status flag: 0 = off, 1 = on. The bit corresponding to a particular exception varies among architectures (see **<sys/ieeefp.h>**).

Example:

```
char *out; int k, ieee_flags( );
ieee_flags("clear", "exception", "all", &out);      /* clear all accrued exceptions */
. . .
code that generates three exceptions: overflow, invalid, inexact
. . .
k = ieee_flags("get", "exception", "overflow", &out);
```

then *out* is **overflow**, and on a Sun-3, *k* is 25.

NAME

ieee_functions, fp_class, finite, ilogb, isinf, isnan, isnormal, issubnormal, iszero, signbit, copysign, fabs, fmod, nextafter, remainder, scalbn – appendix and related miscellaneous functions for IEEE arithmetic

SYNOPSIS

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

enum fp_class_type fp_class(x)
double x;

int finite(x)
double x;

int ilogb(x)
double x;

int isinf(x)
double x;

int isnan(x)
double x;

int isnormal(x)
double x;

int issubnormal(x)
double x;

int iszero(x)
double x;

int signbit(x)
double x;

void ieee_retrospective(f)
FILE *f;

void nonstandard_arithmetic()

void standard_arithmetic()

double copysign(x,y)
double x, y;

double fabs(x)
double x;

double fmod(x,y)
double x, y;

double nextafter(x,y)
double x, y;

double remainder(x,y)
double x, y;

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

DESCRIPTION

Most of these functions provide capabilities required by ANSI/IEEE Std 754-1985 or suggested in its appendix.

fp_class(*x*) corresponds to the IEEE's **class( ) and** classifies *x* as zero, subnormal, normal, ∞, or quiet or signaling *NaN*. **<floatingpoint.h>** defines **enum fp_class_type**. The following functions return 0 if the indicated condition is not satisfied:

finite(*x*)	returns 1 if x is zero, subnormal or normal
isinf(*x*)	returns 1 if *x* is ∞
isnan(*x*)	returns 1 if *x* is *NaN*
isnormal(*x*)	returns 1 if *x* is normal
issubnormal(*x*)	returns 1 if *x* is subnormal
iszero(*x*)	returns 1 if *x* is zero
signbit(*x*)	returns 1 if *x*'s sign bit is set

ilogb(*x*) returns the unbiased exponent of *x* in integer format. **ilogb(±∞) = +MAXINT** and **ilogb(0) = –MAXINT**; **<values.h>** defines MAXINT as the largest int. **ilogb**(*x*) never generates an exception. When *x* is subnormal, **ilogb**(*x*) returns an exponent computed as if *x* were first normalized.

ieee_retrospective(*f*) prints a message to the **FILE** *f* listing all IEEE accrued exception-occurred bits currently on, unless no such bits are on or the only one on is "inexact". It's intended to be used at the end of a program to indicate whether some IEEE floating-point exceptions occurred that might have affected the result.

standard_arithmetic() and **nonstandard_arithmetic**() are meaningful on systems that provide an alternative faster mode of floating-point arithmetic that does not conform to the default IEEE Standard. Nonstandard modes vary among implementations; nonstandard mode may, for instance, result in setting subnormal results to zero or in treating subnormal operands as zero, or both, or something else. **standard_arithmetic**() reverts to the default standard mode. On systems that provide only one mode, these functions have no effect.

copysign(*x*,*y*) returns *x* with *y*'s sign bit.

fabs(*x*) returns the absolute value of *x*.

nextafter(*x*,*y*) returns the next machine representable number from *x* in the direction *y*.

remainder(*x*, *y*) and **fmod**(*x*, *y*) return a remainder of *x* with respect to *y*; that is, the result *r* is one of the numbers that differ from *x* by an integral multiple of *y*. Thus *(x – r)/y* is an integral value, even though it might exceed MAXINT if it were explicitly computed as an int. Both functions return one of the two such r smallest in magnitude. **remainder**(*x*, *y*) is the operation specified in ANSI/IEEE Std 754-1985; the result of **fmod**(*x*, *y*) may differ from **remainder**( )'s result by ±*y*. The magnitude of **remainder**'s result can not exceed half that of *y*; its sign might not agree with either *x* or *y*. The magnitude of **fmod**( )'s result is less than that of *y*; its sign agrees with that of *x*. Neither function can generate an exception as long as both arguments are normal or subnormal. **remainder**(*x*, **0**), **fmod**(*x*, **0**), **remainder**(∞, *y*), and **fmod**(∞, *y*) are invalid operations that produce a *NaN*.

scalbn(*x*, *n*) returns *x** **2**n** computed by exponent manipulation rather than by actually performing an exponentiation or a multiplication. Thus

$$1 \le \mathbf{scalbn}(\mathbf{fabs}(x), -\mathbf{ilogb}(x)) < 2$$

for every *x* except 0, ∞, and *NaN*.

SEE ALSO

floatingpoint(3), **ieee_flags**(3M), **matherr**(3M)

NAME

ieee_handler – IEEE exception trap handler function

SYNOPSIS

#include <floatingpoint.h>

int ieee_handler(action,exception,hdl)
char action[ ], exception[ ];
sigfpe_handler_type hdl;

DESCRIPTION

This function provides easy exception handling to exploit ANSI/IEEE Std 754-1985 arithmetic in a C program. The first two arguments are pointers to strings. Results arising from invalid arguments and invalid combinations are undefined for efficiency.

