Data General

Customer Documentation

Programmer's Reference for the DG/UX™ System (Volume 2)

AViiON®
PRODUCT LINE

Programmer's Reference for the DG/UX™ System (Volume 2)

093-701056-02

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 093-701056

Revision 02, June 1991

NOTICE

Programmer's Reference for the DG/UX System (Volume 2)

093-701056-02

Revision History:	Effective with:
Original Release – February 1990	DG/UX 4.20
Revision 1 – June 1990	DG/UX 4.30
Revision 2 – June 1991	DG/UX 5.4

Preface

This is Volume 2 of the *Programmer's Reference for the DG/UX™ System*. The *Programmer's Reference* describes the programming features of the DG/UX system. It contains individual manual pages that describe commands, system calls, subroutines, file formats, and other useful topics, such as the ASCII table shown on **ascii**(5).

This manual is part of a five-volume reference set. The other manuals are the *System Manager's Reference for the DG/UX System* and the *User's Reference for the DG/UX System*. These manuals contain in printed (typeset) form the online entries released with the DG/UX System in **/usr/catman** for access by the **man** command.

The *Programmer's Reference* provides neither a general overview of the DG/UX system nor details of the implementation of the system. For more details about some of the most often used programming tools, see *Programmer's Guide: ANSI C and Programming Support Tools*, *Programmer's Guide: System Services and Application Packaging Tools*, and the Data General supplements to these two manuals. Other related manuals are listed under "Related Manuals" at the end of this manual.

Man Pages

For historical reasons, each entry is called a "manual page" or "man page," though an entry may occupy more than one physical page and may contain more than one entry. If the man page contains more than one entry, it is alphabetized under its "primary" name; for example, the **abs** manual page describes the **abs** and **labs** files.

Manual pages are assigned to classes ranging from 0 through 8 for easy cross-reference. The class number appears in parentheses following the name; for example, in **accept**(1M) the "1" indicates that **accept** is a command, and the "M" indicates that the man page is in the *System Manager's Reference*.

A command followed by a (1) or (1G) usually means that it is described in the *User's Reference*. (Class 1 commands appropriate for use by programmers are located in the *Programmer's Reference*.) A man page name with a (1M), (4M), (7), or (8) following it means that the entry is in the *System Manager's Reference*. Names with (2) or (3*x*), (4), (5) [except **editread**(5)], or (6F) are in the *Programmer's Reference*. Occasionally, DG/UX man pages refer to other products' man pages, which are not part of the DG/UX documentation; these are so noted.

Manual Organization

Volume 1 contains two chapters:

Chapter 1: Commands (1)
This chapter describes commands that support C and other programming languages.

Chapter 2: System Calls (2) This chapter describes the access to services provided by the DG/UX kernel, including the C language interface and a description of returned error codes.

Volume 2 contains one chapter:

Chapter 3: Subroutines and Libraries (3) This chapter describes the available subroutines and subroutine libraries. Their binary versions reside in various system libraries in the directories **/lib** and **/usr/lib**. See **intro**(3) for descriptions of these libraries and the files in which they are stored. Although these man pages are alphabetized together, each has a letter associated with the number 3 indicating the pertinent library:

3C	C Programming Language Libraries
3E	ELF Library Routines
3G	General Library Routines
3M	Mathematical Library Routines
3N	Networking Support Utilities
3R	Remote Procedure Call Routines
3S	Standard I/O Library Routines
3W	Multinational Language Set (MNLS) Routines
3X	Specialized Libraries

Volume 3 contains three chapters and one appendix:

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

Chapter 5: Miscellaneous Features (5) This chapter contains a variety of facilities. Included are descriptions of character sets, macro packages, and other things.

Chapter 6: Communications Protocols (6) This chapter contains a description of the **unix_ipc** communications facility.

Appendix A: Contents and Permuted Index Man Pages
These manual pages contain information extracted from the DG/UX man pages in all five reference volumes.

Man Page Format

Each man page has at least some of the following sections:

NAME gives the primary name (and secondary names, as the case may be) and briefly states its purpose.

SYNOPSIS summarizes the usage of the program being described.

DESCRIPTION discusses how to use these commands.

EXAMPLES gives examples of usage, where appropriate.

FILES contains the file names that are referenced by the program.

EXIT CODES discusses values set when the command terminates. The value set is available in the shell environment variable "?" (see sh(1)).

DIAGNOSTICS discusses the error messages that may be produced. Messages that are intended to be self-explanatory are not listed.

SEE ALSO offers pointers to related information.

NOTES gives information that may be helpful under the particular circumstances described.

Some man pages may contain other heads such as **ENVIRONMENT** and **CAVEATS**.

Man Page Notation Conventions

This manual uses certain symbols and styles of type to indicate different meanings in man pages. Those symbol and typeface conventions are defined in the following list. You should familiarize yourself with these conventions before reading the manual.

The description of convention meanings uses the terms "command line," "format line," and "syntax line." A command line is an example of a command string that you should type verbatim; it is preceded by a system prompt. A format line shows how to structure a command; it shows the variables that must be supplied and the available options. A syntax line is a fragment of program code that shows how to use a particular routine; some syntax lines contain variables.

Convention	Meaning
boldface	This font is used for section heads and subsection heads. It is also used to distinguish input from output in examples where the two are intermixed.
`constant width/ monospace`	In command formats and code syntax: This typeface indicates text (including punctuation) that you type verbatim from your keyboard. In text: This typeface is used for examples, code samples, pathnames, and the names of commands, files, directories, and manual pages. In all contexts: The following characters, which have special meanings explained below, do not have special meaning but simply represent themselves when they appear in constant-width font: `< > [ ] { } |`. In constant-width font they are are I/O redirection operators, brackets, braces, and the pipe symbol.
italic	In format lines: This font represents variables for which you supply values; for example, the names of your directories and files, your username and password, and possible arguments to commands.
[*optional*]	In format lines: Regular-font brackets surround an optional argument. Don't type the brackets; they only set off what is optional. These brackets should not be confused with constant-width brackets.
choice1|*choice2*	In format lines: The vertical bar indicates a choice between *choice1* and *choice2*.
...	In format lines and syntax lines: You can repeat the preceding argument as many times as desired.
{ }	In format lines: These regular-font braces surround either two or more choices or syntax elements that are repeatable as a group.
< >	In command lines and other examples: Angle brackets distinguish a command sequence or a keystroke (such as <**Ctrl-D**>, <**Esc**>, and <**3dw**>) from surrounding text. Note that these angle brackets are in regular type and that you do not type them; there are, however, constant-width versions of these symbols that you do type.
`$, %, #`	In command lines and other examples: These symbols represent the system command prompt symbols used for the Bourne and Korn shells, the C shell, and the superuser, respectively. Note that your system might use different symbols for the command prompts.

Contacting Data General

Data General wants to assist you in any way it can to help you use its products. Please feel free to contact the company as outlined below.

Manuals

If you require additional manuals, please use the enclosed TIPS order form (United States only) or contact your local Data General sales representative. A list of related documents appears at the end of this manual with the TIPS order form.

For a complete list of AViiON® and DG/UX™ manuals, see the *Guide to AViiON® and DG/UX™ System Documentation* (069-701085). The on-line version of this manual found in **/usr/release/doc_guide** contains the most current list.

Telephone Assistance

If you are unable to solve a problem using any manual you received with your system, free telephone assistance is available with your hardware warranty and with most Data General software service options. If you are within the United States or Canada, contact the Data General Service Center by calling 1-800-DG-HELPS. Lines are open from 8:00 a.m. to 5:00 p.m., your time, Monday through Friday. The center will put you in touch with a member of Data General's telephone assistance staff who can answer your questions.

For telephone assistance outside the United States or Canada, ask your Data General sales representative for the appropriate telephone number.

Joining Our Users Group

Please consider joining the largest independent organization ot Data General users, the North American Data General Users Group (NADGUG). In addition to making valuable contacts, members receive FOCUS monthly magazine, a conference discount, access to the Software Library and Electronic Bulletin Board, an annual Member Directory, Regional and Special Interest Groups, and much more. For more information about membership in the North American Data General Users Group, call 1-800-877-4787 or 1-512-345-5316.

End of Preface

Contents

Chapter 3 — Subroutines and Libraries

intro(3) 3-2
intro(3N) 3-8
a641(3C) 3-9
abort(3C) 3-10
abs(3C) 3-11
addseverity(3C) 3-12
assert(3X) 3-13
atexit(3C) 3-14
basename(3G) 3-15
bcmp(3C) 3-16
bcopy(3C) 3-17
berk_regex(3C) 3-18
berk_signal(3C) 3-19
bessel(3M) 3-21
bgets(3G) 3-22
bsearch(3C) 3-23
bufsplit(3G) 3-25
byteorder(3N) 3-26
bzero(3C) 3-27
catgets(3C) 3-28
catopen(3C) 3-29
cfsetospeed(3C) 3-31
clock(3C) 3-33
conv(3C) 3-34
copylist(3G) 3-35
crypt(3C) 3-36
crypt(3X) 3-37
ctermid(3S) 3-39
ctime(3C) 3-40
ctype(3C) 3-43
curs_addch(3X) 3-45
curs_addchst(3X) 3-47
curs_addchstr(3X) 3-48
curs_addstr(3X) 3-49
curs_addwch(3X) 3-50
curs_addwchstr(3X) 3-51
curs_attr(3X) 3-52
curs_beep(3X) 3-54
curs_bkgd(3X) 3-55
curs_border(3X) 3-56
curs_clear(3X) 3-57
curs_color(3X) 3-58
curs_delch(3X) 3-60
curs_deleteln(3X) 3-61

curs_getch(3X) 3-62
curs_getstr(3X) 3-66
curs_getwch(3X) 3-67
curs_getwstr(3X) 3-71
curs_getyx(3X) 3-72
curs_inch(3X) 3-73
curs_inchstr(3X) 3-74
curs_initscr(3X) 3-75
curs_inopts(3X) 3-77
curs_insch(3X) 3-80
curs_insstr(3X) 3-81
curs_instr(3X) 3-82
curs_inswch(3X) 3-83
curs_inswstr(3X) 3-84
curs_inwch(3X) 3-85
curs_inwchstr(3X) 3-86
curs_inwstr(3X) 3-87
curs_kernel(3X) 3-88
curs_move(3X) 3-90
curs_outopts(3X) 3-91
curs_overlay(3X) 3-93
curs_pad(3X) 3-94
curs_printw(3X) 3-96
curs_refresh(3X) 3-97
curs_scanw(3X) 3-99
curs_scr_dump(3X) 3-100
curs_scroll(3X) 3-101
curs_slk(3X) 3-102
curs_termattrs(3X) 3-104
curs_termcap(3X) 3-106
curs_terminfo(3X) 3-107
curs_touch(3X) 3-110
curs_util(3X) 3-111
curs_window(3X) 3-113
curses(3X) 3-115
cuserid(3S) 3-125
dbm(3X) 3-126
dg_flock(3C) 3-128
dg_seek(3C) 3-130
dial(3C) 3-131
directory(3X) 3-133
dirname(3G) 3-135
div(3C) 3-136
doconfig(3N) 3-137
drand48(3C) 3-139
drem(3M) 3-141
ecvt(3C) 3-142
elf(3E) 3-143
elf_begin(3E) 3-147
elf_cntl(3E) 3-152
elf_end(3E) 3-153
elf_error(3E) 3-154
elf_fill(3E) 3-155

elf_flag(3E) 3-156
elf_fsize(3E) 3-158
elf_getarhdr(3E) 3-159
elf_getarsym(3E) 3-160
elf_getbase(3E) 3-161
elf_getdata(3E) 3-162
elf_getehdr(3E) 3-166
elf_getident(3E) 3-167
elf_getphdr(3E) 3-168
elf_getscn(3E) 3-169
elf_getshdr(3E) 3-171
elf_hash(3E) 3-172
elf_kind(3E) 3-173
elf_next(3E) 3-174
elf_rand(3E) 3-175
elf_rawfile(3E) 3-176
elf_strptr(3E) 3-177
elf_update(3E) 3-178
elf_version(3E) 3-181
elf_xlate(3E) 3-182
end(3C) 3-184
erf(3M) 3-185
ethers(3N) 3-186
exp(3M) 3-188
exportent(3C) 3-190
extended_perror(3C) 3-192
extended_strerror(3C) 3-194
fattach(3C) 3-195
fclose(3S) 3-196
fdetach(3C) 3-197
ferror(3S) 3-198
ffs(3C) 3-199
floor(3M) 3-200
fmtmsg(3C) 3-201
fopen(3S) 3-205
form_cursor(3X) 3-207
form_data(3X) 3-208
form_driver(3X) 3-209
form_field(3X) 3-211
form_field_attributes(3X) 3-212
form_field_buffer(3X) 3-213
form_field_info(3X) 3-214
form_field_just(3X) 3-215
form_field_new(3X) 3-216
form_field_opts(3X) 3-217
form_field_userptr(3X) 3-218
form_field_validation(3X) 3-219
form_fieldtype(3X) 3-220
form_hook(3X) 3-222
form_new(3X) 3-223
form_new_page(3X) 3-224
form_opts(3X) 3-225
form_page(3X) 3-226

form_post(3X) 3-227
form_userptr(3X) 3-228
form_win(3X) 3-229
forms(3X) 3-230
fpgetround(3C) 3-233
fread(3S) 3-235
frexp(3C) 3-236
fseek(3S) 3-238
ftime(3C) 3-239
ftw(3C) 3-240
gamma(3M) 3-242
getc(3S) 3-243
getcwd(3C) 3-244
getdate(3C) 3-245
getenv(3C) 3-248
getfsent(3C) 3-249
getgrent(3C) 3-250
gethostent(3N) 3-252
getlogin(3C) 3-254
getmntent(3C) 3-255
getnetconfig(3N) 3-256
getnetent(3N) 3-258
getnetgrent(3N) 3-260
getnetpath(3N) 3-261
getopt(3C) 3-262
getpass(3C) 3-264
getprotoent(3N) 3-265
getpw(3C) 3-266
getpwent(3C) 3-267
getrpcent(3N) 3-269
getrpcport(3R) 3-270
gets(3S) 3-271
getservent(3N) 3-272
getspent(3C) 3-274
getsubopt(3C) 3-276
gettxt(3C) 3-279
getut(3C) 3-281
getwc(3W) 3-283
getwd(3C) 3-284
getwidth(3W) 3-285
getws(3W) 3-286
gmatch(3G) 3-287
grantpt(3C) 3-288
hsearch(3C) 3-289
hypot(3M) 3-291
ieeefp(3C) 3-292
index(3C) 3-293
inet(3N) 3-295
initgroups(3C) 3-297
insque(3C) 3-298
isalphanum(3C) 3-299
isastream(3C) 3-300
isencrypt(3G) 3-301

ishex(3C) 3-302
isnan(3C) 3-303
itoa(3C) 3-304
jobs(3C) 3-305
l3tol(3C) 3-310
ldahread(3X) 3-311
ldclose(3X) 3-312
ldfhread(3X) 3-313
ldgetname(3X) 3-314
ldlread(3X) 3-315
ldlseek(3X) 3-316
ldohseek(3X) 3-317
ldopen(3X) 3-318
ldrseek(3X) 3-320
ldshread(3X) 3-321
ldsseek(3X) 3-322
ldtbindex(3X) 3-323
ldtbread(3X) 3-324
ldtbseek(3X) 3-325
localeconv(3C) 3-326
lockf(3C) 3-329
logname(3X) 3-331
lsearch(3C) 3-332
main(3C) 3-334
malloc(3C) 3-336
malloc(3X) 3-338
matherr(3C) 3-340
mbchar(3C) 3-343
mbchar(3W) 3-345
mbstring(3C) 3-346
mbstring(3W) 3-347
memory(3C) 3-348
menu_attributes(3X) 3-349
menu_cursor(3X) 3-350
menu_driver(3X) 3-351
menu_format(3X) 3-353
menu_hook(3X) 3-354
menu_item_current(3X) 3-355
menu_item_name(3X) 3-356
menu_item_new(3X) 3-357
menu_item_opts(3X) 3-358
menu_item_userptr(3X) 3-359
menu_item_value(3X) 3-360
menu_item_visible(3X) 3-361
menu_items(3X) 3-362
menu_mark(3X) 3-363
menu_new(3X) 3-364
menu_opts(3X) 3-365
menu_pattern(3X) 3-366
menu_post(3X) 3-367
menu_userptr(3X) 3-368
menu_win(3X) 3-369
menus(3X) 3-370

mkdirp(3G) 3-373
mkfifo(3C) 3-374
mkstemp(3C) 3-375
mktemp(3C) 3-376
mlock(3C) 3-377
mlockall(3C) 3-378
monitor(3C) 3-379
mp(3X) 3-381
msync(3C) 3-383
ndbm(3C) 3-384
netdir(3N) 3-386
nl_langinfo(3C) 3-390
nlist(3C) 3-391
nlsgetcall(3N) 3-392
nlsprovider(3N) 3-393
nlsrequest(3N) 3-394
p2open(3G) 3-395
panel_above(3X) 3-397
panel_move(3X) 3-398
panel_new(3X) 3-399
panel_show(3X) 3-400
panel_top(3X) 3-401
panel_update(3X) 3-402
panel_userptr(3X) 3-403
panel_window(3X) 3-404
panels(3X) 3-405
pathfind(3G) 3-406
perror(3C) 3-407
popen(3S) 3-408
printf(3S) 3-410
printf(3W) 3-415
psignal(3C) 3-416
ptsname(3C) 3-417
putc(3S) 3-418
putenv(3C) 3-419
putpwent(3C) 3-420
puts(3S) 3-421
putspent(3C) 3-422
putwc(3W) 3-423
putws(3W) 3-424
qsort(3C) 3-425
raise(3C) 3-426
rand(3C) 3-427
random(3C) 3-428
rcmd(3X) 3-430
realpath(3C) 3-431
regcmp(3G) 3-432
regcmp(3X) 3-434
regexpr(3G) 3-437
remove(3C) 3-440
remque(3C) 3-441
resolver(3C) 3-443
rexec(3X) 3-445

rindex(3C) 3-446
rpc(3N) 3-448
rtime(3N) 3-464
scandir(3C) 3-465
scanf(3S) 3-466
scanf(3W) 3-470
setbuf(3S) 3-472
setbuffer(3C) 3-473
setjmp(3C) 3-474
setlinebuf(3C) 3-476
setlocale(3C) 3-477
sigsetjmp(3C) 3-479
sigsetops(3C) 3-480
sinh(3M) 3-481
sleep(3C) 3-482
sputl(3X) 3-483
ssignal(3C) 3-484
stdio(3S) 3-485
stdipc(3C) 3-487
str(3G) 3-488
strccpy(3G) 3-489
strcoll(3C) 3-490
strerror(3C) 3-491
strftime(3C) 3-492
string(3C) 3-494
strsave(3C) 3-496
strtod(3C) 3-498
strtol(3C) 3-499
strxfrm(3C) 3-500
swab(3C) 3-501
swapcontext(3C) 3-502
syslog(3C) 3-503
system(3S) 3-505
sysv3_cuserid(3S) 3-506
t_accept(3N) 3-507
t_alloc(3N) 3-509
t_bind(3N) 3-511
t_close(3N) 3-513
t_connect(3N) 3-514
t_error(3N) 3-516
t_free(3N) 3-517
t_getinfo(3N) 3-518
t_getstate(3N) 3-520
t_listen(3N) 3-521
t_look(3N) 3-523
t_open(3N) 3-524
t_optmgmt(3N) 3-526
t_rcv(3N) 3-528
t_rcvconnect(3N) 3-529
t_rcvdis(3N) 3-531
t_rcvrel(3N) 3-533
t_rcvudata(3N) 3-534
t_rcvuderr(3N) 3-536

t_snd(3N) 3-537
t_snddis(3N) 3-539
t_sndrel(3N) 3-541
t_sndudata(3N) 3-542
t_sync(3N) 3-544
t_unbind(3N) 3-546
tcflush(3C) 3-547
tcgetpgrp(3C) 3-550
tcsetattr(3C) 3-551
tcsetpgrp(3C) 3-553
termcap(3X) 3-554
termios(3C) 3-556
tmpfile(3S) 3-560
tmpnam(3S) 3-561
trig(3M) 3-562
tsearch(3C) 3-563
ttyname(3C) 3-567
ttyslot(3C) 3-568
ungetc(3S) 3-569
ungetwc(3W) 3-570
unlockpt(3C) 3-571
vlimit(3C) 3-572
vprintf(3S) 3-573
vprintf(3W) 3-574
vscanf(3S) 3-575
vtimes(3C) 3-576
wconv(3W) 3-577
wctype(3W) 3-578
widec(3W) 3-579
wstring(3W) 3-581
xdr(3N) 3-583
ypclnt(3N) 3-590

Index

Related Documents

Chapter 3
Subroutines and Libraries

This chapter contains in printed form all the online manual entries for DG/UX subroutines and libraries. The entries are in alphabetical order except for **intro**(3) and **intro**(3N), which appear at the beginning. Subroutines are listed by library in **intro**(3).