There are three types of *action* : **get, set,** and **clear**. There are five types of *exception* :

inexact	
division	... division by zero exception
underflow	
overflow	
invalid	
all	... all five exceptions above
common	... invalid, overflow, and division exceptions

Note: **all** and **common** only make sense with **set** or **clear**.

hdl contains the address of a signal-handling routine. **<floatingpoint.h>** defines *sigfpe_handler_type*.

get will return the location of the current handler routine for *exception* cast to an int. **set** will set the routine pointed at by **hdl** to be the handler routine and at the same time enable the trap on *exception*, except when **hdl** == **SIGFPE_DEFAULT** or **SIGFPE_IGNORE**; then **ieee_handler**() will disable the trap on *exception*. When **hdl** == **SIGFPE_ABORT**, any trap on *exception* will dump core using **abort**(3). **clear all** disables trapping on all five exceptions.

Two steps are required to intercept an IEEE-related SIGFPE code with **ieee_handler**:

1) Set up a handler with **ieee_handler**.

2) Perform a floating-point operation that generates the intended IEEE exception.

Unlike **sigfpe**(3), **ieee_handler**( ) also adjusts floating-point hardware mode bits affecting IEEE trapping. For **clear**, **set SIGFPE_DEFAULT**, or **set SIGFPE_IGNORE**, the hardware trap is disabled. For any other **set** , the hardware trap is enabled.

SIGFPE signals can be handled using **sigvec**(2), **signal**(3V), **sigfpe**(3), or **ieee_handler**(3M). In a particular program, to avoid confusion, use only one of these interfaces to handle SIGFPE signals.

DIAGNOSTICS

ieee_handler( ) normally returns 0 for **set** . 1 will be returned if the action is not available (for instance, not supported in hardware). For **get** , the address of the current handler is returned, cast to an int.

EXAMPLE

A user-specified signal handler might look like this:

```
void sample_handler(sig, code, scp, addr)
int sig;          /* sig == SIGFPE always */
int code;
struct sigcontext *scp;
char *addr;
{
        /*
         * Sample user-written sigfpe code handler.
         * Prints a message and continues.
         * struct sigcontext is defined in <signal.h>.
         */
        printf("ieee exception code %x occurred at pc %X \n", code, scp->sc_pc);
}
```

and it might be set up like this:

```
extern void sample_handler();
main()
{
        sigfpe_handler_type hdl, old_handler1, old_handler2;
        /*
         * save current overflow and invalid handlers
         */
        old_handler1 = (sigfpe_handler_type) ieee_handler("get", "overflow", old_handler1);
        old_handler2 = (sigfpe_handler_type) ieee_handler("get", "invalid", old_handler2);
        /*
         * set new overflow handler to sample_handler() and set new
         * invalid handler to SIGFPE_ABORT (abort on invalid)
         */
        hdl = (sigfpe_handler_type) sample_handler;
        if (ieee_handler("set", "overflow", hdl) != 0)
                printf("ieee_handler can't set overflow \n");
        if (ieee_handler("set", "invalid", SIGFPE_ABORT) != 0)
                printf("ieee_handler can't set invalid \n");
        ...
        /*
         * restore old overflow and invalid handlers
         */
        ieee_handler("set", "overflow", old_handler1);
        ieee_handler("set", "invalid", old_handler2);
}
```

SEE ALSO

sigvec(2), abort(3), floatingpoint(3), sigfpe(3), signal(3V)

NAME

ieee_test, logb, scalb, significant – IEEE test functions for verifying standard compliance

SYNOPSIS

```
#include <math.h>

double logb(x)
double x;

double scalb(x,y)
double x; double y;

double significant(x)
double x;
```

DESCRIPTION

These functions allow users to verify compliance to ANSI/IEEE Std 754-1985 by running certain test vectors distributed by the University of California. Their use is not otherwise recommended; instead use **scalbn(*x*,*n*)** and **ilogb(*x*)** described in **ieee_functions**(3M). See the *Numerical Computation Guide* for details.

logb(*x*) returns the unbiased exponent of *x* in floating-point format, for exercising the logb(L) test vector. **logb(±∞) = +∞; logb(0) = –∞** with a division by zero exception. **logb(*x*)** differs from **ilogb(*x*)** in returning a result in floating-point rather than integer format, in sometimes signaling IEEE exceptions, and in not normalizing subnormal *x*.

scalb(*x*,(double)n*)* returns *x * 2**n* computed by exponent manipulation rather than by actually performing an exponentiation or a multiplication, for exercising the scalb(S) test vector. Thus

0 ≤ scalb(fabs(*x*),–logb(*x*)) < 2

for every *x* except 0, ∞ and *NaN*. **scalb(*x*,*y*)** is not defined when *y* is not an integral value. **scalb(*x*,*y*)** differs from **scalbn(*x*,n*)*** in that the second argument is in floating-point rather than integer format.

significant(*x*) computes just

scalb(*x*, (double) -ilogb(*x*)),

for exercising the fraction-part(F) test vector.

FILES

/usr/lib/libm.a

SEE ALSO

floatingpoint(3), **ieee_values**(3M), **ieee_functions**(3M), **matherr**(3M)

NAME

ieee_values, min_subnormal, max_subnormal, min_normal, max_normal, infinity, quiet_nan, signaling_nan, HUGE, HUGE_VAL – functions that return extreme values of IEEE arithmetic

SYNOPSIS

#include <math.h>

double min_subnormal( )

double max_subnormal( )

double min_normal( )

double max_normal( )

double infinity( )

double quiet_nan(n)
long n;

double signaling_nan(n)
long n;

#define HUGE (infinity( ))

#define HUGE_VAL (infinity( ))

DESCRIPTION

These functions return special values associated with ANSI/IEEE Std 754-1985 double-precision floating-point arithmetic: the smallest and largest positive subnormal numbers, the smallest and largest positive normalized numbers, positive infinity, and a quiet and signaling NaN. The long parameters *n* to **quiet_nan(*n*)** and **signaling_nan(*n*)** are presently unused but are reserved for future use to specify the significant of the returned NaN.