NAME

intro – introduction to subroutines and libraries

DESCRIPTION

This section describes functions found in various libraries supplied with the DG/UX System. Declarations for some of these functions may be obtained from `#include` files indicated on the appropriate pages. The man pages are grouped by using various suffixes after the number 3. They are sorted together in the manual.

(3C) These functions make up the "general" C functions, many of which are part of the ANSI C. The others have traditionally been part of libc on AT&T Unix based systems.

(3E) These functions constitute the ELF library.

(3G) These are general C functions not part of `libc`.

(3M) These functions constitute the math library, `libm`. They are not automatically loaded by the C compiler, `cc`(1); however, the link editor searches this library under the `-lm` option. Declarations for these functions may be obtained from the `#include` file `<math.h>`. Several generally useful mathematical constants are also defined there (see `math`(5)).

(3N) These functions constitute the Internet network library, and Network Services library.

(3R) RPC (Remote Procedure Call) related functions.

(3S) These functions constitute the standard I/O package (see `stdio`(3S)). Declarations for them may be obtained from the `#include` file `<stdio.h>`.

(3W) Multinational Language Set (MNLS) functions.

(3X) Various specialized libraries. The library names in which these functions are found are given on the appropriate pages.

The DG/UX System provides a number of Software Development Environments, or SDEs. A SDE represents a target binary or object interface. It is made up of a set of libraries (object and binary interfaces) and header files (constants, data structures, types) that may be used together to create applications that adhere to a particular binary interface standard.

See `sde`(5) and `sde-target`(1) for how to select SDEs. The function groupings described above may not be available in all the SDEs. Some functions even appear in different libraries from one SDE to another.

The DG/UX System supports 2 object formats, commonly known as COFF and ELF. The COFF format is associated with pre-V.4 Unix systems from AT&T. The ELF format originates from V.4 Unix systems. In general, the SDEs deal with one specific object file format. However, the SDEs that handle ELF format can also handle COFF format by converting objects from COFF to ELF.

The following SDEs are available.

m88kdguxelf (and m88kdgux)

This is the System V Release 4 ELF environment. The default environment, m88kdgux, points to m88kdguxelf. This is the environment that contains ELF shared libraries. There are also several ELF static libraries. Also, in order to avoid duplication, several libraries in this environment are COFF libraries and are shared with the COFF environments.

m88kdguxcoff

This is the 4.3x based System V Release 3 COFF development environment. it is provided for development of applications that will run on DG/UX 4.3x systems. This environment corresponds to m88kdgux on 4.3x systems.

m88kocs

This COFF environment provides the 88open OCS standard interface. It should be used for developing applications that include object files that will be linked on the application target machine (i.e. libraries). OCS certification has not taken place at this time.

m88kbcs

The m88kbcs environment is essentially unchanged from 4.3x. It is intended for developing applications consisting entirely of statically linked COFF executables that adhere to the 88open BCS. It provides a System V Interface Definition, Edition 2 compatible **libc** with BSD, POSIX, and ANSI extensions.

m88kdgux

This environment is a pointer to one of the other environments. It provides the default if no other environment is explicitly requested. In 4.3x, this was a pointer to what is now m88kdguxcoff. In DG/UX Release 5.4 this is a pointer to m88kdguxelf.

All of the above environments are intended to provide the set of interfaces specified by the OCS standard. In the ELF environments, these interfaces may not appear in the same libraries specified by the OCS. This is due to restructuring by AT&T in System V Release 4. See below for more information.

A few functions are located in different libraries in the ELF environments than in the COFF environments. The **regex** and **regcmp** functions are available in **libPW** in the COFF environments, and in **libgen** in the ELF environments. The **nlist** function is available in **libc** in the COFF environments, and in **libelf** in the ELF environments.

In addition, many functions normally found in **libc** in the m88kdguxcoff environment are found in **libdgc** in the DG/UX Release 5.4 ELF environment. The reason for this split is to facilitate ABI compliance in the future. The general classes that have been moved are the RPC/YP (3N) functions, socket/internet functions (3N), domain name service functions (3N), Berkeley 4.2 and 4.3 extensions, and miscellaneous DG extensions.

This is the list of symbols in **libdgc** in the ELF environment. These symbols are located in **libc** in the COFF environments.

accept
addexportent
addmntent
alphasort
authdes_create
authdes_getucred
authnone_create
authunix_create
authunix_create_default
bcmp
bcopy
berk_signal
berk_sigpause
bind
bindresvport
bzero
callrpc
cbc_crypt
clnt_broadcast
clnt_create
clnt_pcreateerror
clnt_perrno
clnt_perror
clnt_spcreateerror
clnt_sperrno
clnt_sperror
clntraw_create
clnttcp_create

clntudp_create
connect
dbm_clearerr
dbm_close
dbm_delete
dbm_error
dbm_fetch
dbm_firstkey
dbm_nextkey
dbm_open
dbm_store
des_setparity
dg_allow_shared_descriptor_attach
dg_attach_to_shared_descriptors
dg_block_seek
dg_decryptsessionkey
dg_encryptsessionkey
dg_ext_errno
dg_file_info
dg_flock
dg_fstat
dg_getrootkey
dg_ipc_info
dg_lcntl
dg_lock_kill
dg_lock_reset
dg_lock_wait
dg_mknod
dg_mount
dg_mstat
dg_paging_info
dg_process_info
dg_seek
dg_set_cpd_limits
dg_setsecretkey
dg_stat
dg_sys_info
dg_sysctl
dg_unbuffered_read
dg_unbuffered_write
dg_xtrace
dn_comp
dn_expand
ecb_crypt
endexportent
endfsent
endhostent
endmntent
endnetent
endnetgrent
endprotoent
endrpcent
endservent
ether_aton
ether_hostton
ether_line
ether_ntoa
ether_ntohost
extended_perror
ftime
get_myaddress
getdomainname
getdtablesize
getexportent
getexportopt
getfh
getfsent
getfsfile
getfsspec
getfstype
gethostbyaddr
gethostbyname
gethostent
gethostid
gethostname
getmntent
getnetbyaddr
getnetbyname
getnetent
getnetgrent
getnetname
getpagesize
getpeername
getpgrp2
getpriority
getprotobyname
getprotobynumber
getprotoent
getpsr
getrpcbyname
getrpcbynumber
getrpcent
getrpcport
getrusage
getservbyname
getservbyport
getservent
getsockname

- getsockopt
- getwd
- h_errlist
- h_errno
- hasmntopt
- herror
- host2netname
- index
- inet_addr
- inet_lnaof
- inet_makeaddr
- inet_netof
- inet_network
- inet_ntoa
- initstate
- innetgr
- isalphanum
- ishex
- itoa
- key_decryptsession
- key_encryptsession
- key_gendes
- key_setsecret
- killpg
- listen
- memctl
- mkstemp
- msgsys
- netname2host
- netname2user
- p_type
- pmap_getmaps
- pmap_getport
- pmap_rmtcall
- pmap_set
- pmap_unset
- random
- rcmd
- re_comp
- re_exec
- reboot
- recv
- recvfrom
- recvmsg
- registerrpc
- remexportent
- res_init
- res_mkquery
- res_send
- rexec
- rindex
- rpc_createerr
- rresvport
- rtime
- ruserok
- scandir
- select
- semsys
- send
- sendmsg
- sendto
- setdomainname
- setegid
- seteuid
- setexportent
- setfsent
- sethostent
- sethostid
- sethostname
- setmntent
- setnetent
- setnetgrent
- setpgrp2
- setpriority
- setprotoent
- setpsr
- setregid
- setreuid
- setrpcent
- setservent
- setsockopt
- setstate
- shmsys
- shutdown
- sigblock
- sigret
- sigsetmask
- sigstack
- socket
- socketpair
- srandom
- strnsave
- strsave
- svc_fdset
- svc_getreq
- svc_getreqset
- svc_register
- svc_run

svc_sendreply
svc_unregister
svcerr_auth
svcerr_decode
svcerr_noproc
svcerr_noprog
svcerr_progvers
svcerr_systemerr
svcerr_weakauth
svcfd_create
svcraw_create
svctcp_create
svcudp_create
swapon
user2netname
utimes
vhangup
vlimit
vtimes
wait3
wait4
xdr_accepted_reply
xdr_array
xdr_authunix_parms
xdr_bool
xdr_bytes
xdr_callhdr
xdr_callmsg
xdr_char
xdr_double
xdr_enum
xdr_float
xdr_free
xdr_int
xdr_long
xdr_opaque
xdr_opaque_auth
xdr_pmap
xdr_pmaplist
xdr_pointer
xdr_reference
xdr_rejected_reply
xdr_replymsg
xdr_short
xdr_string
xdr_u_char
xdr_u_int
xdr_u_long
xdr_u_short
xdr_union
xdr_vector
xdr_void
xdr_wrapstring
xdrmem_create
xdrrec_create
xdrrec_endofrecord
xdrrec_eof
xdrrec_skiprecord
xdrstdio_create
xprt_register
xprt_unregister
yp_all
yp_bind
yp_first
yp_get_default_domain
yp_gethostbyname
yp_master
yp_match
yp_next
yp_order
yp_unbind
yp_update
yperr_string
ypprot_err

Many of the internationalization features, such as message catalog facilities (see `gettxt`(3C)) that are available in the m88kdguxelf environment are absent from the m88kdguxcoff, m88kbcs, and m88kocs environments. Many other internationalization features, such as `strcoll`(3C) are present, but offer only C locale behavior.

Definitions

character Any bit pattern able to fit into a byte on the machine.

null character A character with value 0, represented in the C language as '\0'.

character array A sequence of characters.

null-terminated character array
A sequence of characters, the last of which is the *null character*.

string is a designation for a *null-terminated character array*.

null string A character array containing only the null character.

NULL pointer The value that is obtained by casting 0 into a pointer. The C language guarantees that this value will not match that of any legitimate pointer, so many functions that return pointers return it to indicate an error.

NULL **Defined as** 0 in <stdio.h>; the user may include an appropriate definition if not using <stdio.h>.

FILES

/usr/lib/libc.a	Standard C library
/usr/lib/libcrypt.a	Alternate encryption library
/usr/lib/libdbm.a	Database access library
/lib/libcurses.a	Terminal screen handling library
/usr/lib/libm.a	Math library
/usr/lib/libmalloc.a	Alternate memory management library
/usr/lib/libmp.a	Multi-precision integer math library
/usr/lib/libnsl_s.a	Network Services library
/lib/libtermcap.a	termcap access library
/usr/lib/libPW.a	Programmer Workbench library
/usr/sde/*/lib/*	Any SDE specific components linked via elinks
/usr/lib/libdl.a	Dynamic linking interfaces (ELF)
/usr/lib/libdgc.a	Non-AT&T based portion of libc (ELF)
/usr/lib/libelf.a	Object format interfaces (COFF)
/usr/lib/libgen.a	General library functions (COFF)
/usr/lib/libmail.a	Mail file interfaces (ELF)

DIAGNOSTICS

Functions in the C and math Libraries (3C and 3M) may return the conventional values 0 or ±HUGE when the function is undefined for the given arguments or when the value is not representable. These are the largest-magnitude single-precision floating-point numbers; HUGE is defined in the <math.h> header file. In these cases, the external variable errno (see intro(2)) is set to the value EDOM or ERANGE.

SEE ALSO

ar(1), cc(1), f77(1), ld(1), lint(1), nm(1), sde-target(1), intro(2), intro(3N), stdio(3S), math(5).

CAUTION

Many of the functions in the libraries call and/or refer to other functions and external variables described in this section and in Chapter 2 (*System Calls*). If a program inadvertently defines a function or external variable with the same name, the presumed library version of the function or external variable may not be loaded. The lint(1) program checker reports name conflicts of this kind as multiple declarations of the names in question. Definitions for sections 2, 3C, 3N, 3R and 3S are checked automatically. Other definitions can be included by using the -l option (for example, -lm includes definitions for the math library, section 3M). Use of lint is highly recommended.

NAME

intro – introduction to network library functions

DESCRIPTION

This page lists functions that apply to the DARPA Internet network.

List of Functions

Name	*Appears on Page*	*Description*
gethostbyaddr	gethostent(3N)	get network host entry
gethostbyname	gethostent(3N)	get network host entry
gethostent	gethostent(3N)	get network host entry
getnetbyaddr	getnetent(3N)	get network entry
getnetbyname	getnetent(3N)	get network entry
getnetent	getnetent(3N)	get network entry
getprotobyname	getprotoent(3N)	get protocol entry
getprotobynumber	getprotoent.3n	get protocol entry
getprotoent	getprotoent(3N)	get protocol entry
getservbyname	getservent(3N)	get service entry
getservbyport	getservent(3N)	get service entry
getservent	getservent(3N)	get service entry
htonl	byteorder(3N)	convert values between host and network byte order
htons	byteorder(3N)	convert values between host and network byte order
ntohl	byteorder(3N)	convert values between host and network byte order
ntohs	byteorder(3N)	convert values between host and network byte order

SEE ALSO

intro(3).

NAME

`a64l`, `l64a` – convert between long integer and base-64 ASCII string

SYNOPSIS

```
#include <stdlib.h>

long a64l (const char *s);

char *l64a (long l);
```

DESCRIPTION

These functions are used to maintain numbers stored in base-64 ASCII characters. These characters define a notation by which long integers can be represented by up to six characters; each character represents a digit in a radix-64 notation. Like normal numerals, the digits are arranged from right to left.

The characters used to represent "digits" are:

`.`	for 0
`/`	for 1
`0-9`	for 2–11
`A-Z`	for 12–37
`a-z`	for 38–63

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

`a64l` scans the character string from left to right with the least significant digit on the left, decoding each character as a 6-bit radix-64 number.

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

SEE ALSO

`l3tol`(3C).

CAVEATS

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

NAME

abort – generate an abnormal termination signal

SYNOPSIS

```
#include <stdlib.h>

void abort (void);
```

DESCRIPTION

abort first closes all open files, stdio(3S) streams, directory streams and message catalogue descriptors, if possible, then causes the signal SIGABRT to be sent to the calling process.

DIAGNOSTICS

If SIGABRT is neither caught nor ignored, and the current directory is writable, a core dump is produced and the message abort - core dumped is written by the shell [see sh(1)].

SEE ALSO

sdb(1), exit(2), kill(2), signal(2), catopen(3C), stdio(3S).
sh(1) in the *User's Reference Manual.*

NAME

abs, labs – return integer absolute value

SYNOPSIS

```
#include <stdlib.h>

int abs (int val);

long labs (long lval);
```

DESCRIPTION

abs returns the absolute value of its int operand. labs returns the absolute value of its long operand.

SEE ALSO

floor(3M).

NOTES

In 2's-complement representation, the absolute value of the largest magnitude negative integral value is undefined.

NAME

addseverity – build list of severity levels for application to be used with fmtmsg

SYNOPSIS

```
#include <fmtmsg.h>

int addseverity(int severity, const char *string);
```

DESCRIPTION

The addseverity function builds a list of severity levels for an application to be used with the message formatting facility, fmtmsg. *severity* is an integer value indicating the seriousness of the condition, and *string* is a pointer to a string describing the condition (string is not limited to a specific size).

If addseverity is called with an integer value that has not been previously defined, the function adds that new severity value and print string to the existing set of standard severity levels.

If addseverity is called with an integer value that has been previously defined, the function redefines that value with the new print string. Previously defined severity levels may be removed by supplying the NULL string. If addseverity is called with a negative number or an integer value of 0, 1, 2, 3, or 4, the function fails and returns –1. The values 0–4 are reserved for the standard severity levels and cannot be modified. Identifiers for the standard levels of severity are:

MM_HALT indicates that the application has encountered a severe fault and is halting. Produces the print string **HALT**.

MM_ERROR indicates that the application has detected a fault. Produces the print string **ERROR**.

MM_WARNING indicates a condition that is out of the ordinary, that might be a problem, and should be watched. Produces the print string **WARNING**.

MM_INFO provides information about a condition that is not in error. Produces the print string **INFO**.

MM_NOSEV indicates that no severity level is supplied for the message.

Severity levels may also be defined at run time using the SEV_LEVEL environment variable [see fmtmsg(3C)].

EXAMPLES

When the function addseverity is used as follows:

```
addseverity(7,"ALERT")
```

the following call to fmtmsg:

```
fmtmsg(MM_PRINT, "UX:cat", 7, "invalid syntax", "refer to
manual", "UX:cat:001")
```

produces:

```
UX:cat: ALERT: invalid syntax
TO FIX: refer to manual    UX:cat:001
```

DIAGNOSTICS

addseverity returns MM_OK on success or MM_NOTOK on failure.

SEE ALSO

fmtmsg(1M), fmtmsg(3C), gettxt(3C), printf(3S).

NAME

assert – verify program assertion

SYNOPSIS

```
#include <assert.h>

void assert (int expression);
```

DESCRIPTION

This macro is useful for putting diagnostics into programs. When it is executed, if *expression* is false (zero), assert prints

```
Assertion failed: expression, file xyz, line nnn
```

on the standard error output and aborts. In the error message, *xyz* is the name of the source file and *nnn* the source line number of the assert statement. The latter are respectively the values of the preprocessor macros __FILE__ and __LINE__.

Compiling with the preprocessor option -DNDEBUG [see cc(1) or cpp(1)], or with the preprocessor control statement

```
#define NDEBUG
```

ahead of the

```
#include <assert.h>
```

statement will stop assertions from being compiled into the program.

SEE ALSO

cc(1), cpp(1), abort(3C).

NOTES

Since assert is implemented as a macro, the *expression* may not contain any string literals.

NAME

atexit – add program termination routine

SYNOPSIS

```
#include <stdlib.h>

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

DESCRIPTION

atexit adds the function *func* to a list of functions to be called without arguments on normal termination of the program. Normal termination occurs by either a call to the exit system call or a return from main. At most 32 functions may be registered by atexit; the functions will be called in the reverse order of their registration.

atexit returns 0 if the registration succeeds, nonzero if it fails.

SEE ALSO

exit(2).

NAME

basename – return the last element of a path name

SYNOPSIS

`cc [`*flag* `...]` *file* `... -lgen [`*library* `...]`

```
#include <libgen.h>
```

`char *basename (char *`*path*`);`

DESCRIPTION

Given a pointer to a null-terminated character string that contains a path name, basename returns a pointer to the last element of *path*. Trailing "/" characters are deleted.

If *path* or **path* is zero, pointer to a static constant "." is returned.

EXAMPLES

Input string	Output pointer
/usr/lib	lib
/usr/	usr
/	/

SEE ALSO

dirname(3G).
basename(1) in the *User's Reference Manual.*

NAME

bcmp – compare two areas of memory

SYNOPSIS

```
int bcmp(), result, length;
char *ptr1, *ptr2;
...
result = bcmp(ptr1, ptr2, length);
```

where:

ptr1	A pointer to the first area.
ptr2	A pointer to the second area.
length	The number of bytes to compare.

DESCRIPTION

The bcmp function compares two areas of memory, byte by byte, as unsigned values. It compares the number of bytes you specify with *length*; this function comes from the University of California Berkeley UNIX (BSD) system. Unlike the memcmp function, the bcmp function tests only for equality. This function does not indicate whether the first area is greater or less than the second area.

RETURN VALUE

The bcmp function returns 0 if the areas are equal. Otherwise, it returns a nonzero value. However, if length is negative, the function bcmp returns 0.

EXAMPLE

The following program compares an input string to the username mike.