None of these functions are affected by IEEE rounding or trapping modes or generate any IEEE exceptions.

The macro HUGE returns +∞ in accordance with previous SunOS releases. The macro HUGE_VAL returns +∞ in accordance with the System V Interface Definition.

FILES

/usr/lib/libm.a

SEE ALSO

ieee_functions(3M)

NAME

lgamma – log gamma function

SYNOPSIS

```
#include <math.h>

extern int signgam;

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 Γ(x).

IDIOSYNCRASIES

Do *not* use the expression **signgam*exp(lgamma(x))** to compute 'g := Γ(x)'. Instead compute **lgamma( )** first:

```
lg = lgamma(x); g = signgam*exp(lg);
```

only after **lgamma( )** has returned can **signgam** be correct. Note: Γ(x) must overflow when *x* is large enough, underflow when −*x* is large enough, and generate a division by zero exception at the singularities *x* a nonpositive integer. In addition, **lgamma( )** may also set **errno** and call **matherr**(3M).

SEE ALSO

matherr(3M)

NAME

matherr – math library exception-handling function

SYNOPSIS

```
#include <math.h>

int matherr(exc)
struct exception *exc;
```

DESCRIPTION

The SVID (*System V Interface Definition*) specifies that certain **libm** functions call **matherr**( ) when exceptions are detected. Users may define their own mechanisms for handling exceptions, by including a function named **matherr**( ) in their programs. **matherr**( ) is of the form described above. When an exception occurs, a pointer to the exception structure *exc* will be passed to the user-supplied **matherr**( ) function. This structure, which is defined in the **<math.h>** header file, is as follows:

```
struct exception {
        int type;
        char *name;
        double arg1, arg2, retval;
};
```

The element **type** is an integer describing the type of exception that has occurred, from the following list of constants (defined in the header file):

DOMAIN	argument domain exception
SING	argument singularity
OVERFLOW	overflow range exception
UNDERFLOW	underflow range exception

The element **name** points to a string containing the name of the function that incurred the exception. The elements **arg1** and **arg2** are the arguments with which the function was invoked. **retval** is set to the default value that will be returned by the function unless the user's **matherr**( ) sets it to a different value.

If the user's **matherr**( ) function returns non-zero, no exception message will be printed, and **errno** will not be set.

If **matherr**( ) is not supplied by the user, the default matherr exception-handling mechanisms, summarized in the table below, will be invoked upon exception:

DOMAIN==fp_invalid
An IEEE NaN is usually returned, **errno** is set to EDOM, and a message is printed on standard error. **pow**(*0.0*,**0.0**) and **atan2(0.0,0.0)** return numerical default results but set **errno** and print the message.

SING==fp_division
An IEEE ∞ of appropriate sign is returned, **errno** is set to EDOM, and a message is printed on standard error.

OVERFLOW==fp_overflow
In the default rounding direction, an IEEE ∞ of appropriate sign is returned. In optional rounding directions, ±**MAXDOUBLE**, the largest finite double-precision number, is sometimes returned instead of ±∞. **errno** is set to ERANGE.

UNDERFLOW==fp_underflow
An appropriately-signed zero, subnormal number, or smallest normalized number is returned, and **errno** is set to ERANGE.

The facilities provided by **matherr**( ) are not available in situations such as compiling on a Sun-3 system with **/usr/lib/f68881/libm.il** or **/usr/lib/ffpa/libm.il**, in which case some **libm** functions are converted to atomic hardware operations. In these cases setting **errno** and calling **matherr**( ) are not worth the adverse performance impact, but regular ANSI/IEEE Std 754-1985 exception handling remains available. In any case **errno** is not a reliable error indicator in that it may be unexpectedly set by a function in a handler for an asynchronous signal.

DEFAULT ERROR HANDLING PROCEDURES

	Types of Errors			
<math.h> type	DOMAIN	SING	OVERFLOW	UNDERFLOW
errno	EDOM	EDOM	ERANGE	ERANGE
IEEE Exception	Invalid Operation	Division by Zero	Overflow	Underflow
<floatingpoint.h> type	fp_invalid	fp_division	fp_overflow	fp_underflow
ACOS, ASIN:	M, NaN	–	–	–
ATAN2(0,0):	M, ±0.0 or ±π	–	–	–
BESSEL: y0, y1, yn (x < 0) y0, y1, yn (x = 0)	 M, NaN –	 – M, −∞	 – –	 – –
COSH, SINH:	–	–	IEEE Overflow	–
EXP:	–	–	IEEE Overflow	IEEE Underflow
HYPOT:	–	–	IEEE Overflow	–
LGAMMA:	–	M, +∞	IEEE Overflow	–
LOG, LOG10: (x < 0) (x = 0)	 M, NaN –	 – M, −∞	 – –	 – –
POW: usual cases (x < 0) ** (y not an integer) 0 ** 0 0 ** (y < 0)	 – M, NaN M, 1.0 –	 – – – M, ±∞	 IEEE Overflow – – –	 IEEE Underflow – – –
SQRT:	M, NaN	–	–	–

ABBREVIATIONS

M	Message is printed (EDOM exception).
NaN	IEEE NaN result and invalid operation exception.
∞	IEEE ∞ result and division-by-zero exception.
IEEE Overflow	IEEE Overflow result and exception.
IEEE Underflow	IEEE Underflow result and exception.
π	Closest machine-representable approximation to pi.

The interaction of IEEE arithmetic and **matherr( )** is not defined when executing under IEEE rounding modes other than the default round to nearest: **matherr( )** may not be called on overflow or underflow, and the Sun-provided **matherr( )** may return results that differ from those in this table.

EXAMPLE