```
/* Program test for the bcmp() function */

#include <stdio.h>

int bcmp();
char buf[80];

main() {
    printf("Type in a name:  ");
    if (fgets(buf, sizeof(buf), stdin)) {
        if (!bcmp(buf, "mike\n", 5))
            printf("You typed mike!\n");
        else
            printf("You did not type mike.\n");
    }
    return 0;
}
```

SEE ALSO

memcmp(3C).

NAME

bcopy – copy bytes from one area to another

SYNOPSIS

```
int bcopy(), length;
char *source, *destination;
. . .
bcopy(source, destination, length);
```

where:

source	The start of the area to copy from
destination	The start of the area to copy the bytes to
length	The number of bytes to move

DESCRIPTION

The bcopy function copies the number of bytes that you specify from one memory location to another; this function comes from the University of California Berkeley UNIX (BSD) system. Unlike the memcpy function, bcopy behaves well if the two areas overlap. Also note that the source and destination arguments are reversed as compared to memcpy's calling sequence.

EXAMPLE

The following program copies a string into a buffer and then prints it.

```
/* Program test for the bcopy() function */

#include <stdio.h>

int bcopy();
char buffer[80];

main() {
    bcopy("Copy string with bcopy.\n", buffer, 24);
    fputs(buffer, stdout);
    return 0;
}
```

RETURN VALUE

The bcopy function returns no value.

SEE ALSO

memory(3C).

NAME

berk_regex, regex, re_comp, re_exec – handle regular expressions

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 0 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 nulls. The regular expressions recognized are described in the manual entry for ed (1), given the above difference.

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 \)

SEE ALSO

ed(1), ex(1).

NAME

berk_signal, signal – simplified software signal facilities

SYNOPSIS

```
#define _BSD_SIGNAL_FLAVOR
#include <signal.h>

(*signal(sig, func))()
         int (*func)();
```

DESCRIPTION

signal is a simplified interface to the more general sigvec(2) facility. If you define the _BSD_SIGNAL_FLAVOR macro or only the _BSD_SOURCE macro when you compile your C application, you will get the signal functionality described in this entry; otherwise, you will get the functionality described in signal(2). For more information on the _BSD_SIGNAL_FLAVOR and _BSD_SOURCE macros, see *Porting Applications to the DG/UX™ System.*

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 tty(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 cause an interrupt to a specified location. For a list of the signals, see <sys/signal.h>.

If *func* is SIG_DFL, the default action for signal *sig* is reinstated. If *func* is SIG_IGN, the signal is subsequently ignored and pending instances of the signal are discarded. Otherwise, when the signal occurs, further occurences 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, causing the call to terminate prematurely, the call is automatically restarted. In particular this can occur during a read(2) or write(2) on a slow device (such as a terminal; but not a file) and during a wait(2).

The value of *signal* is the previous (or initial) value of *func* for the particular signal.

After a fork(2) or vfork(2) the child inherits all signals. Execve(2) resets all caught signals to the default action; ignored signals remain ignored.

RETURN VALUE

The previous action is returned on a successful call. Otherwise, –1 is returned and errno is set to indicate the error.

DIAGNOSTICS

signal will fail and no action will take place if one of the following occur:

EINVAL *Sig* is not a valid signal number.

EINVAL An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

EINVAL An attempt is made to ignore SIGCONT (by default SIGCONT is ignored).

SEE ALSO

kill(1), ptrace(2), kill(2), sigvec(2), sigblock(2), signal(2), sigsetmask(2), sigpause(2), sigstack(2), setjmp(3C), tty(4)

NOTES

signal(3C) provides compatibility with BSD signal handling while signal(2) is System V based. You can use signal(3C) in either of the following ways:

1) Define _BSD_SIGNAL_FLAVOR (or define _BSD_SOURCE while not defining _POSIX_SOURCE or _SYSV3_SOURCE), include <signal.h>, and call signal(). Calls to signal will translate to berk_signal, which is signal(3C). signal(2) is unavailable with this method.
2) Include <signal.h> and call berk_signal. With this method you can use both System V signal facilities (call signal to get signal(2)) and BSD signal facilities (call berk_signal to get signal(3C)).

STANDARDS

When using m88kbcs as the Software Development Environment (SDE) target, the berk_signal function will be an incomplete emulation of Berkeley semantics. Since we are using BCS system calls, system call restart is not available. Instead, interrupted system calls will fail with errno set to EINTR. Also, since this is an emulation requiring several BCS system calls, a slight performance degradation may be noticed in comparison to using berk_signal in /lib/libc.a.

NAME

bessel: `j0, j1, jn, y0, y1, yn` – Bessel functions

SYNOPSIS

`cc` [*flag* ...] *file* ... `-lm` [*library* ...]

```
#include <math.h>

double j0 (double x);

double j1 (double x);

double jn (int n, double x);

double y0 (double x);

double y1 (double x);

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

DESCRIPTION

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

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

DIAGNOSTICS

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

Arguments too large in magnitude cause `j0`, `j1`, `y0`, and `y1` to return 0 and to set `errno` to `ERANGE`. In addition, a message indicating `TLOSS` error is printed on the standard error output.

Except when the `-Xc` compilation option is used, these error-handling procedures may be changed with the function `matherr`. When the `-Xa` or `-Xc` compilation options are used, `HUGE_VAL` is returned instead of `HUGE` and no error messages are printed.

SEE ALSO

`matherr`(3M).

NAME

bgets – read stream up to next delimiter

SYNOPSIS

cc [*flag* ...] *file* ... -lgen [*library* ...]

```
#include <libgen.h>
```

char *bgets (char *buffer, size_t **count*, FILE **stream*,
        const char **breakstring*);

DESCRIPTION

bgets reads characters from *stream* into *buffer* until either *count* is exhausted or one of the characters in *breakstring* is encountered in the stream. The read data is terminated with a null byte ('\0') and a pointer to the trailing null is returned. If a *breakstring* character is encountered, the last non-null is the delimiter character that terminated the scan.

Note that, except for the fact that the returned value points to the end of the read string rather than to the beginning, the call

```
bgets (buffer, sizeof buffer, stream, "\n");
```

is identical to

```
fgets (buffer, sizeof buffer, stream);
```

There is always enough room reserved in the buffer for the trailing null.

If *breakstring* is a null pointer, the value of *breakstring* from the previous call is used. If *breakstring* is null at the first call, no characters will be used to delimit the string.

EXAMPLES

```
#include    <libgen.h>

char buffer[8];
/* read in first user name from /etc/passwd */
fp = fopen("/etc/passwd","r");
bgets(buffer, 8, fp, ":");
```

DIAGNOSTICS

NULL is returned on error or end-of-file. Reporting the condition is delayed to the next call if any characters were read but not yet returned.

SEE ALSO

gets(3S).

NAME

bsearch – binary search a sorted table

SYNOPSIS

```
#include <stdlib.h>

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

DESCRIPTION

bsearch is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table (an array) indicating where a datum may be found or a null pointer if the datum cannot be found. The table must be previously sorted in increasing order according to a comparison function pointed to by *compar*. *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. *size* is the number of bytes in each element. The function pointed to by *compar* 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 0 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 program reads in strings and either finds the corresponding node and prints out the string and its length, or prints an error message.

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

struct node {                     /* these are stored in the table */
        char *string;
        int length;
};
static struct node table[] =  /* table to be searched */
{
        { "asparagus", 10 },
        { "beans", 6 },
        { "tomato", 7 },
        { "watermelon", 11 },
};

main()
{
        struct node *node_ptr, node;
        /* routine to compare 2 nodes */
        static int node_compare(const void *, const void *);
        char str_space[20];   /* space to read string into */

        node.string = str_space;
        while (scanf("%20s", node.string) != EOF) {
                node_ptr = bsearch( &node,
                           table, sizeof(table)/sizeof(struct node),
                           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: %20s\n", node.string);
                }
        }
        return(0);
}

/* routine to compare two nodes based on an  */
/* alphabetical ordering of the string field */
static int
node_compare(const void *node1, const void *node2)
{
        return (strcmp(
                        ((const struct node *)node1)->string,
                        ((const struct node *)node2)->string));
}
```

DIAGNOSTICS

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

SEE ALSO

hsearch(3C), lsearch(3C), qsort(3C), tsearch(3C).

NOTES

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

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 void *, the value returned should be cast into type pointer-element.

If the number of elements in the table is less than the size reserved for the table, *nel* should be the lower number.

NAME

bufsplit – split buffer into fields

SYNOPSIS

cc [*flag* ...] *file* ... -lgen [*library* ...]

```
#include <libgen.h>
```

size_t *bufsplit* (char **buf*, size_t *n*, char ***a*);

DESCRIPTION

bufsplit examines the buffer, *buf*, and assigns values to the pointer array, *a*, so that the pointers point to the first *n* fields in *buf* that are delimited by tabs or new-lines.

To change the characters used to separate fields, call bufsplit with *buf* pointing to the string of characters, and *n* and *a* set to zero. For example, to use ':', '.', and ',' as separators along with tab and new-line:

```
bufsplit (":.,\t\n", 0, (char**)0 );
```

RETURN VALUE

The number of fields assigned in the array *a*. If *buf* is zero, the return value is zero and the array is unchanged. Otherwise the value is at least one. The remainder of the elements in the array are assigned the address of the null byte at the end of the buffer.

EXAMPLES

```
/*
 * set a[0] = "This", a[1] = "is", a[2] = "a",
 * a[3] = "test"
 */
bufsplit("This\tis\ta\ttest\n", 4, a);
```

SEE ALSO

sefbuf(3S), setbuffer(3C).

NOTES

bufsplit changes the delimiters to null bytes in *buf*.

NAME

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. These routines are defined as null macros in the include file netinet/in.h.

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

SEE ALSO

gethostent(3N), getservent(3N).

NAME

bzero – zero a portion of memory

SYNOPSIS

```
int bzero(), length;
char *ptr;
...
bzero(ptr, length);
```

where:

ptr A pointer to the area to clear
length The number of bytes set to zero

DESCRIPTION

The bzero function fills a specified memory area with zeros. The size of the area of memory is equal to *length.*

This function comes from the University of California Berkeley UNIX (BSD) system.

RETURNS

The bzero function returns no value.

EXAMPLE

```
/* Program test for the bzero() function */

#include <stdio.h>

int bzero();
char buf[80] = "abcdef";

main() {
    printf("buf = '%s'\n", buf);
    bzero(buf, sizeof(buf));
    printf("buf = '%s'\n", buf);
    return 0;
}
```

A call to this program generates the output

```
buf = 'abcdef'
buf = ''
```

SEE ALSO

memory(3C).

NAME

`catgets` – read a program message

SYNOPSIS

```
#include <nl_types.h>
```

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

DESCRIPTION

`catgets` attempts to read message *msg_num*, in set *set_num*, from the X/Open-style message catalogue identified by *catd*. *catd* is a catalogue descriptor returned from an earlier call to `catopen`. *s* points to a default message string which will be returned by `catgets` if the identified message catalogue is not currently available.

DIAGNOSTICS

If the identified message is retrieved successfully, `catgets` returns a pointer to an internal buffer area containing the null terminated message string. If the call is unsuccessful because the message catalogue identified by *catd* is not currently available, a pointer to *s* is returned.

SEE ALSO

`catopen`(3C).
`gettxt`(3C) — AT&T-style message facility.

NAME

catopen, catclose – open/close a message catalogue

SYNOPSIS

```
#include <nl_types.h>

nl_catd catopen (char *name, int oflag);

int catclose (nl_catd catd);
```

DESCRIPTION

catopen opens an X/Open-style message catalogue and returns a catalogue descriptor. *name* specifies the name of the message catalogue to be opened. If *name* contains a "/" then *name* specifies a pathname for the message catalogue. Otherwise, the environment variable NLSPATH is used. If NLSPATH does not exist in the environment, or if a message catalogue cannot be opened in any of the paths specified by NLSPATH, then the default path is used [see nl_types(5)].

The names of message catalogues, and their location in the filestore, can vary from one system to another. Individual applications can choose to name or locate message catalogues according to their own special needs. A mechanism is therefore required to specify where the catalogue resides.

The NLSPATH variable provides both the location of message catalogues, in the form of a search path, and the naming conventions associated with message catalogue files. For example:

```
NLSPATH=/usr/lib/nls/msg/%L/%N.cat:/usr/lib/nls/msg/%N/%L
```

The metacharacter % introduces a substitution field, where %L substitutes the current setting of the LANG environment variable (see following section), and %N substitutes the value of the *name* parameter passed to catopen. Thus, in the above example, catopen will search in /usr/lib/nls/msg/$LANG/*name*.cat, then in /usr/lib/nls/msg/*name*/$LANG, for the required message catalogue.

NLSPATH will normally be set up on a system wide basis (e.g., in /etc/profile) and thus makes the location and naming conventions associated with message catalogues transparent to both programs and users.

The full set of metacharacters is:

%N	The value of the name parameter passed to catopen.
%L	The value of LANG.
%l	The value of the language element of LANG.
%t	The value of the territory element of LANG.
%c	The value of the codeset element of LANG.
%%	A single %.

The LANG environment variable provides the ability to specify the user's requirements for native languages, local customs and character set, as an ASCII string in the form

```
LANG=language[_territory[.codeset]]
```

A user who speaks German as it is spoken in Austria and has a terminal which operates in the PC 850 codeset, would want the setting of the LANG variable to be

```
LANG=de_AT.850
```

With this setting it should be possible for that user to find any relevant catalogues should they exist.

Should the LANG variable not be set then the value of LC_MESSAGES as returned by setlocale is used. If this is NULL then the default path as defined in nl_types is used.

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

catclose closes the message catalogue identified by *catd*.

DIAGNOSTICS

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

catclose returns 0 if successful, otherwise −1.

SEE ALSO

gencat(1), catexstr(1), catgets(3C), setlocale(3C), environ(5), nl_types(5).
mkmsgs(1), gettxt(3C) — AT&T-style message facilities.

NAME

cfgetispeed, cfgetospeed, cfsetispeed, cfsetospeed – baud rate functions

SYNOPSIS

```
#include <termios.h>

speed_t cfgetospeed (termios_p)
struct termios *termios_p;

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

speed_t cfgetispeed (termios_p)
struct termios *termios_p;

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

DESCRIPTION

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

The input and output baud rates are stored in the *termios* structure. The values shown in the table "*termios* Baud Rate Values" are supported. The name symbols in this table are defined in <termios.h>.

termios Baud Rate Values

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

The type *speed_t* shall be defined in <termios.h> and shall be an unsigned integral type.

The *termios_p* argument is a pointer to a *termios* structure.

cfgetospeed() returns the output baud rate stored in the *termios* structure pointed to by *termios_p*.

cfsetospeed() sets the output baud rate stored in the *termios* structure pointed to by *termios_p* to *speed*. The zero baud rate, 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.

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

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

RETURNS

See DESCRIPTION.

DIAGNOSTICS

This standard does not specify any error conditions that are required to be detected for the `cfgetispeed()`, `cfgetospeed()`, `cfsetispeed()`, or `cfsetospeed()` functions. Some errors may be detected under implementation-defined conditions.

SEE ALSO

`tcsetattr`(3C).

COPYRIGHTS

Portions of this text are reprinted from IEEE Std 1003.1-1988, *Portable Operating System Interface for Computer Environment*, copyright © 1988 by the Institute of Electrical and Electronics Engineers, Inc., with the permission of the IEEE Standards Department. To purchase IEEE Standards, call 800/678-IEEE.

In the event of a discrepancy between the electronic and the original printed version, the original version takes precedence.

STANDARDS

Attempts to set unsupported baud rates will be ignored, and will not cause any error to be returned by the functions `cfsetispeed()` or `cfsetospeed()`.

There are no implementation-defined conditions under which the `cfsetispeed()` or `cfsetospeed()` functions will detect additional errors.

NAME

clock – report CPU time used

SYNOPSIS

```
#include <time.h>

clock_t clock (void);
```

DESCRIPTION

clock returns the amount of CPU time (in microseconds) used since the first call to clock by the calling process. 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 the wait(2) system call, the pclose(3S) function, or the system(3S) function.

Dividing the value returned by clock by the constant CLOCKS_PER_SEC, defined in the time.h header file, will give the time in seconds.

The resolution of the clock is 10 milliseconds on Data General processors.

SEE ALSO

times(2), wait(2), popen(3S), system(3S).

NOTES

The value returned by clock is defined in microseconds for compatibility with systems that have CPU clocks with much higher resolution. Because of this, the value returned will wrap around after accumulating only 2147 seconds of CPU time (about 36 minutes). If the process time used is not available or cannot be represented, clock returns the value (clock_t)-1.

NAME

conv: toupper, tolower, _toupper, _tolower, toascii – translate characters

SYNOPSIS

```
#include <ctype.h>

int toupper (int c);

int tolower (int c);

int _toupper (int c);

int _tolower (int c);

int toascii (int c);
```

DESCRIPTION

toupper and tolower have as their domain the range of the function getc: all values represented in an unsigned char and the value of the macro EOF as defined in stdio.h. If the argument of toupper represents a lower-case letter, the result is the corresponding upper-case letter. If the argument of tolower represents an upper-case letter, the result is the corresponding lower-case letter. All other arguments in the domain are returned unchanged.

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

toascii yields its argument with all bits turned off that are not part of a standard 7-bit ASCII character; it is intended for compatibility with other systems.

toupper, tolower, _toupper, and_tolower are affected by LC_CTYPE. In the C locale, or in a locale where shift information is not defined, these functions determine the case of characters according to the rules of the ASCII-coded character set. Characters outside the ASCII range of characters are returned unchanged.

SEE ALSO

ctype(3C), getc(3S), setlocale(3C), environ(5).

NAME

copylist – copy a file into memory

SYNOPSIS

cc [*flag* ...] *file* ... -lgen [*library* ...]

```
#include <libgen.h>
```

```
char *copylist (const char *filenm, off_t *szptr);
```

DESCRIPTION

copylist copies a list of items from a file into freshly allocated memory, replacing new-lines with null characters. It expects two arguments: a pointer *filenm* to the name of the file to be copied, and a pointer *szptr* to a variable where the size of the file will be stored.

Upon success, copylist returns a pointer to the memory allocated. Otherwise it returns NULL if it has trouble finding the file, calling malloc, or opening the file.

EXAMPLES

```
/* read "file" into buf */
off_t size;
char *buf;
buf = copylist("file", &size);
for (i = 0; i < size; i++)
      if(buf[i])
            putchar(buf[i]);
      else
            putchar('\n');
```

SEE ALSO

malloc(3C).

NAME

crypt, setkey, encrypt – generate encryption

SYNOPSIS

```
#include <crypt.h>

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

void setkey (const char *key);

void encrypt (char *block, int edflag);
```

DESCRIPTION

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

key is the input string to encrypt, for instance, a user's typed password. Only the first eight characters are used; the rest are ignored. *salt* is a two-character string chosen from the set a-zA-Z0-9./; this string is used to perturb the hashing algorithm in one of 4096 different ways, after which the input string is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted input string. The first two characters of the return value are the *salt* itself.

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

The *block* argument of encrypt is a character array of length 64 containing only the characters with numerical value 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the hashing algorithm using the key set by setkey. The argument *edflag*, indicating decryption rather than encryption, is ignored; use encrypt in libcrypt [see crypt(3X)] for decryption.

SEE ALSO

getpass(3C), crypt(3X), passwd(4).
login(1), passwd(1) in the *User's Reference for the DG/UX System*.

DIAGNOSTICS

If *edflag* is set to anything other than zero, errno will be set to ENOSYS.

NOTES

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

NAME

crypt – password and file encryption functions

SYNOPSIS

```
cc [flag ...] file ...  -lcrypt [library ...]

#include <crypt.h>

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

void setkey (const char *key);

void encrypt (char *block, int flag);

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

void des_setkey (const char *key);

void des_encrypt (char *block, int flag);

int run_setkey (int *p, const char *key);

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

int crypt_close(int *p);
```

DESCRIPTION

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

key is a user's typed password. *salt* is a two-character string chosen from the set [a-zA-Z0-9./]; this string is used to perturb the hashing algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password. The first two characters are the salt itself.

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

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

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

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

The routines run_setkey and run_crypt are designed for use by applications that need cryptographic capabilities [such as ed(1) and vi(1)] that must be compatible with the crypt(1) user-level utility. run_setkey establishes a two-way pipe connection with the crypt utility, using *key* as the password argument. run_crypt takes a block of characters and transforms the cleartext or ciphertext into their ciphertext or cleartext using the `crypt` utility. *offset* is the relative byte position from the beginning of the file that the block of text provided in *block* is coming from. *count* is the number of characters in *block*, and *connection* is an array containing indices to a table of input and output file streams. When encryption is finished, `crypt_close` is used to terminate the connection with the `crypt` utility.

`run_setkey` returns −1 if a connection with the `crypt` utility cannot be established. This result will occur in international versions of the DG/UX system in which the `crypt` utility is not available. If a null key is passed to `run_setkey`, 0 is returned. Otherwise, 1 is returned. `run_crypt` returns −1 if it cannot write output or read input from the pipe attached to `crypt`. Otherwise it returns 0.

The program must be linked with the object file access routine library `libcrypt.a`.

DIAGNOSTICS

In the international version of `crypt`(3X), a flag argument of 1 to `encrypt` or `des_encrypt` is not accepted, and `errno` is set to `ENOSYS` to indicate that the functionality is not available.

SEE ALSO

`getpass`(3C), `passwd`(4).
`crypt`(1), `login`(1), `passwd`(1) in the *User's Reference for the DG/UX System*.

NOTES

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

NAME

ctermid – generate file name for terminal

SYNOPSIS

```
#include <stdio.h>

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

DESCRIPTION

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

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

SEE ALSO

ttyname(3C).

NOTES

The difference between ctermid and ttyname(3C) is that ttyname must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while ctermid returns a string (/dev/tty) that 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.

NAME

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

SYNOPSIS

```
#include <time.h>

char *ctime (const time_t *clock);

struct tm *localtime (const time_t *clock);

struct tm *gmtime (const time_t *clock);

char *asctime (const struct tm *tm);

extern time_t timezone, altzone;

extern int daylight;

extern char *tzname[2];

void tzset (void);
```

DESCRIPTION

ctime, localtime, and gmtime accept arguments of type time_t, pointed to by clock, representing the time in seconds since 00:00:00 UTC, January 1, 1970. ctime returns a pointer to a 26-character string as shown below. Time zone and daylight savings corrections are made before the string is generated. The fields are constant in width:

```
Fri Sep 13 00:00:00 1986\n\0
```

localtime and gmtime return pointers to tm structures, described below. localtime corrects for the main time zone and possible alternate ("daylight savings") time zone; gmtime converts directly to Coordinated Universal Time (UTC), which is the time the DG/UX system uses internally.

asctime converts a tm structure to a 26-character string, as shown in the above example, and returns a pointer to the string.

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

```
struct    tm {
     int    tm_sec;      /* seconds after the minute    [0, 61] */
                            /* for leap seconds */
     int    tm_min;      /* minutes after the hour    [0, 59] */
     int    tm_hour;     /* hour since midnight    [0, 23] */
     int    tm_mday;     /* day of the month    [1, 31] */
     int    tm_mon;      /* months since January    [0, 11] */
     int    tm_year;     /* years since 1900 */
     int    tm_wday;     /* days since Sunday    [0, 6] */
     int    tm_yday;     /* days since January 1    [0, 365] */
     int    tm_isdst;    /* flag for alternate daylight */
                            /* savings time */
  };
```

The value of tm_isdst is positive if daylight savings time is in effect, zero if daylight savings time is not in effect, and negative if the information is not available. (Previously, the value of tm_isdst was defined as non-zero if daylight savings time was in effect.)

The external time_t variable altzone contains the difference, in seconds, between Coordinated Universal Time and the alternate time zone. The external variable timezone contains the difference, in seconds, between UTC and local standard time (in EST, timezone is 5*60*60). The external variable daylight indicates whether time should reflect daylight savings time. Both timezone and altzone default to 0 (UTC). The external variable daylight is non-zero if an alternate time zone exists. The time zone names are contained in the external variable tzname, which by default is set to:

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

These functions know about the peculiarities of this conversion for various time periods for the U.S. (specifically, the years 1974, 1975, and 1987). They will handle the new daylight savings time starting with the first Sunday in April, 1987.

tzset uses the contents of the environment variable TZ to override the value of the different external variables. The function tzset is called by asctime and may also be called by the user. See environ(5) for a description of the TZ environment variable.

tzset scans the contents of the environment variable and assigns the different fields to the respective variable. For example, the most complete setting for New Jersey in 1986 could be

```
EST5EDT4,116/2:00:00,298/2:00:00
```

or simply

```
EST5EDT
```

An example of a southern hemisphere setting such as the Cook Islands could be

```
KDT9:30KST10:00,63/5:00,302/20:00
```

In the longer version of the New Jersey example of TZ, tzname[*0*] is EST, timezone will be set to 5*60*60, tzname[*1*] is EDT, altzone will be set to 4*60*60, the starting date of the alternate time zone is the 117th day at 2 AM, the ending date of the alternate time zone is the 299th day at 2 AM (using zero-based Julian days), and daylight will be set positive. Starting and ending times are relative to the alternate time zone. If the alternate time zone start and end dates and the time are not provided, the days for the United States that year will be used and the time will be 2 AM. If the start and end dates are provided but the time is not provided, the time will be 2 AM. The effects of tzset are thus to change the values of the external variables timezone, altzone, daylight, and tzname. ctime, localtime, mktime, and strftime will also update these external variables as if they had called tzset at the time specified by the time_t or struct tm value that they are converting.

Note that in most installations, TZ is set to the correct value by default when the user logs on, via the local /etc/profile file [see profile(4) and timezone(4)].

time(2) is quite useful for producing the values with which to call ctime(3C).

FILES

/usr/lib/locale/*language*/LC_TIME – file containing locale specific date and time information

SEE ALSO

time(2), getenv(3C), difftime(3C), mktime(3C), putenv(3C), printf(3S), setlocale(3C), strftime(3C), cftime(4), profile(4), timezone(4), environ(5), zic(1).

NOTES

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

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

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

NAME

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

SYNOPSIS

```
#include <ctype.h>

int isalpha(int c);

int isupper(int c);

int islower(int c);

int isdigit(int c);

int isxdigit(int c);

int isalnum(int c);

int isspace(int c);

int ispunct(int c);

int isprint(int c);

int isgraph(int c);

int iscntrl(int c);

int isascii(int c);
```

DESCRIPTION

These macros classify character-coded integer values. Each is a predicate returning non-zero for true, zero for false. The behavior of these macros, except isascii, is affected by the current locale [see setlocale(3C)]. To modify the behavior, change the LC_TYPE category using setlocale, that is, setlocale (LC_CTYPE, *newlocale*), or setlocale (LC_ALL, *newlocale*). In the C locale, or in a locale where character type information is not defined, characters are classified according to the rules of the US-ASCII 7-bit coded character set.

The macro isascii is defined on all integer values; the rest are defined only where the argument is an int, the value of which is representable as an unsigned char, or EOF, which is defined by the stdio.h header file and represents end-of-file.

isalpha — tests for any character for which isupper or islower is true, or any character that is one of an implementation-defined set of characters for which none of iscntrl, isdigit, ispunct, or isspace is true. In the C locale, isalpha returns true only for the characters for which isupper or islower is true.

isupper — tests for any character that is an upper-case letter or is one of an implementation-defined set of characters for which none of iscntrl, isdigit, ispunct, isspace, or islower is true. In the C locale, isupper returns true only for the characters defined as upper-case ASCII characters.

islower — tests for any character that is a lower-case letter or is one of an implementation-defined set of characters for which none of iscntrl, isdigit, ispunct, isspace, or isupper is true. In the C locale, islower returns true only for the characters defined as lower-case ASCII characters.

isdigit tests for any decimal-digit character.

isxdigit tests for any hexadecimal-digit character ([0-9], [A-F] or [a-f]).

isalnum tests for any character for which isalpha or isdigit is true (letter or digit).

isspace tests for any space, tab, carriage-return, newline, vertical-tab or form-feed (standard white-space characters) or for one of an implementation-defined set of characters for which isalnum is false. In the C locale, isspace returns true only for the standard white-space characters.

ispunct tests for any printing character which is neither a space nor a character for which isalnum is true.

isprint tests for any printing character, including space (" ").

isgraph tests for any printing character, except space.

iscntrl tests for any "control character" as defined by the character set.

isascii tests for any ASCII character, code between 0 and 0177 inclusive.

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

Functions exist for all the above-defined macros. To get the function form, the macro name must be undefined (e.g., #undef isdigit).

FILES

/usr/lib/locale/*locale*/LC_CTYPE

DIAGNOSTICS

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

SEE ALSO

chrtbl(1M), setlocale(3C), stdio(3S), ascii(5), environ(5).

NAME

curs_addch: addch, waddch, mvaddch, mvwaddch, echochar, wechochar – add a character (with attributes) to a curses window

SYNOPSIS

```
#include <curses.h>

addch(chtype ch);

waddch(WINDOW *win, chtype ch);

mvaddch(int y, int x, chtype ch);

mvwaddch(WINDOW *win, int y, int x, chtype ch);

echochar(chtype ch);

wechochar(WINDOW *win, chtype ch);
```

DESCRIPTION

With the addch, waddch, mvaddch and mvwaddch routines, 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. At the right margin, an automatic newline is performed. At the bottom of the scrolling region, if scrollok is enabled, the scrolling region is scrolled up one line.

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

Video attributes can be combined with a character by OR-ing them into the parameter. This results in these attributes also being set. (The intent here is that text, including attributes, can be copied from one place to another using inch and addch.) [see standout, predefined video attribute constants, on the curs_attr(3X) page].

The echochar and wechochar routines are functionally equivalent to a call to addch followed by a call to refresh, or a call to waddch followed by a call to wrefresh. The knowledge that only a single character is being output is taken into consideration and, for non-control characters, a considerable performance gain might be seen by using these routines instead of their equivalents.

Line Graphics

The following variables may be used to add line drawing characters to the screen with routines of the addch family. When variables are defined for the terminal, the A_ALTCHARSET bit is turned on [see curs_attr(3X)]. Otherwise, the default character listed below is stored in the variable. The names chosen are consistent with the VT100 nomenclature.

Name	*Default*	*Glyph Description*
ACS_ULCORNER	+	upper left-hand corner
ACS_LLCORNER	+	lower left-hand corner
ACS_URCORNER	+	upper right-hand corner
ACS_LRCORNER	+	lower right-hand 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 VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion, unless otherwise noted in the preceding routine descriptions.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that addch, mvaddch, mvwaddch, and echochar may be macros.

SEE ALSO

curses(3X), curs_attr(3X), curs_clear(3X), curs_inch(3X), curs_outopts(3X), curs_refresh(3X) putc(3S).

NAME

curs_addchstr: addchstr, addchnstr, waddchstr, waddchnstr, mvaddchstr, mvaddchnstr, mvwaddchstr, mvwaddchnstr – add string of characters (and attributes) to a curses window

SYNOPSIS

```
#include <curses.h>

int addchstr(chtype *chstr);

int addchnstr(chtype *chstr, int n);

int waddchstr(WINDOW *win, chtype *chstr);

int waddchnstr(WINDOW *win, chtype *chstr, int n);

int mvaddchstr(int y, int x, chtype *chstr);

int mvaddchnstr(int y, int x, chtype *chstr, int n);

int mvwaddchstr(WINDOW *win, int y, int x, chtype *chstr);

int mvwaddchnstr(WINDOW *win, int y, int x,
      chtype *chstr, int n);
```

DESCRIPTION

All of these routines copy *chstr* directly into the window image structure starting at the current cursor position. The four routines with *n* as the last argument copy at most *n* elements, but no more than will fit on the line. If n=-1 then the whole string is copied, to the maximum number that fit on the line.

The position of the window cursor is NOT advanced. These routines works faster than waddnstr because they merely copy *chstr* into the window image structure. On the other hand, care must be taken when using these functions because they don't perform any kind of checking (such as for the newline character), they don't advance the current cursor position, and they truncate the string, rather then wrapping it around to the new line.

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion, unless otherwise noted in the preceding routine descriptions.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that all routines except waddchnstr may be macros.

SEE ALSO

curses(3X).

NAME

curs_addchstr: addchstr, addchnstr, waddchstr, waddchnstr, mvaddchstr, mvaddchnstr, mvwaddchstr, mvwaddchnstr – add string of characters (and attributes) to a curses window

SYNOPSIS

```
#include <curses.h>

int addchstr(chtype *chstr);

int addchnstr(chtype *chstr, int n);

int waddchstr(WINDOW *win, chtype *chstr);

int waddchnstr(WINDOW *win, chtype *chstr, int n);

int mvaddchstr(int y, int x, chtype *chstr);

int mvaddchnstr(int y, int x, chtype *chstr, int n);

int mvwaddchstr(WINDOW *win, int y, int x, chtype *chstr);

int mvwaddchnstr(WINDOW *win, int y, int x,
      chtype *chstr, int n);
```

DESCRIPTION

All of these routines copy *chstr* directly into the window image structure starting at the current cursor position. The four routines with *n* as the last argument copy at most *n* elements, but no more than will fit on the line. If `n=-1` then the whole string is copied, to the maximum number that fit on the line.

The position of the window cursor is NOT advanced. These routines works faster than `waddnstr` because they merely copy *chstr* into the window image structure. On the other hand, care must be taken when using these functions because they don't perform any kind of checking (such as for the newline character), they don't advance the current cursor position, and they truncate the string, rather then wrapping it around to the new line.

RETURN VALUE

All routines return the integer `ERR` upon failure and an integer value other than `ERR` upon successful completion, unless otherwise noted in the preceding routine descriptions.

NOTES

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that all routines except `waddchnstr` may be macros.

SEE ALSO

`curses`(3X).

NAME

curs_addstr: addstr, addnstr, waddstr, waddnstr, mvaddstr, mvaddnstr, mvwaddstr, mvwaddnstr – add a string of characters to a curses window and advance cursor

SYNOPSIS

```
#include <curses.h>

int addstr(char *str);

int addnstr(char *str, int n);

int waddstr(WINDOW *win, char *str);

int waddnstr(WINDOW *win, char *str, int n);

int mvaddstr(y, int x, char *str);

int mvaddnstr(y, int x, char *str, int n);

int mvwaddstr(WINDOW *win, int y, int x, char *str);

int mvwaddnstr(WINDOW *win, int y, int x, char *str,
      int n);
```

DESCRIPTION

All of these routines write all the characters of the null terminated character string *str* on the given window. It is similar to calling waddch once for each character in the string. The four routines with *n* as the last argument write at most *n* characters. If *n* is negative, then the entire string will be added.

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that all of these routines except waddstr and waddnstr may be macros.

SEE ALSO

curses(3X), curs_addch(3X).

NAME

curs_addwch: addwch, waddwch, mvaddwch, mvwaddwch, echowchar, wechowchar – add a wchar_t character to a curses window

SYNOPSIS

```
#include <curses.h>

int addwch(chtype wch);

int waddwch(WINDOW *win, chtype wch);

int mvaddwch(int y, int x, chtype wch);

int mvwaddwch(WINDOW *win, int y, int x, chtype wch);

int echowchar(chtype wch);

int wechowchar(WINDOW *win, chtype wch);
```

DESCRIPTION

With the addwch, waddwch, mvaddwch and mvwaddwch routines, the character *wch* which is holding a wchar_t character 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 putwchar in the C multibyte library. At the right margin, an automatic newline is performed. At the bottom of the scrolling region, if scrollok is enabled, the scrolling region is scrolled up one line.

If *wch* is a tab, newline, or backspace, the cursor is moved appropriately within the window. A newline also does a clrtoeol before moving. Tabs are considered to be at every eighth column. If *wch* is another control character, it is drawn in the *^X* notation. Calling winwch after adding a control character does not return the control character, but instead returns the representation of the control character.

The echowchar and wechowchar routines are functionally equivalent to a call to addwch fcllowed by a call to refresh, or a call to waddwch followed by a call to wrefresh. The knowledge that only a single character is being output is taken into consideration and, for non-control characters, a considerable performance gain might be seen by using these routines instead of their equivalents.

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion, unless otherwise noted in the preceding routine descriptions.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that addwch, mvaddwch, mvwaddwch, and echowchar may be macros.

SEE ALSO

curses(3X), curs_attr(3X), curs_clear(3X), curs_inwch(3X), curs_outopts(3X), curs_refresh(3X) putwc(3W).

NAME

curs_addwchstr: addwchstr, addwchnstr, waddwchstr, waddwchnstr, mvaddwchstr, mvaddwchnstr, mvwaddwchstr, mvwaddwchnstr – add string of wchar_t characters to a curses window

SYNOPSIS

```
#include <curses.h>

int addwchstr(chtype *wchstr);

int addwchnstr(chtype *wchstr, int n);

int waddwchstr(WINDOW *win, chtype *wchstr);

int waddwchnstr(WINDOW *win, chtype *wchstr, int n);

int mvaddwchstr(int y, int x, chtype *wchstr);

int mvaddwchnstr(int y, int x, chtype *wchstr, int n);

int mvwaddwchstr(WINDOW *win, int y, int x, chtype *wchstr);

int mvwaddwchnstr(WINDOW *win, int y, int x,
      chtype *wchstr, int n);
```

DESCRIPTION

All of these routines copy *wchstr* which points to the string of wchar_t characters directly into the window image structure starting at the current cursor position. The four routines with *n* as the last argument copy at most *n* elements, but no more than will fit on the line. If n=-1 then the whole string is copied, to the maximum number that fit on the line.

The position of the window cursor is NOT advanced. These routines works faster than waddnwstr because they merely copy *wchstr* into the window image structure. On the other hand, care must be taken when using these functions because they don't perform any kind of checking (such as for the newline character), they don't advance the current cursor position, and they truncate the string, rather then wrapping it around to the new line.

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion, unless otherwise noted in the preceding routine descriptions.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that all routines except waddwchnstr may be macros.

SEE ALSO

curses(3X).

NAME

curs_attr: attroff, wattroff, attron, wattron, attrset, wattrset, standend, wstandend, standout, wstandout – curses character and window attribute control routines

SYNOPSIS

```
#include <curses.h>

int attroff(int attrs);

int wattroff(WINDOW *win, int attrs);

int attron(int attrs);

int wattron(WINDOW *win, int attrs);

int attrset(int attrs);

int wattrset(WINDOW *win, int attrs);

int standend(void);

int wstandend(WINDOW *win);

int standout(void);

int wstandout(WINDOW *win);
```

DESCRIPTION

All of these routines manipulate the current attributes of the named window. The current attributes of a window are applied to all characters that are written into the window with waddch, waddstr and wprintw. Attributes are a property of the character, and move with the character through any scrolling and insert/delete line/character operations. To the extent possible on the particular terminal, they are displayed as the graphic rendition of characters put on the screen.

The routine attrset sets the current attributes of the given window to *attrs*. The routine attroff turns off the named attributes without turning any other attributes on or off. The routine attron turns on the named attributes without affecting any others. The routine standout is the same as attron(A_STANDOUT). The routine standend is the same as attrset(0), that is, it turns off all attributes.

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	Best highlighting mode of the terminal.
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 a character
COLOR_PAIR(*n*)	Color-pair number *n*

The following macro is the reverse of COLOR_PAIR(*n*):

PAIR_NUMBER(*attrs*)	Returns the pair number associated with the COLOR_PAIR(*n*) attribute.