```
#include <math.h>

int
matherr(x)
register struct exception *x;
{
        switch (x–>type) {
        case
                DOMAIN:
                /* change sqrt to return sqrt(–arg1), not NaN */
                if (!strcmp(x–>name, "sqrt")) {
                        x–>retval = sqrt(–x–>arg1);
                        return (0); /* print message and set errno */
        } /* fall through */
        case
                SING:
                /* all other domain or sing exceptions, print message and abort */
                fprintf(stderr, "domain exception in %s\n", x–>name);
                abort( );
                break;
        }
        return (0); /* all other exceptions, execute default procedure */
}
```

NAME

aint, anint, ceil, floor, rint, irint, nint – round to integral value in floating-point or integer format

SYNOPSIS

```
#include <math.h>

double aint(x)
double x;

double anint(x)
double x;

double ceil(x)
double x;

double floor(x)
double x;

double rint(x)
double x;

int irint(x)
double x;

int nint(x)
double x;
```

DESCRIPTION

aint(), **anint()**, **ceil()**, **floor()**, and **rint()** convert a double value into an integral value in double format. They vary in how they choose the result when the argument is not already an integral value. Here an "integral value" means a value of a mathematical integer, which however might be too large to fit in a particular computer's int format. All sufficiently large values in a particular floating-point format are already integral; in IEEE double-precision format, that means all values >= 2**52. Zeros, infinities, and quiet NaNs are treated as integral values by these functions, which always preserve their argument's sign.

aint() returns the integral value between *x* and 0, nearest *x*. This corresponds to IEEE rounding toward zero and to the Fortran generic intrinsic function **aint()**.

anint() returns the nearest integral value to *x*, except halfway cases are rounded to the integral value larger in magnitude. This corresponds to the Fortran generic intrinsic function **anint()**.

ceil() returns the least integral value greater than or equal to *x*. This corresponds to IEEE rounding toward positive infinity.

floor() returns the greatest integral value less than or equal to *x*. This corresponds to IEEE rounding toward negative infinity.

rint() rounds *x* to an integral value according to the current IEEE rounding direction.

irint() converts *x* into int format according to the current IEEE rounding direction.

nint() converts *x* into int format rounding to the nearest int value, except halfway cases are rounded to the int value larger in magnitude. This corresponds to the Fortran generic intrinsic function **nint()**.

NAME

single_precision – single-precision access to libm functions

SYNOPSIS

```
#include <math.h>

FLOATFUNCTIONTYPE r_acos_ (x)
FLOATFUNCTIONTYPE r_acospi_ (x)
FLOATFUNCTIONTYPE r_acosh_ (x)
FLOATFUNCTIONTYPE r_aint_ (x)
FLOATFUNCTIONTYPE r_anint_ (x)
FLOATFUNCTIONTYPE r_annuity_ (x)
FLOATFUNCTIONTYPE r_asin_ (x)
FLOATFUNCTIONTYPE r_asinpi_ (x)
FLOATFUNCTIONTYPE r_asinh_ (x)
FLOATFUNCTIONTYPE r_atan_ (x)
FLOATFUNCTIONTYPE r_atanpi_ (x)
FLOATFUNCTIONTYPE r_atanh_ (x)
FLOATFUNCTIONTYPE r_atan2_ (x,y)
FLOATFUNCTIONTYPE r_atan2pi_ (x,y)
FLOATFUNCTIONTYPE r_cbrt_ (x)
FLOATFUNCTIONTYPE r_ceil_ (x)
enum fp_class_type ir_fp_class_ (x)
FLOATFUNCTIONTYPE r_compound_ (x,y)
FLOATFUNCTIONTYPE r_copysign_ (x,y)
FLOATFUNCTIONTYPE r_cos_ (x)
FLOATFUNCTIONTYPE r_cospi_ (x)
FLOATFUNCTIONTYPE r_cosh_ (x)
FLOATFUNCTIONTYPE r_erf_ (x)
FLOATFUNCTIONTYPE r_erfc_ (x)
FLOATFUNCTIONTYPE r_exp_ (x)
FLOATFUNCTIONTYPE r_expm1_ (x)
FLOATFUNCTIONTYPE r_exp2_ (x)
FLOATFUNCTIONTYPE r_exp10_ (x)
FLOATFUNCTIONTYPE r_fabs_ (x)
int ir_finite_ (x)
FLOATFUNCTIONTYPE r_floor_ (x)
FLOATFUNCTIONTYPE r_fmod_ (x,y)
FLOATFUNCTIONTYPE r_hypot_ (x,y)
int ir_ilogb_ (x)
int ir_irint_ (x)
int ir_isinf_ (x)
int ir_isnan_ (x)
int ir_isnormal_ (x)
int ir_issubnormal_ (x)
int ir_iszero_ (x)
int ir_nint_ (x)
FLOATFUNCTIONTYPE r_infinity_ ()
FLOATFUNCTIONTYPE r_j0_ (x)
FLOATFUNCTIONTYPE r_j1_ (x)
FLOATFUNCTIONTYPE r_jn_ (n,x)
FLOATFUNCTIONTYPE r_lgamma_ (x)
FLOATFUNCTIONTYPE r_logb_ (x)
FLOATFUNCTIONTYPE r_log_ (x)
FLOATFUNCTIONTYPE r_log1p_ (x)
FLOATFUNCTIONTYPE r_log2_ (x)
FLOATFUNCTIONTYPE r_log10_ (x)
FLOATFUNCTIONTYPE r_max_normal_ ()
FLOATFUNCTIONTYPE r_max_subnormal_ ()
FLOATFUNCTIONTYPE r_min_normal_ ()
FLOATFUNCTIONTYPE r_min_subnormal_ ()
FLOATFUNCTIONTYPE r_nextafter_ (x,y)
FLOATFUNCTIONTYPE r_pow_ (x,y)
FLOATFUNCTIONTYPE r_quiet_nan_ (n)
FLOATFUNCTIONTYPE r_remainder_ (x,y)
FLOATFUNCTIONTYPE r_rint_ (x)
FLOATFUNCTIONTYPE r_scalb_ (x,y)
FLOATFUNCTIONTYPE r_scalbn_ (x,n)
FLOATFUNCTIONTYPE r_signaling_nan_ (n)
int ir_signbit_ (x)
FLOATFUNCTIONTYPE r_significant_ (x)
FLOATFUNCTIONTYPE r_sin_ (x)
FLOATFUNCTIONTYPE r_sinpi_ (x)
void r_sincos_ (x,s,c)
void r_sincospi_ (x,s,c)
FLOATFUNCTIONTYPE r_sinh_ (x)
FLOATFUNCTIONTYPE r_sqrt_ (x)
FLOATFUNCTIONTYPE r_tan_ (x)
FLOATFUNCTIONTYPE r_tanpi_ (x)
FLOATFUNCTIONTYPE r_tanh_ (x)
FLOATFUNCTIONTYPE r_y0_ (x)
FLOATFUNCTIONTYPE r_y1_ (x)
FLOATFUNCTIONTYPE r_yn_ (n,x)