RETURN VALUE

These routines always return 1.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that attroff, wattroff, attron, wattron, attrset, wattrset, standend and standout may be macros.

SEE ALSO

curses(3X), curs_addch(3X), curs_addstr(3X), curs_printw(3X).

NAME

curs_beep: beep, flash – curses bell and screen flash routines

SYNOPSIS

```
#include <curses.h>

int beep(void);

int flash(void);
```

DESCRIPTION

The beep and flash routines are used to signal the terminal user. The routine beep sounds the audible alarm on the terminal, if possible; if that is not possible, it flashes the screen (visible bell), if that is possible. The routine flash flashes the screen, and if that is not possible, sounds the audible signal. If neither signal is possible, nothing happens. Nearly all terminals have an audible signal (bell or beep), but only some can flash the screen.

RETURN VALUE

These routines always return OK.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

SEE ALSO

curses(3X).

NAME

curs_bkgd: bkgdset, wbkgdset, bkgd, wbkgd – curses window background manipulation routines

SYNOPSIS

```
#include <curses.h>

void bkgdset(chtype ch);

void wbkgdset(WINDOW *win, chtype ch);

int bkgd(chtype ch);

int wbkgd(WINDOW *win, chtype ch);
```

DESCRIPTION

The bkgdset and wbkgdset routines manipulate the background of the named window. Background is a chtype consisting of any combination of attributes and a character. The attribute part of the background is combined (ORed) with all non-blank characters that are written into the window with waddch. Both the character and attribute parts of the background are combined with the blank characters. The background becomes a property of the character and moves with the character through any scrolling and insert/delete line/character operations. To the extent possible on a particular terminal, the attribute part of the background is displayed as the graphic rendition of the character put on the screen.

The bkgd and wbkgd routines combine the new background with every position in the window. Background is any combination of attributes and a character. Only the attribute part is used to set the background of non-blank characters, while both character and attributes are used for blank positions. To the extent possible on a particular terminal, the attribute part of the background is displayed as the graphic rendition of the character put on the screen.

RETURN VALUE

bkgd and wbkgd return the integer OK, or a non-negative integer, if immedok is set.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that bkgdset and bkgd may be macros.

SEE ALSO

curses(3X), curs_addch(3X), curs_outopts(3X).

NAME

curs_border: border, wborder, box, whline, wvline – create curses borders, horizontal and vertical lines

SYNOPSIS

```
#include <curses.h>

int border(chtype ls, chtype rs, chtype ts, chtype bs,
   chtype tl, chtype tr, chtype bl, chtype br);

int wborder(WINDOW *win, chtype ls, chtype rs,
   chtype ts, chtype bs, chtype tl, chtype tr,
   chtype bl, chtype br);

int box(WINDOW *win, chtype verch, chtype horch);

int hline(chtype ch, int n);

int whline(WINDOW *win, chtype ch, int n);

int vline(chtype ch, int n);

int wvline(WINDOW *win, chtype ch, int n);
```

DESCRIPTION

With the border, wborder and box routines, a border is drawn around the edges of the window. The argument *ls* is a character and attributes used for the left side of the border, *rs* - right side, *ts* - top side, *bs* - bottom side, *tl* - top left-hand corner, *tr* - top right-hand corner, *bl* - bottom left-hand corner, and *br* - bottom right-hand corner. If any of these arguments is zero, then the following default values (defined in <curses.h>) are used instead: ACS_VLINE, ACS_VLINE, ACS_HLINE, ACS_HLINE, ACS_ULCORNER, ACS_URCORNER, ACS_BLCORNER, ACS_BRCORNER.

box(*win*, *verch*, *horch*) is a shorthand for the following call: wborder(*win*, *verch*, *verch*, *horch*, *horch*, 0, 0, 0, 0).

hline and whline draw a horizontal (left to right) line using *ch* starting at the current cursor position in the window. The current cursor position is not changed. The line is at most *n* characters long, or as many as fit into the window.

vline and wvline draw a vertical (top to bottom) line using *ch* starting at the current cursor position in the window. The current cursor position is not changed. The line is at most *n* characters long, or as many as fit into the window.

RETURN VALUE

All routines return the integer OK, or a non-negative integer if immedok is set.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that border and box may be macros.

SEE ALSO

curses(3X), curs_outopts(3X).

NAME

curs_clear: erase, werase, clear, wclear, clrtobot, wclrtobot, clrtoeol, wclrtoeol – clear all or part of a curses window

SYNOPSIS

```
# include <curses.h>

int erase(void);

int werase(WINDOW *win);

int clear(void);

int wclear(WINDOW *win);

int clrtobot(void);

int wclrtobot(WINDOW *win);

int clrtoeol(void);

int wclrtoeol(WINDOW *win);
```

DESCRIPTION

The erase and werase routines copy blanks to every position in the window.

The clear and wclear routines are like erase and werase, but they also call clearok, so that the screen is cleared completely on the next call to wrefresh for that window and repainted from scratch.

The clrtobot and wclrtobot routines erase all lines below the cursor in the window. Also, the current line to the right of the cursor, inclusive, is erased.

The clrtoeol and wclrtoeol routines erase the current line to the right of the cursor, inclusive.

RETURN VALUE

All routines return the integer OK, or a non-negative integer if immedok is set.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that erase, werase, clear, wclear, clrtobot, and clrtoeol may be macros.

SEE ALSO

curses(3X), curs_outopts(3X), curs_refresh(3X).

NAME

curs_color: start_color, init_pair, init_color, has_colors, can_change_color, color_content, pair_content – curses color manipulation routines

SYNOPSIS

```
# include <curses.h>

int start_color(void);

int init_pair(short pair, short f, short b);

int init_color(short color, short r, short g, short b);

bool has_colors(void);

bool can_change_color(void);

int color_content(short color, short *r, short *g, short *b);

int pair_content(short pair, short *f, short *b);
```

DESCRIPTION

Overview

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

Routine Descriptions

The start_color routine requires no arguments. It must be called if the programmer wants to use colors, and before any other color manipulation routine is called. It is good practice to call this routine right after initscr. start_color initializes eight basic colors (black, red, green, yellow, blue, magenta, cyan, and white), and two global variables, COLORS and COLOR_PAIRS (respectively defining the maximum number of colors and color-pairs the terminal can support). It also restores the colors on the terminal to the values they had when the terminal was just turned on.

The init_pair routine changes the definition of a color-pair. It takes three arguments: the number of the color-pair to be changed, the foreground color number, and the background color number. The value of the first argument must be between 1 and COLOR_PAIRS-1. The value of the second and third arguments must be between 0 and COLORS. If the color-pair was previously initialized, the screen is refreshed and all occurrences of that color-pair is changed to the new definition.

The init_color routine changes the definition of a color. It takes four arguments: the number of the color to be changed followed by three RGB values (for the amounts of red, green, and blue components). The value of the first argument must be between 0 and COLORS. (See the section **Colors** for the default color index.) Each of the last three arguments must be a value between 0 and 1000. When init_color is used, all occurrences of that color on the screen immediately change to the new definition.

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

The can_change_color routine requires no arguments. It returns TRUE if the terminal supports colors and can change their definitions; other, it returns FALSE. This routine facilitates writing terminal-independent programs.

The color_content routine gives users a way to find the intensity of the red, green, and blue (RGB) components in a color. It requires four arguments: the color number, and three addresses of shorts for storing the information about the amounts of red, green, and blue components in the given color. The value of the first argument must be between 0 and COLORS. The values that are stored at the addresses pointed to by the last three arguments are between 0 (no component) and 1000 (maximum amount of component).

The pair_content routine allows users to find out what colors a given color-pair consists of. It requires three arguments: the color-pair number, and two addresses of shorts for storing the foreground and the background color numbers. The value of the first argument must be between 1 and COLOR_PAIRS-1. The values that are stored at the addresses pointed to by the second and third arguments are between 0 and COLORS.

Colors

In <curses.h> the following macros are defined. These are the default colors. curses also assumes that COLOR_BLACK is the default background color for all terminals.

```
COLOR_BLACK
COLOR_RED
COLOR_GREEN
COLOR_YELLOW
COLOR_BLUE
COLOR_MAGENTA
COLOR_CYAN
COLOR_WHITE
```

RETURN VALUE

All routines that return an integer return ERR upon failure and OK upon successful completion.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

SEE ALSO

curses(3X), curs_initscr(3X), curs_attr(3X).

NAME

curs_delch: delch, wdelch, mvdelch, mvwdelch – delete character under cursor in a curses window.

SYNOPSIS

```
#include <curses.h>

int delch(void);

int wdelch(WINDOW *win);

int mvdelch(int y, int x);

int mvwdelch(WINDOW *win, int y, int x);
```

DESCRIPTION

With these routines the character under the cursor in the window is deleted; all characters to the right of the cursor 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.)

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that delch, mvdelch, and mvwdelch may be macros.

SEE ALSO

curses(3X).

NAME

curs_deleteln: deleteln, wdeleteln, insdelln, winsdelln, insertln, winsertln – delete and insert lines in a curses window

SYNOPSIS

```
#include <curses.h>

int deleteln(void);

int wdeleteln(WINDOW *win);

int insdelln(int n);

int winsdelln(WINDOW *win, int n);

int insertln(void);

int winsertln(WINDOW *win);
```

DESCRIPTION

With the deleteln and wdeleteln routines, 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 a hardware delete line feature.)

With the insdelln and winsdelln routines, for positive *n*, insert *n* lines into the specified window above the current line. The *n* bottom lines are lost. For negative *n*, delete *n* lines (starting with the one under the cursor), and move the remaining lines up. The bottom *n* lines are cleared. The current cursor position remains the same.

With the insertln and insertln routines, a blank line is inserted above the current line and the bottom line is lost. (This does not imply use of a hardware insert line feature.)

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that all but winsdelln may be a macros.

SEE ALSO

curses(3X).

NAME

curs_getch: getch, wgetch, mvgetch, mvwgetch, ungetch – get (or push back) characters from curses terminal keyboard

SYNOPSIS

```
#include <curses.h>

int getch(void);

int wgetch(WINDOW *win);

int mvgetch(int y, int x);

int mvwgetch(WINDOW *win, int y, int x);

int ungetch(int ch);
```

DESCRIPTION

With the getch, wgetch, mvgetch and mvwgetch, routines a character is read from the terminal associated with the window. In no-delay mode, if no input is waiting, the value ERR is returned. In delay mode, the program waits until the system passes text through to the program. Depending on the setting of cbreak, this is after one character (cbreak mode), or after the first newline (nocbreak mode). In half-delay mode, the program waits 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.

If the window is not a pad, and it has been moved or modified since the last call to wrefresh, wrefresh will be called before another character is read.

If keypad is TRUE, and a function key is pressed, the token for that function key is returned instead of the raw characters. Possible function keys are defined in <curses.h> with integers beginning with 0401, whose names begin with KEY_. If a character that could be the beginning of a function key (such as escape) is received, curses sets a timer. If the remainder of the sequence does not come in within the designated time, the character is passed through; otherwise, the function key value is returned. For this reason, many terminals experience a delay between the time a user presses the escape key and the escape is returned to the program. Since tokens returned by these routines are outside the ASCII range, they are not printable.

The ungetch routine places *ch* back onto the input queue to be returned by the next call to wgetch.

Function Keys

The following function keys, defined in <curses.h>, might be returned by getch if keypad has been enabled. Note that not all of these may be supported on a particular terminal if the terminal does not transmit a unique code when the key is pressed or if the definition for the key is not present in the *terminfo* database.

Name	*Key name*
KEY_BREAK	Break key
KEY_DOWN	The four arrow keys ...
KEY_UP	
KEY_LEFT	
KEY_RIGHT	
KEY_HOME	Home key (upward+left arrow)
KEY_BACKSPACE	Backspace
KEY_F0	Function keys; space for 64 keys is reserved.
KEY_F(*n*)	For $0 \leq n \leq 63$
KEY_DL	Delete line
KEY_IL	Insert line
KEY_DC	Delete character
KEY_IC	Insert char or enter insert mode
KEY_EIC	Exit insert char mode
KEY_CLEAR	Clear screen
KEY_EOS	Clear to end of screen
KEY_EOL	Clear to end of line
KEY_SF	Scroll 1 line forward
KEY_SR	Scroll 1 line backward (reverse)
KEY_NPAGE	Next page
KEY_PPAGE	Previous page
KEY_STAB	Set tab
KEY_CTAB	Clear tab
KEY_CATAB	Clear all tabs
KEY_ENTER	Enter or send
KEY_SRESET	Soft (partial) reset
KEY_RESET	Reset or hard reset
KEY_PRINT	Print or copy
KEY_LL	Home down or bottom (lower left). Keypad is arranged like this: A1 up A3 left B2 right C1 down C3
KEY_A1	Upper left of keypad
KEY_A3	Upper right of keypad
KEY_B2	Center of keypad
KEY_C1	Lower left of keypad
KEY_C3	Lower right of keypad
KEY_BTAB	Back tab key
KEY_BEG	Beg(inning) key
KEY_CANCEL	Cancel key
KEY_CLOSE	Close key
KEY_COMMAND	Cmd (command) key
KEY_COPY	Copy key
KEY_CREATE	Create key
KEY_END	End key
KEY_EXIT	Exit key

KEY_FIND	Find key
KEY_HELP	Help key
KEY_MARK	Mark key
KEY_MESSAGE	Message key
KEY_MOVE	Move key
KEY_NEXT	Next object key
KEY_OPEN	Open key
KEY_OPTIONS	Options key
KEY_PREVIOUS	Previous object key
KEY_REDO	Redo key
KEY_REFERENCE	Ref(erence) key
KEY_REFRESH	Refresh key
KEY_REPLACE	Replace key
KEY_RESTART	Restart key
KEY_RESUME	Resume key
KEY_SAVE	Save key
KEY_SBEG	Shifted beginning key
KEY_SCANCEL	Shifted cancel key
KEY_SCOMMAND	Shifted command key
KEY_SCOPY	Shifted copy key
KEY_SCREATE	Shifted create key
KEY_SDC	Shifted delete char key
KEY_SDL	Shifted delete line key
KEY_SELECT	Select key
KEY_SEND	Shifted end key
KEY_SEOL	Shifted clear line key
KEY_SEXIT	Shifted exit key
KEY_SFIND	Shifted find key
KEY_SHELP	Shifted help key
KEY_SHOME	Shifted home key
KEY_SIC	Shifted input key
KEY_SLEFT	Shifted left arrow key
KEY_SMESSAGE	Shifted message key
KEY_SMOVE	Shifted move key
KEY_SNEXT	Shifted next key
KEY_SOPTIONS	Shifted options key
KEY_SPREVIOUS	Shifted prev key
KEY_SPRINT	Shifted print key
KEY_SREDO	Shifted redo key
KEY_SREPLACE	Shifted replace key
KEY_SRIGHT	Shifted right arrow
KEY_SRSUME	Shifted resume key
KEY_SSAVE	Shifted save key
KEY_SSUSPEND	Shifted suspend key
KEY_SUNDO	Shifted undo key
KEY_SUSPEND	Suspend key
KEY_UNDO	Undo key

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Use of the escape key by a programmer for a single character function is discouraged.

When using getch, wgetch, mvgetch, or mvwgetch, nocbreak mode (nocbreak) and echo mode (echo) should not be used at the same time. Depending on the state of the tty driver when each character is typed, the program may produce undesirable results.

Note that getch, mvgetch, and mvwgetch may be macros.

SEE ALSO

curses(3X), curs_inopts(3X), curs_move(3X), curs_refresh(3X).

NAME

curs_getstr: getstr, getnstr, wgetstr, wgetnstr, mvgetstr, mvgetnstr, mvwgetstr, mvwgetnstr – get character strings from curses terminal keyboard

SYNOPSIS

```
#include <curses.h>

int getstr(char *str);

int getnstr(char *str, int n);

int wgetstr(WINDOW *win, char *str);

int wgetnstr(WINDOW *win, char *str, int n);

int mvgetstr(int y, int x, char *str);

int mvgetnstr(int y, int x, char *str, int n);

int mvwgetstr(WINDOW *win, int y, int x, char *str);

int mvwgetnstr(WINDOW *win, int y, int x, char *str, int n);
```

DESCRIPTION

The effect of getstr is as though a series of calls to getch were made, until a newline and carriage return is received. The resulting value is placed in the area pointed to by the character pointer *str*. getnstr reads at most *n* characters, thus preventing a possible overflow of the input buffer. The user's erase and kill characters are interpreted, as well as any special keys (such as function keys, "home" key, "clear" key, *etc.*).

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that all routines except wgetnstr may be macros.

SEE ALSO

curses(3X), curs_getch(3X).

NAME

curs_getwch: getwch, wgetwch, mvgetwch, mvwgetwch, ungetwch – get (or push back) wchar_t characters from curses terminal keyboard

SYNOPSIS

```
#include <curses.h>

int getwch(void);

int wgetwch(WINDOW *win);

int mvgetwch(int y, int x);

int mvwgetwch(WINDOW *win, int y, int x);

int ungetwch(int wch);
```

DESCRIPTION

With the getwch, wgetwch, mvgetwch and mvwgetwch routines, a *EUC* character is read from the terminal associated with the window, it is transformed into a wchar_t character, and a wchar_t character is returned. In no-delay mode, if no input is waiting, the value ERR is returned. In delay mode, the program waits until the system passes text through to the program. Depending on the setting of cbreak, this is after one character (cbreak mode), or after the first newline (nocbreak mode). In half-delay mode, the program waits 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.

If the window is not a pad, and it has been moved or modified since the last call to wrefresh, wrefresh will be called before another character is read.

If keypad is TRUE, and a function key is pressed, the token for that function key is returned instead of the raw characters. Possible function keys are defined in <curses.h> with integers beginning with 0401, whose names begin with KEY_. If a character that could be the beginning of a function key (such as escape) is received, curses sets a timer. If the remainder of the sequence does not come in within the designated time, the character is passed through; otherwise, the function key value is returned. For this reason, many terminals experience a delay between the time a user presses the escape key and the escape is returned to the program.

The ungetwch routine places *wch* back onto the input queue to be returned by the next call to wgetwch.

Function Keys

The following function keys, defined in <curses.h>, might be returned by getwch if keypad has been enabled. Note that not all of these may be supported on a particular terminal if the terminal does not transmit a unique code when the key is pressed or if the definition for the key is not present in the *terminfo* database.

Name	*Key name*
KEY_BREAK	Break key
KEY_DOWN	The four arrow keys ...
KEY_UP	
KEY_LEFT	
KEY_RIGHT	
KEY_HOME	Home key (upward+left arrow)
KEY_BACKSPACE	Backspace
KEY_F0	Function keys; space for 64 keys is reserved.
KEY_F(*n*)	For $0 \leq n \leq 63$
KEY_DL	Delete line
KEY_IL	Insert line
KEY_DC	Delete character
KEY_IC	Insert char or enter insert mode
KEY_EIC	Exit insert char mode
KEY_CLEAR	Clear screen
KEY_EOS	Clear to end of screen
KEY_EOL	Clear to end of line
KEY_SF	Scroll 1 line forward
KEY_SR	Scroll 1 line backward (reverse)
KEY_NPAGE	Next page
KEY_PPAGE	Previous page
KEY_STAB	Set tab
KEY_CTAB	Clear tab
KEY_CATAB	Clear all tabs
KEY_ENTER	Enter or send
KEY_SRESET	Soft (partial) reset
KEY_RESET	Reset or hard reset
KEY_PRINT	Print or copy
KEY_LL	Home down or bottom (lower left). Keypad is arranged like this: A1 up A3 left B2 right C1 down C3
KEY_A1	Upper left of keypad
KEY_A3	Upper right of keypad
KEY_B2	Center of keypad
KEY_C1	Lower left of keypad
KEY_C3	Lower right of keypad
KEY_BTAB	Back tab key
KEY_BEG	Beg(inning) key
KEY_CANCEL	Cancel key
KEY_CLOSE	Close key
KEY_COMMAND	Cmd (command) key
KEY_COPY	Copy key
KEY_CREATE	Create key
KEY_END	End key
KEY_EXIT	Exit key

KEY_FIND	Find key
KEY_HELP	Help key
KEY_MARK	Mark key
KEY_MESSAGE	Message key
KEY_MOVE	Move key
KEY_NEXT	Next object key
KEY_OPEN	Open key
KEY_OPTIONS	Options key
KEY_PREVIOUS	Previous object key
KEY_REDO	Redo key
KEY_REFERENCE	Ref(erence) key
KEY_REFRESH	Refresh key
KEY_REPLACE	Replace key
KEY_RESTART	Restart key
KEY_RESUME	Resume key
KEY_SAVE	Save key
KEY_SBEG	Shifted beginning key
KEY_SCANCEL	Shifted cancel key
KEY_SCOMMAND	Shifted command key
KEY_SCOPY	Shifted copy key
KEY_SCREATE	Shifted create key
KEY_SDC	Shifted delete char key
KEY_SDL	Shifted delete line key
KEY_SELECT	Select key
KEY_SEND	Shifted end key
KEY_SEOL	Shifted clear line key
KEY_SEXIT	Shifted exit key
KEY_SFIND	Shifted find key
KEY_SHELP	Shifted help key
KEY_SHOME	Shifted home key
KEY_SIC	Shifted input key
KEY_SLEFT	Shifted left arrow key
KEY_SMESSAGE	Shifted message key
KEY_SMOVE	Shifted move key
KEY_SNEXT	Shifted next key
KEY_SOPTIONS	Shifted options key
KEY_SPREVIOUS	Shifted prev key
KEY_SPRINT	Shifted print key
KEY_SREDO	Shifted redo key
KEY_SREPLACE	Shifted replace key
KEY_SRIGHT	Shifted right arrow
KEY_SRSUME	Shifted resume key
KEY_SSAVE	Shifted save key
KEY_SSUSPEND	Shifted suspend key
KEY_SUNDO	Shifted undo key
KEY_SUSPEND	Suspend key
KEY_UNDO	Undo key

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Use of the escape key by a programmer for a single character function is discouraged.

When using getwch, wgetwch, mvgetwch, or mvwgetwch, nocbreak mode (nocbreak) and echo mode (echo) should not be used at the same time. Depending on the state of the tty driver when each character is typed, the program may produce undesirable results.

Note that getwch, mvgetwch, and mvwgetwch may be macros.

SEE ALSO

curses(3X), curs_inopts(3X), curs_move(3X), curs_refresh(3X).

NAME

curs_getwstr: getwstr, getnwstr, wgetwstr, wgetnwstr, mvgetwstr, mvgetnwstr, mvwgetwstr, mvwgetnwstr – get wchar_t character strings from curses terminal keyboard

SYNOPSIS

```
#include <curses.h>

int getwstr(wchar_t *wstr);

int getnwstr(wchar_t *wstr, int n);

int mvgetwstr(int y, int x, wchar_t *wstr);

int mvgetnwstr(int y, int x, wchar_t *wstr, int n);

int mvwgetwstr(WINDOW *win, int y, int x, wchar_t *wstr);

int mvwgetnwstr(WINDOW *win, int y, int x, wchar_t *wstr, int n);

int wgetwstr(WINDOW *win, wchar_t *wstr);

int wgetnwstr(WINDOW *win, wchar_t *wstr, int n);
```

DESCRIPTION

The effect of getwstr is as though a series of calls to getwch were made, until a newline and carriage return is received. The resulting value is placed in the area pointed to by the wchar_t pointer *str*. getnwstr reads at most *n* wchar_t characters, thus preventing a possible overflow of the input buffer. The user's erase and kill characters are interpreted, as well as any special keys (such as function keys, "home" key, "clear" key, *etc.*).

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that all routines except wgetnwstr may be macros.

SEE ALSO

curses(3X), curs_getwch(3X).

NAME

curs_getyx: getyx, getparyx, getbegyx, getmaxyx – get curses cursor and window coordinates

SYNOPSIS

```
#include <curses.h>

void getyx(WINDOW *win, int y, int x);

void getparyx(WINDOW *win, int y, int x);

void getbegyx(WINDOW *win, int y, int x);

void getmaxyx(WINDOW *win, int y, int x);
```

DESCRIPTION

With the getyx macro, the cursor position of the window is placed in the two integer variables *y* and *x*.

With the getparyx macro, if *win* is a subwindow, the beginning coordinates of the subwindow relative to the parent window are placed into two integer variables, *y* and *x*. Otherwise, -1 is placed into *y* and *x*.

Like getyx, the getbegyx and getmaxyx macros store the current beginning coordinates and size of the specified window.

RETURN VALUE

The return values of these macros are undefined (*i.e.*, they should not be used as the right-hand side of assignment statements).

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that all of these interfaces are macros and that "&" is not necessary before the variables *y* and *x*.

SEE ALSO

curses(3X).

NAME

curs_inch: inch, winch, mvinch, mvwinch – get a character and its attributes from a curses window

SYNOPSIS

```
#include <curses.h>

chtype inch(void);

chtype winch(WINDOW *win);

chtype mvinch(int y, int x);

chtype mvwinch(WINDOW *win, int y, int x);
```

DESCRIPTION

With these routines, the character, of type chtype, at the current position in the named window is returned. If any attributes are set for that position, their values are OR-ed into the value returned. Constants defined in <curses.h> can be used with the & (logical AND) operator to extract the character or attributes alone.

Attributes

The following bit-masks may be AND-ed with characters returned by winch.

A_CHARTEXT	Bit-mask to extract character
A_ATTRIBUTES	Bit-mask to extract attributes
A_COLOR	Bit-mask to extract color-pair field information

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that all of these routines may be macros.

SEE ALSO

curses(3X).

NAME

curs_inchstr: inchstr, inchnstr, winchstr, winchnstr, mvinchstr, mvinchnstr, mvwinchstr, mvwinchnstr – get a string of characters (and attributes) from a curses window

SYNOPSIS

```
#include <curses.h>

int inchstr(chtype *chstr);

int inchnstr(chtype *chstr, int n);

int winchstr(WINDOW *win, chtype *chstr);

int winchnstr(WINDOW *win, chtype *chstr, int n);

int mvinchstr(int y, int x, chtype *chstr);

int mvinchnstr(int y, int x, chtype *chstr, int n);

int mvwinchstr(WINDOW *win, int y, int x, chtype *chstr);

int mvwinchnstr(WINDOW *win, int y, int x, chtype *chstr, int n);
```

DESCRIPTION

With these routines, a string of type chtype, starting at the current cursor position in the named window and ending at the right margin of the window, is returned. The four functions with *n* as the last argument, return the string at most *n* characters long. Constants defined in <curses.h> can be used with the & (logical AND) operator to extract the character or the attribute alone from any position in the *chstr* [see curs_inch(3X)].

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that all routines except winchnstr may be macros.

SEE ALSO

curses(3X), curs_inch(3X).

NAME

curs_initscr: initscr, newterm, endwin, isendwin, set_term, delscreen − curses screen initialization and manipulation routines

SYNOPSIS

```
#include <curses.h>

WINDOW *initscr(void);

int endwin(void);

int isendwin(void);

SCREEN *newterm(char *type, FILE *outfd, FILE *infd);

SCREEN *set_term(SCREEN *new);

void delscreen(SCREEN* sp);
```

DESCRIPTION

initscr is almost always the first routine that should be called (the exceptions are slk_init, filter, ripoffline, use_env and, for multiple-terminal applications, newterm.) This determines the terminal type and initializes all curses data structures. initscr also causes the first call to refresh to clear the screen. If errors occur, initscr writes an appropriate error message to standard error and exits; otherwise, a pointer is returned to stdscr. If the program needs an indication of error conditions, newterm() should be used instead of initscr; initscr should only be called once per application.

A program that outputs to more than one terminal should use the newterm routine for each terminal instead of initscr. A program that needs an indication of error conditions, so it can continue to run in a line-oriented mode if the terminal cannot support a screen-oriented program, would also use this routine. The routine newterm should be called once for each terminal. It returns a variable of type SCREEN * which should be saved as a reference to that terminal. The arguments are the *type* of the terminal to be used in place of $TERM, a file pointer for output to the terminal, and another file pointer for input from the terminal (if *type* is NULL, $TERM will be used). The program must also call endwin for each terminal being used before exiting from curses. 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.

A program should always call endwin before exiting or escaping from curses mode temporarily. This routine restores tty modes, moves the cursor to the lower left-hand corner of the screen and resets the terminal into the proper non-visual mode. Calling refresh or doupdate after a temporary escape causes the program to resume visual mode.

The isendwin routine returns TRUE if endwin has been called without any subsequent calls to wrefresh, and FALSE otherwise.

The set_term routine is used to switch between different terminals. The screen reference new becomes the new current terminal. The previous terminal is returned by the routine. This is the only routine which manipulates SCREEN pointers; all other routines affect only the current terminal.

The delscreen routine frees storage associated with the SCREEN data structure. The endwin routine does not do this, so delscreen should be called after endwin if a particular SCREEN is no longer needed.

RETURN VALUE

endwin returns the integer ERR upon failure and OK upon successful completion.

Routines that return pointers always return NULL on error.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that initscr and newterm may be macros.

SEE ALSO

curses(3X), curs_kernel(3X), curs_refresh(3X), curs_slk(3X), curs_util(3X).

NAME

curs_inopts: cbreak, nocbreak, echo, noecho, halfdelay, intrflush, keypad, meta, nodelay, notimeout, raw, noraw, noqiflush, qiflush, timeout, wtimeout, typeahead – curses terminal input option control routines

SYNOPSIS

```
#include <curses.h>

int cbreak(void);

int nocbreak(void);

int echo(void);

int noecho(void);

int halfdelay(int tenths);

int intrflush(WINDOW *win, bool bf);

int keypad(WINDOW *win, bool bf);

int meta(WINDOW *win, bool bf);

int nodelay(WINDOW *win, bool bf);

int notimeout(WINDOW *win, bool bf);

int raw(void);

int noraw(void);

void noqiflush(void);

void qiflush(void);

void timeout(int delay);

void wtimeout(WINDOW *win, int delay);

int typeahead(int fd);
```

DESCRIPTION

The cbreak and nocbreak routines put the terminal into and out of cbreak mode, respectively. In this mode, characters typed by the user are immediately available to the program, and erase/kill character-processing is not performed. When out of this mode, the tty driver buffers the typed characters until a newline or carriage return is typed. Interrupt and flow control characters are unaffected by this mode. Initially the terminal may or may not be in cbreak mode, as the mode is inherited; therefore, a program should call cbreak or nocbreak explicitly. Most interactive programs using curses set the cbreak mode.

Note that cbreak overrides raw. [See curs_getch(3X) for a discussion of how these routines interact with echo and noecho.]

The echo and noecho 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 curs_getch(3X) for a discussion of how these routines interact with cbreak and nocbreak.]

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

If the intrflush option is enabled, (*bf* is TRUE), 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 (*bf* is FALSE), the option prevents the flush. The default for the option is inherited from the tty driver settings. The window argument is ignored.

The keypad option enables the keypad of the user's terminal. If enabled (*bf* is TRUE), the user can press a function key (such as an arrow key) and wgetch returns a single value representing the function key, as in KEY_LEFT. If disabled (*bf* is FALSE), curses does not treat function keys specially and the program has to interpret the escape sequences itself. If the keypad in the terminal can be turned on (made to transmit) and off (made to work locally), turning on this option causes the terminal keypad to be turned on when wgetch is called. The default value for keypad is false.

Initially, whether the terminal returns 7 or 8 significant bits on input depends on the control mode of the tty driver [see termio(7)]. To force 8 bits to be returned, invoke meta(*win*, TRUE). To force 7 bits to be returned, invoke meta(*win*, FALSE). The window argument, *win*, is always ignored. If the terminfo capabilities smm (meta_on) and rmm (meta_off) are defined for the terminal, smm is sent to the terminal when meta(*win*, TRUE) is called and rmm is sent when meta(*win*, FALSE) is called.

The nodelay option causes getch to be a non-blocking call. If no input is ready, getch returns ERR. If disabled (*bf* is FALSE), getch waits until a key is pressed.

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

With the raw and noraw routines, 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 all passed through uninterpreted, instead of generating a signal. The behavior of the BREAK key depends on other bits in the tty driver that are not set by curses.

When the noqiflush routine is used, normal flush of input and output queues associated with the INTR, QUIT and SUSP characters will not be done [see termio(7)]. When qiflush is called, the queues will be flushed when these control characters are read.

The timeout and wtimeout routines set blocking or non-blocking read for a given window. If *delay* is negative, blocking read is used (*i.e.*, waits indefinitely for input). If *delay* is zero, then non-blocking read is used (*i.e.*, read returns ERR if no input is waiting). If *delay* is positive, then read blocks for *delay* milliseconds, and returns ERR if there is still no input. Hence, these routines provide the same functionality as nodelay, plus the additional capability of being able to block for only *delay* milliseconds (where *delay* is positive).

curses does "line-breakout optimization" by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update is postponed until refresh or doupdate is called again. This allows faster response to commands typed in advance. Normally, the 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 *fd* is to be used to check for typeahead instead. If *fd* is −1, then no typeahead checking is done.

RETURN VALUE

All routines that return an integer return ERR upon failure and an integer value other than ERR upon successful completion, unless otherwise noted in the preceding routine descriptions.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that echo, noecho, halfdelay, intrflush, meta, nodelay, notimeout, noqiflush, qiflush, timeout, and wtimeout may be macros.

SEE ALSO

curses(3X), curs_getch(3X), curs_initscr(3X), termio(7).

NAME

curs_insch: insch, winsch, mvinsch, mvwinsch – insert a character before the character under the cursor in a curses window

SYNOPSIS

```
#include <curses.h>

int insch(chtype ch);

int winsch(WINDOW *win, chtype ch);

int mvinsch(int y, int x, chtype ch);

int mvwinsch(WINDOW *win, int y, int x, chtype ch);
```

DESCRIPTION

With these routines, the character *ch* is inserted before the character under the cursor. All characters to the right of the cursor are moved one space to the right, with the possibility of the rightmost character on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified). (This does not imply use of the hardware insert character feature.)

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that insch, mvinsch, and mvwinsch may be macros.

SEE ALSO

curses(3X).

NAME

curs_instr: insstr, insnstr, winsstr, winsnstr, mvinsstr, mvinsnstr, mvwinsstr, mvwinsnstr – insert string before character under the cursor in a curses window

SYNOPSIS

```
#include <curses.h>

int insstr(char *str);

int insnstr(char *str, int n);

int winsstr(WINDOW *win, char *str);

int winsnstr(WINDOW *win, char *str, int n);

int mvinsstr(int y, int x, char *str);

int mvinsnstr(int y, int x, char *str, int n);

int mvwinsstr(WINDOW *win, int y, int x, char *str);

int mvwinsnstr(WINDOW *win, int y, int x, char *str, int n);
```

DESCRIPTION

With these routines, a character string (as many characters as will fit on the line) is inserted before the character under the cursor. All characters to the right of the cursor are moved to the right, with the possibility of the rightmost characters on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified). (This does not imply use of the hardware insert character feature.) The four routines with *n* as the last argument insert at most *n* characters. If *n*<=0, then the entire string is inserted.

If a character in *str* is a tab, newline, carriage return or backspace, the cursor is moved appropriately within the window. A newline also does a clrtoeol before moving. Tabs are considered to be at every eighth column. If a character in *str* is another control character, it is drawn in the *^X* notation. Calling winch after adding a control character (and moving to it, if necessary) does not return the control character, but instead returns the representation of the control character.

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that all but winsnstr may be macros.

SEE ALSO

curses(3X), curs_clear(3X), curs_inch(3X).

NAME

curs_instr: instr, innstr, winstr, winnstr, mvinstr, mvinnstr, mvwinstr, mvwinnstr – get a string of characters from a curses window

SYNOPSIS

```
#include <curses.h>

int instr(char *str);

int innstr(char *str, int n);

int winstr(WINDOW *win, char *str);

int winnstr(WINDOW *win, char *str, int n);

int mvinstr(int y, int x, char *str);

int mvinnstr(int y, int x, char *str, int n);

int mvwinstr(WINDOW *win, int y, int x, char *str);

int mvwinnstr(WINDOW *win, int y, int x, char *str, int n);
```

DESCRIPTION

These routines return a string of characters in *str*, starting at the current cursor position in the named window and ending at the right margin of the window. Attributes are stripped from the characters. The four functions with *n* as the last argument return the string at most *n* characters long.

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that all routines except winnstr may be macros.

SEE ALSO

curses(3X).

NAME

curs_inswch: inswch, winswch, mvinswch, mvwinswch – insert a wchar_t character before the character under the cursor in a curses window

SYNOPSIS

```
#include <curses.h>

int inswch(chtype wch);

int winswch(WINDOW *win, chtype wch);

int mvinswch(int y, int x, chtype wch);

int mvwinswch(WINDOW *win, int y, int x, chtype wch);
```

DESCRIPTION

With these routines, the character *wch* holding a wchar_t character is inserted before the character under the cursor. All characters to the right of the cursor are moved one space to the right, with the possibility of the rightmost character on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified). (This does not imply use of the hardware insert character feature.)

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that inswch, mvinswch, and mvwinswch may be macros.

SEE ALSO

curses(3X).

NAME

curs_instr: inswstr, insnwstr, winswstr, winsnwstr, mvinswstr, mvinsnwstr, mvwinswstr, mvwinsnwstr – insert wchar_t string before character under the cursor in a curses window

SYNOPSIS

```
#include <curses.h>

int inswstr(char *wstr);

int insnwstr(char *wstr, int n);

int winswstr(WINDOW *win, char *wstr);

int winsnwstr(WINDOW *win, char *wstr, int n);

int mvinswstr(int y, int x, char *wstr);

int mvinsnwstr(int y, int x, char *wstr, int n);

int mvwinswstr(WINDOW *win, int y, int x, char *wstr);

int mvwinsnwstr(WINDOW *win, int y, int x, char *wstr, int n);
```

DESCRIPTION

With these routines, a wchar_t character string (as many wchar_t characters as will fit on the line) is inserted before the character under the cursor. All characters to the right of the cursor are moved to the right, with the possibility of the rightmost characters on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified). (This does not imply use of the hardware insert character feature.) The four routines with *n* as the last argument insert at most *n* wchar_t characters. If *n*<=0, then the entire string is inserted.

If a character in *wstr* is a tab, newline, carriage return or backspace, the cursor is moved appropriately within the window. A newline also does a clrtoeol before moving. Tabs are considered to be at every eighth column. If a character in *wstr* is another control character, it is drawn in the *^X* notation. Calling winch after adding a control character (and moving to it, if necessary) does not return the control character, but instead returns the representation of the control character.

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that all but winsnwstr may be macros.

SEE ALSO

curses(3X), curs_clear(3X), curs_inwch(3X).

NAME

curs_inwch: inwch, winwch, mvinwch, mvwinwch – get a wchar_t character from a curses window

SYNOPSIS

```
#include <curses.h>

chtype inwch(void);

chtype winwch(WINDOW *win);

chtype mvinwch(int y, int x);

chtype mvwinwch(WINDOW *win, int y, int x);
```

DESCRIPTION

With these routines, the wchar_t character, of type chtype, at the current position in the named window is returned. If any attributes are set for that position, their values are OR-ed into the value returned. Constants defined in <curses.h> can be used with the & (logical AND) operator to extract the character or attributes alone.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that all of these routines may be macros.

SEE ALSO

curses(3X).

NAME

curs_inwchstr: inwchstr, inwchnstr, winwchstr, winwchnstr, mvinwchstr, mvinwchnstr, mvwinwchstr, mvwinwchnstr – get a string of wchar_t characters from a curses window

SYNOPSIS

```
#include <curses.h>

int inwchstr(chtype *wchstr);

int inwchnstr(chtype *wchstr, int n);

int winwchstr(WINDOW *win, chtype *wchstr);

int winwchnstr(WINDOW *win, chtype *wchstr, int n);

int mvinwchstr(int y, int x, chtype *wchstr);

int mvinwchnstr(int y, int x, chtype *wchstr, int n);

int mvwinwchstr(WINDOW *win, int y, int x, chtype *wchstr);

int mvwinwchnstr(WINDOW *win, int y, int x, chtype *wchstr, int n);
```

DESCRIPTION

With these routines, a string of type chtype holding wchar_t characters, starting at the current cursor position in the named window and ending at the right margin of the window, is returned. The four functions with *n* as the last argument, return the string at most *n* wchar_t characters long. Constants defined in <curses.h> can be used with the & (logical AND) operator to extract the wchar_t character alone from any position in the *chstr* [see curs_inch(3X)].