float *x, *y, *s, *c
int *n
```

DESCRIPTION

These functions are single-precision versions of certain **libm** functions. Primarily for use by Fortran programmers, these functions may also be used in other languages. The single-precision floating-point results are deviously declared to avoid C's automatic type conversion to double.

FILES

/usr/lib/libm.a

NAME

sqrt, cbrt – cube root, square root

SYNOPSIS

```
#include <math.h>

double cbrt(x)
double x;

double sqrt(x)
double x;
```

DESCRIPTION

sqrt(*x*) returns the square root of *x*, correctly rounded according to ANSI/IEEE 754-1985. In addition, **sqrt**( ) may also set **errno** and call **matherr**(3M).

cbrt(*x*) returns the cube root of *x*. **cbrt**( ) is accurate to within 0.7 *ulps*.

SEE ALSO

matherr(3M)

NAME

sin, cos, tan, asin, acos, atan, atan2 – trigonometric functions

SYNOPSIS

```
#include <math.h>

double sin(x)
double x;

double cos(x)
double x;

void sincos(x, s, c)
double x, *s, *c;

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;

double sinpi(x)
double x;

double cospi(x)
double x;

void sincospi(x, s, c)
double x, *s, *c;

double tanpi(x)
double x;

double asinpi(x)
double x;

double acospi(x)
double x;

double atanpi(x)
double x;