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that all routines except winwchnstr may be macros.

SEE ALSO

curses(3X), curs_inwch(3X).

NAME

curs_inwstr: inwstr, innwstr, winwstr, winnwstr, mvinwstr, mvinnwstr, mvwinwstr, mvwinnwstr – get a string of wchar_t characters from a curses window

SYNOPSIS

```
#include <curses.h>

int inwstr(char *str);

int innwstr(char *str, int n);

int winwstr(WINDOW *win, char *str);

int winnwstr(WINDOW *win, char *str, int n);

int mvinwstr(int y, int x, char *str);

int mvinnwstr(int y, int x, char *str, int n);

int mvwinwstr(WINDOW *win, int y, int x, char *str);

int mvwinnwstr(WINDOW *win, int y, int x, char *str, int n);
```

DESCRIPTION

These routines return a string of wchar_t characters in *str*, starting at the current cursor position in the named window and ending at the right margin of the window. Attributes are stripped from the characters. The four functions with *n* as the last argument return the string at most *n* wchar_t characters long.

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that all routines except winnwstr may be macros.

SEE ALSO

curses(3X).

NAME

curs_kernel: def_prog_mode, def_shell_mode, reset_prog_mode, reset_shell_mode, resetty, savetty, getsyx, setsyx, ripoffline, curs_set, napms – low-level curses routines

SYNOPSIS

```
#include <curses.h>

int def_prog_mode(void);

int def_shell_mode(void);

int reset_prog_mode(void);

int reset_shell_mode(void);

int resetty(void);

int savetty(void);

int getsyx(int y, int x);

int setsyx(int y, int x);

int ripoffline(int line, int (*init)(WINDOW *, int));

int curs_set(int visibility);

int napms(int ms);
```

DESCRIPTION

The following routines give low-level access to various curses functionality. Theses routines typically are used inside library routines.

The def_prog_mode and def_shell_mode routines 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.

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

The resetty and savetty routines save and restore the state of the terminal modes. savetty saves the current state in a buffer and resetty restores the state to what it was at the last call to savetty.

With the getsyx routine, the current coordinates of the virtual screen cursor are returned in *y* and *x*. If leaveok is currently TRUE, then -1,-1 is returned. If lines have been removed from the top of the screen, using ripoffline, *y* and *x* include these lines; therefore, *y* and *x* should be used only as arguments for setsyx.

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

The ripoffline routine provides access to the same facility that slk_init [see curs_slk(3X)] uses to reduce the size of the screen. ripoffline must be called before initscr or newterm is called. If *line* is positive, a line is removed from the top of stdscr; if *line* is negative, a line is removed from the bottom. When this is done inside initscr, the routine init (supplied by the user) is called with two arguments: a window pointer to the one-line window that has been allocated and an integer with the number of columns in the window. Inside this initialization routine, the integer variables LINES and COLS (defined in <curses.h>) are not guaranteed to be accurate and wrefresh or doupdate must not be called. It is allowable to call wnoutrefresh during the initialization routine.

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

With the curs_set routine, the cursor state is set to invisible, normal, or very visible for visibility equal to 0, 1, or 2 respectively. If the terminal supports the *visibility* requested, the previous *cursor* state is returned; otherwise, ERR is returned.

The napms routine is used to sleep for *ms* milliseconds.

RETURN VALUE

Except for curs_set, these routines always return OK. curs_set returns the previous cursor state, or ERR if the requested *visibility* is not supported.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that getsyx is a macro, so & is not necessary before the variables *y* and *x*.

SEE ALSO

curses(3X), curs_initscr(3X), curs_outopts(3X), curs_refresh(3X), curs_scr_dump(3X), curs_slk(3X).

NAME

curs_move: move, wmove – move curses window cursor

SYNOPSIS

```
#include <curses.h>

int move(int y, int x);

int wmove(WINDOW *win, int y, int x);
```

DESCRIPTION

With these routines, the cursor associated with the window is moved to line *y* and column *x*. This routine does not move the physical cursor of the terminal until refresh is called. The position specified is relative to the upper left-hand corner of the window, which is (0,0).

RETURN VALUE

These routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that move may be a macro.

SEE ALSO

curses(3X), curs_refresh(3X).

NAME

curs_outopts: clearok, idlok, idcok immedok, leaveok, setscrreg, wsetscrreg, scrollok, nl, nonl - curses terminal output option control routines

SYNOPSIS

```
#include <curses.h>

int clearok(WINDOW *win, bool bf);

int idlok(WINDOW *win, bool bf);

void idcok(WINDOW *win, bool bf);

void immedok(WINDOW *win, bool bf);

int leaveok(WINDOW *win, bool bf);

int setscrreg(int top, int bot);

int wsetscrreg(WINDOW *win, int top, int bot);

int scrollok(WINDOW *win, bool bf);

int nl(void);

int nonl(void);
```

DESCRIPTION

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

With the clearok routine, 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. If the *win* argument to clearok is the global variable curscr, the next call to wrefresh with any window causes the screen to be cleared and repainted from scratch.

With the idlok routine, if enabled (*bf* is TRUE), curses considers using the hardware insert/delete line feature of terminals so equipped. If disabled (*bf* is FALSE), curses very seldom uses this feature. (The insert/delete character feature is always considered.) This option should be enabled only if the application needs insert/delete line, for example, for a screen editor. It is disabled by default because insert/delete line tends to be visually annoying when used in applications where it isn't really needed. If insert/delete line cannot be used, curses redraws the changed portions of all lines.

With the idcok routine, if enabled (*bf* is TRUE), curses considers using the hardware insert/delete character feature of terminals so equipped. This is enabled by default.

With the immedok routine, if enabled (*bf* is TRUE), any change in the window image, such as the ones caused by waddch, wclrtobot, wscrl, *etc.*, automatically cause a call to wrefresh. However, it may degrade the performance considerably, due to repeated calls to wrefresh. It is disabled by default.

Normally, the hardware cursor is left at the location of the window cursor being refreshed. The leaveok 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.

The setscrreg and wsetscrreg routines allow the application programmer to set a software scrolling region in a window. *top* and *bot* are the line numbers of the top and bottom margin of the scrolling region. (Line 0 is the top line of the window.) If this option and scrollok are enabled, an attempt to move off the bottom margin line causes all lines in the scrolling region to scroll up one line. Only the text of the window is scrolled. (Note that this has nothing to do with the use of a physical scrolling region capability in the terminal, like that in the VT100. 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.)

The scrollok option controls what happens when the cursor of a window is moved off the edge of the window or scrolling region, either as a result of a newline action 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. If enabled, (*bf* is TRUE), wrefresh is called on the window, and the physical terminal and window are scrolled up one line. [Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call idlok.]

The nl and nonl routines control whether newline is translated into carriage return and linefeed on output, and whether return is translated into newline on input. Initially, the translations do occur. By disabling these translations using nonl, curses is able to make better use of the linefeed capability, resulting in faster cursor motion.

RETURN VALUE

setscrreg and wsetscrreg return OK upon success and ERR upon failure. All other routines that return an integer always return OK.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that clearok, leaveok, scrollok, idcok, nl, nonl and setscrreg may be macros.

The immedok routine is useful for windows that are used as terminal emulators.

SEE ALSO

curses(3X), curs_addch(3X), curs_clear(3X), curs_initscr(3X), curs_scroll(3X), curs_refresh(3X).

NAME

curs_overlay: overlay, overwrite, copywin – overlap and manipulate overlapped curses windows

SYNOPSIS

```
#include <curses.h>

int overlay(WINDOW *srcwin, WINDOW *dstwin);

int overwrite(WINDOW *srcwin, WINDOW *dstwin);

int copywin(WINDOW *srcwin, WINDOW *dstwin, int sminrow,
      int smincol, int dminrow, int dmincol, int dmaxrow,
      int dmaxcol, int overlay);
```

DESCRIPTION

The overlay and overwrite routines overlay *srcwin* on top of *dstwin*. *scrwin* and *dstwin* are not required to 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) whereas overwrite is destructive.

The copywin routine provides a finer granularity 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.

RETURN VALUE

Routines that return an integer return ERR upon failure and an integer value other than ERR upon successful completion.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that overlay and overwrite may be macros.

SEE ALSO

curses(3X), curs_pad(3X), curs_refresh(3X).

NAME

curs_pad: newpad, subpad, prefresh, pnoutrefresh, pechochar, pechowchar – create and display curses pads

SYNOPSIS

```
#include <curses.h>

WINDOW *newpad(int nlines, int ncols);

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

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

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

int pechochar(WINDOW *pad, chtype ch);

int pechowchar(WINDOW *pad, chtype wch);
```

DESCRIPTION

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

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

The prefresh and pnoutrefresh routines are analogous to wrefresh and wnoutrefresh except that they relate to pads instead of windows. The additional parameters are needed to indicate what part of the pad and screen are involved. *pminrow* and *pmincol* specify the upper left-hand corner of the rectangle to be displayed in the pad. *sminrow*, *smincol*, *smaxrow*, and *smaxcol* specify the edges of the rectangle to be displayed on the screen. The lower right-hand corner of the rectangle to be displayed in the pad 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.

The pechochar routine is functionally equivalent to a call to addch followed by a call to refresh, a call to waddch followed by a call to wrefresh, or a call to waddch followed by a call to prefresh. The knowledge that only a single character is being output is taken into consideration and, for non-control characters, a considerable performance gain might 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.

The pechowchar routine is functionally equivalent to a call to addwch followed by a call to refresh, a call to waddwch followed by a call to wrefresh, or a call to waddwch followed by a call to prefresh.

RETURN VALUE

Routines that return an integer return ERR upon failure and an integer value other than ERR upon successful completion.

Routines that return pointers return NULL on error.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that pechochar may be a macro.

SEE ALSO

curses(3X), curs_refresh(3X), curs_touch(3X), curs_addch(3X), curs_addwch(3X).

NAME

curs_printw: printw, wprintw, mvprintw, mvwprintw, vwprintw – print formatted output in curses windows

SYNOPSIS

```
#include <curses.h>

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

int wprintw(WINDOW *win, char *fmt [, arg] ...);

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

int mvwprintw(WINDOW *win, int y, int x,
      char *fmt [, arg] ...);

#include <varargs.h>

int vwprintw(WINDOW *win, char *fmt, varglist);
```

DESCRIPTION

The printw, wprintw, mvprintw and mvwprintw routines are analogous to printf [see printf(3S)]. In effect, the string that would be output by printf is output instead as though waddstr were used on the given window.

The vwprintw routine is analogous to vprintf [see vprintf(3S)] and 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>.

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

SEE ALSO

curses(3X), printf(3S), printf(3W), vprintf(3S).

NAME

curs_refresh: refresh, wrefresh, wnoutrefresh, doupdate, redrawwin, wredrawln – refresh curses windows and lines

SYNOPSIS

```
#include <curses.h>

int refresh(void);

int wrefresh(WINDOW *win);

int wnoutrefresh(WINDOW *win);

int doupdate(void);

int redrawwin(WINDOW *win);

int wredrawln(WINDOW *win, int beg_line, int num_lines);
```

DESCRIPTION

The refresh and wrefresh routines (or wnoutrefresh and doupdate) must be called to get any output on the terminal, as other routines merely manipulate data structures. The routine wrefresh copies the named window to the physical terminal screen, taking into account what is already there in order to do optimizations. The refresh routine is the same, using stdscr as the default window. Unless leaveok has been enabled, the physical cursor of the terminal is left at the location of the cursor for that window.

The wnoutrefresh and doupdate routines allow multiple updates with more efficiency than wrefresh alone. In addition to all the window structures, curses keeps two data structures representing the terminal screen: a physical screen, describing what is actually on the screen, and a virtual screen, describing what the programmer wants to have on the screen.

The routine wrefresh works by first calling wnoutrefresh, which copies the named window to the virtual screen, and then calling doupdate, which compares the virtual screen to the physical screen and does the actual update. If the programmer wishes to output several windows at once, a series of calls to wrefresh results in alternating calls to wnoutrefresh and doupdate, causing several bursts of output to the screen. By first calling wnoutrefresh for each window, it is then possible to call doupdate once, resulting in only one burst of output, with fewer total characters transmitted and less CPU time used. If the *win* argument to wrefresh is the global variable curscr, the screen is immediately cleared and repainted from scratch.

The redrawwin routine indicates to curses that some screen lines are corrupted and should be thrown away before anything is written over them. These routines could be used for programs such as editors, which want a command to redraw some part of the screen or the entire screen. The routine redrawln is preferred over redrawwin where a noisy communication line exists and redrawing the entire window could be subject to even more communication noise. Just redrawing several lines offers the possibility that they would show up unblemished.

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that refresh and redrawwin may be macros.

SEE ALSO

curses(3X), curs_outopts(3X).

NAME

curs_scanw: scanw, wscanw, mvscanw, mvwscanw, vwscanw – convert formatted input from a curses widow

SYNOPSIS

```
#include <curses.h>

int scanw(char *fmt [, arg] ...);

int wscanw(WINDOW *win, char *fmt [, arg] ...);

int mvscanw(int y, int x, char *fmt [, arg] ...);

int mvwscanw(WINDOW *win, int y, int x,
      char *fmt [, arg] ...);

int vwscanw(WINDOW *win, char *fmt, va_list varglist);
```

DESCRIPTION

The scanw, wscanw and mvscanw routines correspond to scanf [see scanf(3S)]. The effect of these routines is as though wgetstr were called on the window, and the resulting line used as input for the scan. Fields which do not map to a variable in the *fmt* field are lost.

The vwscanw routine is similar to vwprintw in that it performs a wscanw using a variable argument list. The third argument is a *va_list*, a pointer to a list of arguments, as defined in <varargs.h>.

RETURN VALUE

vwscanw returns ERR on failure and an integer equal to the number of fields scanned on success.

Applications may interrogate the return value from the scanw, wscanw, mvscanw and mvwscanw routines to determine the number of fields which were mapped in the call.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

SEE ALSO

curses(3X), curs_getstr(3X), curs_printw(3X), scanf(3S), scanf(3W).

NAME

curs_scr_dump: scr_dump, scr_restore, scr_init, scr_set – read (write) a curses screen from (to) a file

SYNOPSIS

```
#include <curses.h>

int scr_dump(char *filename);

int scr_restore(char *filename);

int scr_init(char *filename);

int scr_set(char *filename);
```

DESCRIPTION

With the scr_dump routine, the current contents of the virtual screen are written to the file *filename*.

With the scr_restore routine, the virtual screen is set to the contents of *filename*, which must have been written using scr_dump. The next call to doupdate restores the screen to the way it looked in the dump file.

With the scr_init routine, the contents of *filename* are read in and used to initialize the curses data structures about what the terminal currently has on its screen. If the data is determined to be valid, curses bases its next update of the screen on this information rather than clearing the screen and starting from scratch. scr_init is used after initscr or a system [see system(BA_LIB)] call to share the screen with another process which has done a scr_dump after its endwin call. The data is declared invalid if the time-stamp of the tty is old or the terminfo capabilities rmcup and nrrmc exist.

The scr_set routine is a combination of scr_restore and scr_init. It tells the program that the information in *filename* is what is currently on the screen, and also what the program wants on the screen. This can be thought of as a screen inheritance function.

To read (write) a window from (to) a file, use the getwin and putwin routines [see curs_util(3X)].

RETURN VALUE

All routines return the integer ERR upon failure and OK upon success.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that scr_init, scr_set, and scr_restore may be macros.

SEE ALSO

curses(3X), curs_initscr(3X), curs_refresh(3X), curs_util(3X), system(3S).

NAME

curs_scroll: scroll, srcl, wscrl – scroll a curses window

SYNOPSIS

```
#include <curses.h>

int scroll(WINDOW *win);

int scrl(int n);

int wscrl(WINDOW *win, int n);
```

DESCRIPTION

With the scroll routine, the window is scrolled up one line. This involves moving the lines in the window data structure. As an optimization, if the scrolling region of the window is the entire screen, the physical screen is scrolled at the same time.

With the scrl and wscrl routines, for positive *n* scroll the window up *n* lines (line *i+n* becomes *i*); otherwise scroll the window down *n* lines. This involves moving the lines in the window character image structure. The current cursor position is not changed.

For these functions to work, scrolling must be enabled via scrollok.

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that scrl and scroll may be macros.

SEE ALSO

curses(3X), curs_outopts(3X).

NAME

curs_slk: slk_init, slk_set, slk_refresh, slk_noutrefresh, slk_label, slk_clear, slk_restore, slk_touch, slk_attron, slk_attrset, slk_attroff – curses soft label routines

SYNOPSIS

```
#include <curses.h>

int slk_init(int fmt);

int slk_set(int labnum, char *label, int fmt);

int slk_refresh(void);

int slk_noutrefresh(void);

char *slk_label(int labnum);

int slk_clear(void);

int slk_restore(void);

int slk_touch(void);

int slk_attron(chtype attrs);

int slk_attrset(chtype attrs);

int slk_attroff(chtype attrs);
```

DESCRIPTION

curses manipulates the set of soft function-key labels that exist on many terminals. For those terminals that do not have soft labels, curses takes over the bottom line of stdscr, reducing the size of stdscr and the variable LINES. curses standardizes on eight labels of up to eight characters each.

To use soft labels, the slk_init routine must be called before initscr or newterm is called. If initscr eventually uses a line from stdscr to emulate the soft labels, then *fmt* determines how the labels are arranged on the screen. Setting *fmt* to 0 indicates a 3-2-3 arrangement of the labels; 1 indicates a 4-4 arrangement.

With the slk_set routine, *labnum* is the label number, from 1 to 8. *label* is the string to be put on the label, up to eight characters in length. A null string or a null pointer sets up a blank label. *fmt* is either 0, 1, or 2, indicating whether the label is to be left-justified, centered, or right-justified, respectively, within the label.

The slk_refresh and slk_noutrefresh routines correspond to the wrefresh and wnoutrefresh routines.

With the slk_label routine, the current label for label number *labnum* is returned with leading and trailing blanks stripped.

With the slk_clear routine, the soft labels are cleared from the screen.

With the slk_restore routine, the soft labels are restored to tne screen after a slk_clear is performed.

With the slk_touch routine, all the soft labels are forced to be output the next time a slk_noutrefresh is performed.

The slk_attron, slk_attrset and slk_attroff routines correspond to attron, attrset, and attroff. They have an effect only if soft labels are simulated on the bottom line of the screen.

RETURN VALUE

Routines that return an integer return ERR upon failure and an integer value other than ERR upon successful completion.

`slk_label` returns NULL on error.

NOTES

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Most applications would use `slk_noutrefresh` because a `wrefresh` is likely to follow soon.

SEE ALSO

`curses`(3X), `curs_attr`(3X), `curs_initscr`(3X), `curs_refresh`(3X).

NAME

curs_termattrs: baudrate, erasechar, has_ic, has_il, killchar, longname, termattrs, termname – curses environment query routines

SYNOPSIS

```
#include <curses.h>

int baudrate(void);

char erasechar(void);

int has_ic(void);

int has_il(void);

char killchar(void);

char *longname(void);

chtype termattrs(void);

char *termname(void);
```

DESCRIPTION

The baudrate routine returns the output speed of the terminal. The number returned is in bits per second, for example 9600, and is an integer.

With the erasechar routine, the user's current erase character is returned.

The has_ic routine is true if the terminal has insert- and delete-character capabilities.

The has_il routine is true if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This might be used to determine if it would be appropriate to turn on physical scrolling using scrollok.

With the killchar routine, the user's current line kill character is returned.

The longname 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.