double atan2pi(y, x)
double y, x;
```

DESCRIPTION

sin( ), **cos( )**, **sincos( )**, and **tan( )** return trigonometric functions of radian arguments. The values of trigonometric functions of arguments exceeding π/4 in magnitude are affected by the precision of the approximation to π/2 used to reduce those arguments to the range –π/4 to π/4. Argument reduction may occur in hardware or software; if in software, the variable **fp_pi** defined in **<math.h>** allows changing that precision at run time. Trigonometric argument reduction is discussed in the *Numerical Computation Guide*. Note: **sincos(x,s,c)** allows simultaneous computation of ***s = sin(x)** and ***c = cos(x)**.

asin( ) returns the arc sin in the range –π/2 to π/2.

acos( ) returns the arc cosine in the range 0 to π.

atan( ) returns the arc tangent of *x* in the range −π/2 to π/2.

atan2(*y*,*x*) and **hypot**(x,y) (see **hypot**(3M)) convert rectangular coordinates (*x*,*y*) to polar (*r*,θ); **atan2**( ) computes θ, the argument or phase, by computing an arc tangent of *y*/*x* in the range −π to π. **atan2**(0.0,0.0) is ±0.0 or ±π, in conformance with 4.3BSD, as discussed in the *Numerical Computation Guide*.

sinpi(), **cospi**(), and **tanpi**() avoid range-reduction issues because their definition **sinpi**(x)==**sin**(π∗x) permits range reduction that is fast and exact for all *x*. The corresponding inverse functions compute **asinpi**(x)==**asin**(x)/π. Similarly **atan2pi**(y,x)==**atan2**(y,x)/π.

DIAGNOSTICS

These functions handle exceptional arguments in the spirit of ANSI/IEEE Std 754-1985. **sin**(±∞), **cos**(±∞), **tan**(±∞), or **asin**(*x*) or **acos**(*x*) with |x|>1, return NaN; **sinpi**(*x*) et. al. are similar. In addition, **asin**(), **acos**( ), and **atan2**( ) may also set **errno** and call **matherr**(3M).

SEE ALSO

hypot(3M), **matherr**(3M)

NAME

intro – introduction to RPC service library functions and protocols

DESCRIPTION

These functions constitute the RPC service library. Most of these describe RPC protocols. The PROTOCOL section describes how to access the protocol description file. This file may be compiled with **rpcgen(1)** to produce data definitions and XDR routines. Procompiled versions of header files sometimes exist as **<rpcsvc/*.h>** and precompiled XDR routines and programming interfaces to the protocols sometimes exist in *librpcsvc*. Warning: some of these header files and XDR routines were hand-written because they existed before *rpcgen*. They do not correspond to their protocol description file. In order to get the link editor to load this library, use the **–lrpcsvc** option of cc(1V). Information about the availability of programming interfaces to these protocols is available under PROGRAMMING section of each manual page.

Some routines in the **librpcsvc** library do not correspond to protocols, but are useful utilities for RPC programming. These are distinguished by the presence of the SYNOPSIS section instead of the usual PROTOCOL section.

LIST OF STANDARD RPC SERVICES

Name	**Appears on Page**	**Description**
bootparam	**bootparam**(3R)	bootparam protocol
ether	**ether**(3R)	monitor traffic on the Ethernet
getpublickey	**publickey**(3R)	get public or secret key
getrpcport	**getrpcport**(3R)	get RPC port number
getsecretkey	**publickey**(3R)	get public or secret key
ipalloc	**ipalloc**(3R)	determine or temporarily allocate IP address
klm_prot	**klm_prot**(3R)	protocol between kernel and local lock manager
mount	**mount**(3R)	keep track of remotely mounted filesystems
nlm_prot	**nlm_prot**(3R)	protocol between local and remote network lock managers
passwd2des	**xcrypt**(3R)	hex encryption and utility routines
pnp	**pnp**(3R)	automatic network installation
publickey	**publickey**(3R)	get public or secret key
rex	**rex**(3R)	remote execution protocol
rnusers	**rnusers**(3R)	return information about users on remote machines
rquota	**rquota**(3R)	implement quotas on remote machines
rstat	**rstat**(3R)	get performance data from remote kernel
rusers	**rnusers**(3R)	return information about users on remote machines
rwall	**rwall**(3R)	write to specified remote machines
sm_inter	**sm_inter**(3R)	status monitor protocol
spray	**spray**(3R)	scatter data in order to check the network
xcrypt	**xcrypt**(3R)	hex encryption and utility routines
xdecrypt	**xcrypt**(3R)	hex encryption and utility routines
xencrypt	**xcrypt**(3R)	hex encryption and utility routines
yp	**yp**(3R)	NIS protocol
yppasswd	**yppasswd**(3R)	update user password in NIS

NAME

bootparam – bootparam protocol

PROTOCOL

/usr/include/rpcsvc/bootparam_prot.x

DESCRIPTION

The bootparam protocol is used for providing information to the diskless clients necessary for booting.

PROGRAMMING

#include <rpcsvc/bootparam.h>

XDR Routines

The following XDR routines are available in **librpcsvc**:

xdr_bp_whoami_arg
xdr_bp_whoami_res
xdr_bp_getfile_arg
xdr_bp_getfile_res

SEE ALSO

bootparams(5), **bootparamd**(8)

NAME

ether – monitor traffic on the Ethernet

PROTOCOL

/usr/include/rpcsvc/ether.x

DESCRIPTION

The ether protocol is used for monitoring traffic on the ethernet.

PROGRAMMING

#include <rpcsvc/ether.h>

The following XDR routines are available in **librpcsvc**:

xdr_etherstat
xdr_etheraddrs
xdr_etherhtable
xdr_etherhmem
xdr_addrmask

SEE ALSO

traffic(1C), **etherfind**(8C), **etherd**(8C)

NAME

getrpcport – get RPC port number

SYNOPSIS

```
int getrpcport(host, prognum, versnum, proto)
char *host;
int prognum, versnum, proto;
```

DESCRIPTION

getrpcport() returns the port number for version *versnum* of the RPC program *prognum* running on *host* and using protocol *proto*. It returns 0 if it cannot contact the portmapper, or if *prognum* is not registered. If *prognum* is registered but not with version *versnum*, it will still return a port number (for some version of the program) indicating that the program is indeed registered. The version mismatch will be detected upon the first call to the service.

NAME

ipalloc – determine or temporarily allocate IP address

PROTOCOL

/usr/include/rpcsvc/ipalloc.x

AVAILABILITY

Available only on Sun 386i systems running a SunOS 4.0*x* release or earlier. Not a SunOS 4.1 release feature.

DESCRIPTION

ipalloc( ) is the protocol for allocating the IP address that a system should use.

PROGRAMMING

#include <rpcsvc/ipalloc.h>

The following RPC calls are available in version 2 of this protocol:

NULLPROC

This is a standard null entry, used to ping a service to measure overhead or to discover servers.

IP_ALLOC

Returns an IP address corresponding to a given Ethernet address, if possible. This RPC must be called using DES authentication, from a client authorized to allocate IP addresses. A cache of allocated addresses is maintained.

The first action taken on receipt of this RPC is to verify that no existing mapping between the *etheraddr* and the *netnum* exists in the Network Information Service (NIS) database. If one is found, then that is returned. Otherwise, an internal cache is checked, and if an entry is found there for the given *etheraddr* on the right network, that entry is used. If no address was found either in the NIS database or in the cache, a new one may be allocated and returned, and the *ip_success* status is returned.

If an unusable entry was found in the cache, this RPC returns **ip_failure** status.

IP_TONAME

Used to determine whether a given IP address is known to the NIS service, since NIS allows a delay between the posting of an address and its availability in some locations on the network.

IP_FREE

This RPC is used to delete *ipaddr* entries from the cache when they are no longer needed there. It requires the same protections as the **IP_ALLOC** RPC.

SEE ALSO

ipallocd(8C), **pnpboot**(8C)

NOTES

The Network Information Service (NIS) was formerly known as Sun Yellow Pages (YP). The functionality of the two remains the same; only the name has changed.

NAME

klm_prot – protocol between kernel and local lock manager

PROTOCOL

/usr/include/klm_prot.x

DESCRIPTION

The protocol is used for communication between kernel and local lock manager.

PROGRAMMING

#include <rpcsvc/klm_prot.h>

XDR Routines

The following XDR routines are available in **librpcsvc:**

```
xdr_klm_testargs
xdr_klm_testrply
xdr_klm_lockargs
xdr_klm_unlockargs
xdr_klm_stat
```

SEE ALSO

lockd(8C)

NAME

mount – keep track of remotely mounted filesystems

PROTOCOL

/usr/include/rpcsvc/mount.x

DESCRIPTION

The mount protocol is separate from, but related to, the NFS protocol. It provides all of the operating system specific services to get the NFS off the ground — looking up path names, validating user identity, and checking access permissions. Clients use the mount protocol to get the first file handle, which allows them entry into a remote filesystem.

The mount protocol is kept separate from the NFS protocol to make it easy to plug in new access checking and validation methods without changing the NFS server protocol.

Note: the protocol definition implies stateful servers because the server maintains a list of client's mount requests. The mount list information is not critical for the correct functioning of either the client or the server. It is intended for advisory use only, for example, to warn people when a server is going down.

PROGRAMMING

#include <rpcsvc/mount.h>

The following XDR routines are available in **librpcsvc**:
xdr_exportbody
xdr_exports
xdr_fhandle
xdr_fhstatus
xdr_groups
xdr_mountbody
xdr_mountlist
xdr_path

SEE ALSO

mount(8), **mountd**(8C), **showmount**(8)

NFS Protocol Spec, in *Network Programming*

NAME

nlm_prot – protocol between local and remote network lock managers

PROTOCOL

/usr/include/rpcsvc/nlm_prot.x

DESCRIPTION

The network lock manager protocol is used for communication between local and remote lock managers.

PROGRAMMING

#include <rpcsvc/nlm_prot.h>

XDR Routines

The following XDR routines are available in **librpcsvc**:

```
xdr_nlm_testargs
xdr_nlm_testres
xdr_nlm_lockargs
xdr_nlm_cancargs
xdr_nlm_unlockargs
xdr_nlm_res
```

SEE ALSO

lockd(8C)

NAME

pnp – automatic network installation

PROTOCOL

/usr/include/rpcsvc/pnprpc.x

AVAILABILITY

Available only on Sun 386i systems running a SunOS 4.0*.x* release or earlier. Not a SunOS 4.1 release feature.

DESCRIPTION

pnp( ) is used during unattended network installation, and routine booting, of Sun386i systems on a Sun386i network. Each network cable (subnetwork or full network) must have at least one **pnpd**(8C) server running on it to support PNP.

PROGRAMMING

#include <rpcsvc/pnprpc.h>

The following RPC calls are available in version 2 of the PNP protocol:

NULLPROC

Finds a PNP daemon on the local network. Used with **clntudp_broadcast**( ), often to measure network overhead.

PNP_WHOAMI

Used early in the boot process to acquire network configuration information about a system, or to determine that a system is not known by the network.

PNP_ACQUIRE

Used to acquire a server willing to configure a new system after a **PNP_WHOAMI** request fails. This RPC is typically broadcast; any successful reply may be used.

PNP_SETUP

Requests a network configuration from a PNP daemon that has responded to a previous **PNP_ACQUIRE** RPC.

PNP_POLL

After a **PNP_SETUP** request, if the status is **in_progress**, the procedure is to wait 20 seconds, and issue a **PNP_POLL** request, and then check the status again. Once the status is **success**, the system will be configured for the network. Entries in the yp database may be added or old ones deleted, and file storage may be assigned, according to the architecture and boot type.

If the server misses 5 **PNP_POLL** requests, it will assume that the client system crashed and back out of the procedure. Similarly, if the client system does not receive responses from the server for **PNP_MISSEDPOLLS** consecutive requests, it should assume the server crashed and begin its PNP sequence again.

SEE ALSO

pnpboot(8C), **pnpd**(8C)

NAME

publickey, getpublickey, getsecretkey – get public or secret key

SYNOPSIS

```
#include <rpc/rpc.h>
#include <rpc/key_prot.h>

getpublickey(netname, publickey)
char netname[MAXNETNAMELEN+1];
char publickey[HEXKEYBYTES+1];

getsecretkey(netname, secretkey, passwd)
char netname[MAXNETNAMELEN+1];
char secretkey[HEXKEYBYTES+1];
char *passwd;
```

DESCRIPTION

These routines are used to get public and secret keys from the YP database. **getsecretkey( )** has an extra argument, *passwd*, which is used to decrypt the encrypted secret key stored in the database. Both routines return 1 if they are successful in finding the key, 0 otherwise. The keys are returned as NULL-terminated, hexadecimal strings. If the password supplied to **getsecretkey( )** fails to decrypt the secret key, the routine will return 1 but the *secretkey* argument will be a NULL string.

SEE ALSO

publickey(5)

RPC Programmer's Manual in *Network Programming*

NAME

rex – remote execution protocol

PROTOCOL

/usr/include/rpcsvc/rex.x

DESCRIPTION

This server will execute commands remotely. The working directory and environment of the command can be specified, and the standard input and output of the command can be arbitrarily redirected. An option is provided for interactive I/O for programs that expect to be running on terminals. Note: this service is only provided with the TCP transport.

PROGRAMMING

```
#include <sys/ioctl.h>
#include <rpcsvc/rex.h>  /* not compiled with rpgen */
```

The following XDR routines are available in **librpcsvc:**

```
xdr_rex_start( )
xdr_rex_result( )
xdr_rex_ttymode( )
xdr_rex_ttysize( )
```

SEE ALSO

on(1C), **rexd**(8C)

NAME

rnusers, rusers – return information about users on remote machines

PROTOCOL

/usr/include/rpcsvc/rnusers.x

DESCRIPTION

rnusers( ) returns the number of users logged on to *host* (–1 if it cannot determine that number). **rusers**( ) fills the **utmpidlearr** structure with data about *host*, and returns 0 if successful.

PROGRAMMING

```
#include <rpcsvc/rusers.h>
rnusers(host)
char *host
rusers(host, up)
char *host
struct utmpidlearr *up;
```

The following XDR routines are also available:

xdr_utmpidle
xdr_utmpidlearr

SEE ALSO

rusers(1C)

NAME

rquota – implement quotas on remote machines

PROTOCOL

/usr/include/rpcsvc/rquota.x

DESCRIPTION

The **rquota**( ) protocol inquires about quotas on remote machines. It is used in conjunction with NFS, since NFS itself does not implement quotas.

PROGRAMMING

#include <rpcsvc/rquota.h>

The following XDR routines are available in **librpcsvc**:
xdr_getquota_arg
xdr_getquota_rslt
xdr_rquota

SEE ALSO

quota(1), **quotactl**(2)

NAME

rstat – get performance data from remote kernel

PROTOCOL

/usr/include/rpcsvc/rstat.x

DESCRIPTION

The **rstat**( ) protocol is used to gather statistics from remote kernel. Statistics are available on items such as paging, swapping and cpu utilization.

PROGRAMMING

```
#include <rpcsvc/rstat.h>

havedisk(host)
char *host;

rstat(host, statp)
char *host;
struct statstime *statp;
```

havedisk( ) returns 1 if *host* has a disk, 0 if it does not, and –1 if this cannot be determined. **rstat**( ) fills in the **statstime** structure for *host*, and returns 0 if it was successful.

The following XDR routines are available in **librpcsvc**:
xdr_statstime
xdr_statsswtch
xdr_stats

SEE ALSO

perfmeter(1), **rup**(1C), **rstatd**(8C)

NAME

rwall – write to specified remote machines

SYNOPSIS

```
#include <rpcsvc/rwall.h>

rwall(host, msg);
        char *host, *msg;
```

DESCRIPTION

host prints the string *msg* to all its users. It returns 0 if successful.

RPC INFO

program number:

WALLPROG

procs:

WALLPROC_WALL

Takes string as argument (wrapstring), returns no arguments.
Executes *wall* on remote host with string.

versions:

RSTATVERS_ORIG

SEE ALSO

rwall(1C), **rwalld**(8C), **shutdown**(8)

NAME

sm_inter – status monitor protocol

PROTOCOL

/usr/include/rpcsvc/sm_inter.x

DESCRIPTION

The status monitor protocol is used for monitoring the status of remote hosts.

PROGRAMMING

#include <rpcsvc/sm_inter.h>

XDR Routines

The following XDR routines are available in **librpcsvc**:

xdr_sm_name
xdr_mon
xdr_mon_id
xdr_sm_stat_res
xdr_sm_stat

SEE ALSO

statd(8C)

NAME

spray – scatter data in order to check the network

PROTOCOL

/usr/include/rpcsvc/spray.x

DESCRIPTION

The spray protocol sends packets to a given machine to test the speed and reliability of it.

PROGRAMMING

#include <rpcsvc/spray.h>

The following XDR routines are available in **librpcsvc**:

xdr_sprayarr
xdr_spraycumul

SEE ALSO

spray(8C), **sprayd**(8C)

NAME

xcrypt, xencrypt, xdecrypt, passwd2des – hex encryption and utility routines

SYNOPSIS

```
xencrypt(data, key)
char *data;
char *key;

xdecrypt(data, key)
char *data;
char *key;

passwd2des(pass, key)
char *pass;
char *key;
```

DESCRIPTION

The routines **xencrypt** and **xdecrypt** take null-terminated hexadecimal strings as arguments, and encrypt them using the 8-byte *key* as input to the DES algorithm. The input strings must have a length that is a multiple on 16 hex digits (64 bits is the DES block size).

passwd2des converts a password, of arbitrary length, into an 8-byte DES key, with odd-parity set in the low bit of each byte. The high-order bit of each input byte is ignored.

These routines are used by the DES authentication subsystem for encrypting and decrypting the secret keys stored in the publickey database.

SEE ALSO

des_crypt(3), **publickey**(5)

NAME

yp – NIS protocol

PROTOCOL

/usr/include/rpcsvc/yp.x

DESCRIPTION

The Network Information Service (NIS) is used for the administration of network-wide databases. The service is composed mainly of two programs: **YPBINDPROG** for finding a NIS server and **YPPROG** for accessing the NIS databases.

PROGRAMMING

Refer to **ypclnt**(3N) for information on the programmatic interface to NIS servers and databases.

SEE ALSO

ypclnt(3N), **yppasswd**(3R)

NOTES

The Network Information Service (NIS) was formerly known as Sun Yellow Pages (YP). The functionality of the two remains the same; only the name has changed. The name Yellow Pages is a registered trademark in the United Kingdom of British Telecommunications plc, and may not be used without permission.

NAME

yppasswd – update user password in NIS

PROTOCOL

/usr/include/rpcsvc/yppasswd.x

DESCRIPTION

The **yppasswd()** protocol is used to change a user's password entry in the Network Information Service (NIS) password database.

If *oldpass* is indeed the old user password, this routine replaces the password entry with *newpw*. It returns 0 if successful.

PROGRAMMING

```
#include <rpcsvc/yppasswd.h>

yppasswd(oldpass, newpw)
        char *oldpass
        struct passwd *newpw;
```

SEE ALSO

yppasswd(1), **yppasswdd**(8C)

NOTES

The Network Information Service (NIS) was formerly known as Sun Yellow Pages (YP). The functionality of the two remains the same; only the name has changed. The name Yellow Pages is a registered trademark in the United Kingdom of British Telecommunications plc, and may not be used without permission.