If a given terminal doesn't support a video attribute that an application program is trying to use, curses may substitute a different video attribute for it. The termattrs function returns a logical OR of all video attributes supported by the terminal. This information is useful when a curses program needs complete control over the appearance of the screen.

The termname routine returns the value of the environmental variable TERM (truncated to 14 characters).

RETURN VALUE

longname and termname return NULL on error.

Routines that return an integer return ERR upon failure and an integer value other than ERR upon successful completion.

NOTES

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

Note that `termattrs` may be a macro.

SEE ALSO

`curses`(3X), `curs_initscr`(3X), `curs_outopts`(3X).

NAME

curs_termcap: tgetent, tgetflag, tgetnum, tgetstr, tgoto, tputs – curses interfaces (emulated) to the termcap library

SYNOPSIS

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

int tgetent(char *bp, char *name);

int tgetflag(char id[2]);

int tgetnum(char id[2]);

char *tgetstr(char id[2], char **area);

char *tgoto(char *cap, int col, int row);

int tputs(char *str, int affcnt, int (*putc)(void));
```

DESCRIPTION

These routines are included as a conversion aid for programs that use the *termcap* library. Their parameters are the same and the routines are emulated using the *terminfo* database. These routines are supported at Level 2 and should not be used in new applications.

The tgetent routine looks up the termcap entry for *name*. The emulation ignores the buffer pointer *bp*.

The tgetflag routine gets the boolean entry for *id*.

The tgetnum routine gets the numeric entry for *id*.

The tgetstr routine returns the string entry for *id*. Use tputs to output the returned string.

The tgoto routine instantiates the parameters into the given capability. The output from this routine is to be passed to tputs.

The tputs routine is described on the curs_terminfo(4) manual page.

RETURN VALUE

Routines that return an integer return ERR upon failure and an integer value other than ERR upon successful completion.

Routines that return pointers return NULL on error.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

SEE ALSO

curses(3X), curs_terminfo(4), putc(3S).

NAME

curs_terminfo: setupterm, setterm, set_curterm, del_curterm, restartterm, tparm, tputs, putp, vidputs, vidattr, mvcur, tigetflag, tigetnum, tigetstr – curses interfaces to terminfo database

SYNOPSIS

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

int setupterm(char *term, int fildes, int *errret);

int setterm(char *term);

int set_curterm(TERMINAL *nterm);

int del_curterm(TERMINAL *oterm);

int restartterm(char *term, int fildes, int *errret);

char *tparm(char *str, long int p1, long int p2, long int p3,
      long int p4, long int p5, long int p6, long int p7,
      long int p8, long int p9);

int tputs(char *str, int affcnt, int (*putc)(char));

int putp(char *str);

int vidputs(chtype attrs, int (*putc)(char));

int vidattr(chtype attrs);

int mvcur(int oldrow, int oldcol, int newrow, int newcol);

int tigetflag(char *capname);

int tigetnum(char *capname);

int tigetstr(char *capname);
```

DESCRIPTION

These low-level routines must be called by programs that have to deal directly with the *terminfo* database to handle certain terminal capabilities, such as programming function keys. For all other functionality, curses routines are more suitable and their use is recommended.

Initially, setupterm should be called. Note that setupterm is automatically called by initscr and newterm. This defines the set of terminal-dependent variables [listed in terminfo(4)]. The *terminfo* variables lines and columns are initialized by setupterm as follows: If use_env(FALSE) has been called, values for lines and columns specified in *terminfo* are used. Otherwise, if the environment variables LINES and COLUMNS exist, their values are used. If these environment variables do not exist and the program is running in a window, the current window size is used. Otherwise, if the environment variables do not exist, the values for lines and columns specified in the *terminfo* 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* strings [including the output of tparm] should be printed with tputs or putp. Call the reset_shell_mode to restore the tty modes before exiting [see curs_kernel(3X)]. Programs which use cursor addressing should output enter_ca_mode upon startup and should output exit_ca_mode before exiting. 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.

The setupterm routine reads in the *terminfo* database, initializing the *terminfo* structures, but does not set up the output virtualization structures used by curses. The terminal type is the character string *term;* if *term* is null, the environment variable TERM is used. All output is to file descriptor fildes which is initialized for output. If *errret* is not null, then setupterm returns OK or ERR and stores a status value in the integer pointed to by *errret*. A status of 1 in *errret* is normal, 0 means that the terminal could not be found, and -1 means that the *terminfo* database could not be found. If *errret* is null, setupterm prints an error message upon finding an error and exits. Thus, the simplest call is:

```
        setupterm((char *)0, 1, (int *)0);,
```

which uses all the defaults and sends the output to stdout.

The setterm routine is being replaced by setupterm. The call:

```
        setupterm(term, 1, (int *)0)
```

provides the same functionality as setterm(*term*). The setterm routine is included here for compatibility and is supported at Level 2.

The set_curterm routine sets the variable cur_term to *nterm*, and makes all of the *terminfo* boolean, numeric, and string variables use the values from *nterm*.

The del_curterm routine frees the space pointed to by *oterm* and makes it available for further use. If *oterm* is the same as cur_term, references to any of the *terminfo* boolean, numeric, and string variables thereafter may refer to invalid memory locations until another setupterm has been called.

The restartterm routine is similar to setupterm and initscr, except that it is called after restoring memory to a previous state. It assumes that the windows and the input and output options are the same as when memory was saved, but the terminal type and baud rate may be different.

The tparm routine instantiates the string *str* with parameters *pi*. A pointer is returned to the result of *str* with the parameters applied.

The tputs routine applies padding information to the string *str* and outputs it. The *str* must be a terminfo string variable or the return value from tparm, tgetstr, or tgoto. *affcnt* is the number of lines affected, or 1 if not applicable. *putc* is a putchar-like routine to which the characters are passed, one at a time.

The putp routine calls tputs(*str*, 1, putchar). Note that the output of putp always goes to stdout, not to the *fildes* specified in setupterm.

The vidputs routine displays the string on the terminal in the video attribute mode *attrs*, which is any combination of the attributes listed in curses(3X). The characters are passed to the putchar-like routine *putc*.

The vidattr routine is like the vidputs routine, except that it outputs through putchar.

The mvcur routine provides low-level cursor motion.

The tigetflag, tigetnum and tigetstr routines return the value of the capability corresponding to the *terminfo capname* passed to them, such as xenl.

With the `tigetflag` routine, the value `-1` is returned if *capname* is not a boolean capability.

With the `tigetnum` routine, the value `-2` is returned if *capname* is not a numeric capability.

With the `tigetstr` routine, the value `(char *)-1` is returned if *capname* is not a string capability.

The *capname* for each capability is given in the table column entitled *capname* code in the capabilities section of `terminfo(4)`.

`char *boolnames, *boolcodes, *boolfnames`

`char *numnames, *numcodes, *numfnames`

`char *strnames, *strcodes, *strfnames`

These null-terminated arrays contain the *capnames*, the *termcap* codes, and the full C names, for each of the *terminfo* variables.

RETURN VALUE

All routines return the integer `ERR` upon failure and an integer value other than `ERR` upon successful completion, unless otherwise noted in the preceding routine descriptions.

Routines that return pointers always return `NULL` on error.

NOTES

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

The `setupterm` routine should be used in place of `setterm`.

Note that `vidattr` and `vidputs` may be macros.

SEE ALSO

`curses(3X)`, `curs_initscr(3X)`, `curs_kernel(3X)`, `curs_termcap(3X)`, `putc(3S)`, `terminfo(4)`.

NAME

curs_touch: touchwin, touchline, untouchwin, wtouchln, is_linetouched, is_wintouched – curses refresh control routines

SYNOPSIS

```
#include <curses.h>

int touchwin(WINDOW *win);

int touchline(WINDOW *win, int start, int count);

int untouchwin(WINDOW *win);

int wtouchln(WINDOW *win, int y, int n, int changed);

int is_linetouched(WINDOW *win, int line);

int is_wintouched(WINDOW *win);
```

DESCRIPTION

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

The untouchwin routine marks all lines in the window as unchanged since the last call to wrefresh.

The wtouchln routine makes *n* lines in the window, starting at line *y*, look as if they have (*changed*=1) or have not (*changed*=0) been changed since the last call to wrefresh.

The is_linetouched and is_wintouched routines return TRUE if the specified line/window was modified since the last call to wrefresh; otherwise they return FALSE. In addition, is_linetouched returns ERR if *line* is not valid for the given window.

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion, unless otherwise noted in the preceding routine descriptions.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that all routines except wtouchln may be macros.

SEE ALSO

curses(3X), curs_refresh(3X).

NAME

curs_util: unctrl, keyname, filter, use_env, putwin, getwin, delay_output, flushinp – miscellaneous curses utility routines

SYNOPSIS

```
#include <curses.h>

char *unctrl(chtype c);

char *keyname(int c);

int filter(void);

void use_env(char bool);

int putwin(WINDOW *win, FILE *filep);

WINDOW *getwin(FILE *filep);

int delay_output(int ms);

int flushinp(void);
```

DESCRIPTION

The unctrl 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.

With the keyname routine, a character string corresponding to the key *c* is returned.

The filter routine, if used, is called before initscr or newterm are called. It makes curses think that there is a one-line screen. curses does not use any terminal capabilities that assume that they know on what line of the screen the cursor is positioned.

The use_env routine, if used, is called before initscr or newterm are called. When called with FALSE as an argument, the values of lines and columns specified in the *terminfo* database will be used, even if environment variables LINES and COLUMNS (used by default) are set, or if curses is running in a window (in which case default behavior would be to use the window size if LINES and COLUMNS are not set).

With the putwin routine, all data associated with window *win* is written into the file to which *filep* points. This information can be later retrieved using the getwin function.

The getwin routine reads window related data stored in the file by putwin. The routine then creates and initializes a new window using that data. It returns a pointer to the new window.

The delay_output routine inserts an *ms* millisecond pause in output. This routine should not be used extensively because padding characters are used rather than a CPU pause.

The flushinp routine throws away any typeahead that has been typed by the user and has not yet been read by the program.

RETURN VALUE

Except for flushinp, routines that return an integer return ERR upon failure and an integer value other than ERR upon successful completion.

flushinp always returns OK.

Routines that return pointers return NULL on error.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

Note that unctrl is a macro, which is defined in <unctrl.h>.

SEE ALSO

curses(3X), curs_initscr(3X), curs_scr_dump(3X).

NAME

curs_window: newwin, delwin, mvwin, subwin, derwin, mvderwin, dupwin, wsyncup, syncok, wcursyncup, wsyncdown – create curses windows

SYNOPSIS

```
#include <curses.h>

WINDOW *newwin(int nlines, int ncols, int begin_y,
      intbegin_x);

int delwin(WINDOW *win);

int mvwin(WINDOW *win, int y, int x);

WINDOW *subwin(WINDOW *orig, int nlines, int ncols,
      int begin_y, int begin_x);

WINDOW *derwin(WINDOW *orig, int nlines, int ncols,
      int begin_y, int begin_x);

int mvderwin(WINDOW *win, int par_y, int par_x);

WINDOW *dupwin(WINDOW *win);

void wsyncup(WINDOW *win);

int syncok(WINDOW *win, bool bf);

void wcursyncup(WINDOW *win);

void wsyncdown(WINDOW *win);
```

DESCRIPTION

The newwin routine creates and returns a pointer to a new window with the given number of lines, *nlines*, and columns, *ncols*. The upper left-hand corner of the window is at line *begin_y*, column *begin_x*. If either *nlines* or *ncols* is zero, they default to LINES *begin_y* and COLS *begin_x*. A new full-screen window is created by calling newwin(0,0,0,0).

The delwin routine deletes the named window, freeing all memory associated with it. Subwindows must be deleted before the main window can be deleted.

The mvwin routine moves the window so that the upper left-hand corner is at position (*x*, *y*). If the move would cause the window to be off the screen, it is an error and the window is not moved. Moving subwindows is allowed, but should be avoided.

The subwin routine creates and returns a pointer to a new window with the given number of lines, *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. The subwindow shares memory with the window *orig*. When using this routine, it is necessary to call touchwin or touchline on *orig* before calling wrefresh on the subwindow.

The derwin routine is the same as subwin, except that *begin_y* and *begin_x* are relative to the origin of the window *orig* rather than the screen. There is no difference between the subwindows and the derived windows.

The mvderwin routine moves a derived window (or subwindow) inside its parent window. The screen-relative parameters of the window are not changed. This routine is used to display different parts of the parent window at the same physical position on the screen.

The dupwin routine creates an exact duplicate of the window *win*.

Each curses window maintains two data structures: the character image structure and the status structure. The character image structure is shared among all windows in the window hierarchy (*i.e.*, the window with all subwindows). The status structure, which contains information about individual line changes in the window, is private to each window. The routine wrefresh uses the status data structure when performing screen updating. Since status structures are not shared, changes made to one window in the hierarchy may not be properly reflected on the screen.

The routine wsyncup causes the changes in the status structure of a window to be reflected in the status structures of its ancestors. If syncok is called with second argument TRUE then wsyncup is called automatically whenever there is a change in the window.

The routine wcursyncup updates the current cursor position of all the ancestors of the window to reflect the current cursor position of the window.

The routine wsyncdown updates the status structure of the window to reflect the changes in the status structures of its ancestors. Applications seldom call this routine because it is called automatically by wrefresh.

RETURN VALUE

Routines that return an integer return the integer ERR upon failure and an integer value other than ERR upon successful completion.

delwin returns the integer ERR upon failure and OK upon successful completion.

Routines that return pointers return NULL on error.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

If many small changes are made to the window, the wsyncup option could degrade performance.

Note that syncok may be a macro.

SEE ALSO

curses(3X), curs_refresh(3X), curs_touch(3X).

NAME

curses – CRT screen handling and optimization package

SYNOPSIS

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

DESCRIPTION

The curses library routines give the user a terminal-independent method of updating character screens with reasonable optimization. A program using these routines must be compiled with the -lcurses option of cc.

The curses package allows: overall screen, window and pad manipulation; output to windows and pads; reading terminal input; control over terminal and curses input and output options; environment query routines; color manipulation; use of soft label keys; terminfo access; and access to low-level curses routines.

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. The routine endwin must be called before exiting. To get character-at-a-time input without echoing (most interactive, screen oriented programs want this), the following sequence should be used:

```
initscr,cbreak,noecho;
```

Most programs would additionally use the sequence:

```
nonl,intrflush(stdscr,FALSE),keypad(stdscr,TRUE);
```

Before a curses program is run, the tab stops of the terminal should be set and its initialization strings, if defined, must be output. This can be done by executing the tput init command after the shell environment variable TERM has been exported. [See terminfo(4) for further details.]

The curses library permits manipulation of data structures, called *windows*, which can be thought of as two-dimensional arrays of characters representing all or part of a CRT screen. A default window called stdscr, which is the size of the terminal screen, is supplied. Others may be created with newwin.

Windows are referred to by variables declared as WINDOW *. These data structures are manipulated with routines described on 3X paages (whose names begin "curs_"). Among which the most basic routines are move and addch. More general versions of these routines are included with names beginning with w, allowing the user to specify a window. The routines not beginning with w affect stdscr.)

After using routines to manipulate a window, refresh is called, telling curses to make the user's CRT screen look like stdscr. The characters in a window are actually of type chtype, (character and attribute data) so that other information about the character may also be stored with each character.

Special windows called *pads* may also be manipulated. These are windows which are not constrained to the size of the screen and whose contents need not be completely displayed. See curs_pad(3X) for more information.

In addition to drawing characters on the screen, video attributes and colors may be included, causing the characters to show up in such modes as underlined, in reverse video, or in color 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.

If the environment variables LINES and COLUMNS are set, or if the program is executing in a window environment, line and column information in the environment will override information read by *terminfo*. This would effect a program running in an AT&T 630 layer, for example, where the size of a screen is changeable.

If the environment variable TERMINFO is defined, any program using curses checks for a local terminal definition before checking in the standard place. For example, if TERM is set to att4424, then the compiled terminal definition is found in

```
        /usr/share/lib/terminfo/a/att4424.
```

(The a is copied from the first letter of att4424 to avoid creation of huge directories.) However, if TERMINFO is set to $HOME/myterms, curses first checks

```
        $HOME/myterms/a/att4424,
```

and if that fails, it then checks

```
        /usr/share/lib/terminfo/a/att4424.
```

This is useful for developing experimental definitions or when write permission in /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. The constants TRUE and FALSE have the values 1 and 0, respectively.

The curses routines also define the WINDOW * variable curscr which is used for certain low-level operations like clearing and redrawing a screen containing garbage. The curscr can be used in only a few routines.

International Functions

In addition to the standard curses library, the curses32 library is available for handling multibyte characters in 32 bit process code. The curses32 library provides additional functions, as well as enhancements to existing curses library functions. This section describes characteristics of the curses32 library. A program using the curses32 library functions must be compiled with the -lcurses32 option of cc.

The number of bytes and the number of columns to hold a character from a supplementary character set is locale-specific (locale category LC_CTYPE) and can be specified in the character class table.

For editing, operating at the character level is entirely appropriate. For screen formatting, arbitrary movement of characters on the screen is not desirable.

Overwriting characters (for example, addch) operates on a screen level. Overwriting a character by a character which requires a different number of columns may produce *orphaned columns*. These orphaned columns are filled with the background character.

Inserting characters (for example, insch) operates on a character level (that is, at the character boundaries). The specified character is inserted right before the character, regardless of whichever column of a character the cursor points to. Before insertion, the cursor position is adjusted to the first column of the character.

As with inserting characters, deleting characters (for example, delch) operates on a character level (that is, at the character boundaries). The character at the cursor is deleted regardless of whichever columns of the character the cursor points to. Before deletion, the cursor position is adjusted to the first column of the character.

Multi-column characters cannot be put on the last column of lines. When such attempts are made, the last column is set to the background character. In addition, when such an operation creates orphaned columns, the columns are also filled with the background character.

Overlapping and overwriting windows follows the operation of overwriting characters around its edge. The orphaned columns, if any, are handled in the same manner as the character operations.

The cursor is allowed to be placed anywhere in a window. If the insertion or deletion is made when the cursor points to the second or later column position of a character which holds multiple columns, the cursor is adjusted to the first column of it before the insertion or deletion.

Routine and Argument Names

Many curses 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 an *x* and *y* coordinate to move to before performing the appropriate action. The mv routines imply a call to move before the call to the other routine. The coordinate *y* always refers to the row (of the window), and *x* always refers to the column. The upper left-hand corner is always (0,0), not (1,1).

The routines prefixed with mvw take both a window argument and *x* and *y* coordinates. The window argument is always specified before the coordinates.

In each case, *win* is the window affected, and *pad* is the pad affected; *win* and *pad* are always pointers to type WINDOW.

Option setting routines require a Boolean flag *bf* with the value TRUE or FALSE; *bf* is always of type bool. The variables *ch* and *attrs* below are always of type chtype. The types WINDOW, SCREEN, bool, and chtype are defined in <curses.h>. The type TERMINAL is defined in <term.h>. All other arguments are integers.

Routine Name Index

The following table lists each curses routine and the name of the manual page on which it is described. The routines marked with an "*" provide new or enhanced function in the curses32 library.

curses Routine Name	Manual Page Name
addch	curs_addch(3X)
addchnstr	curs_addchstr(3X)
addchstr	curs_addchstr(3X)
addnstr	curs_addstr(3X)
*addnwstr	curs_addwstr(3X)
addstr	curs_addstr(3X)
*addwch	curs_addwch(3X)
*addwchnstr	curs_addwchstr(3X)
*addwchstr	curs_addwchstr(3X)
*addwstr	curs_addwstr(3X)
attroff	curs_attr(3X)
attron	curs_attr(3X)
attrset	curs_attr(3X)
baudrate	curs_termattrs(3X)
beep	curs_beep(3X)
bkgd	curs_bkgd(3X)

bkgdset	curs_bkgd(3X)
border	curs_border(3X)
box	curs_border(3X)
can_change_color	curs_color(3X)
cbreak	curs_inopts(3X)
clear	curs_clear(3X)
clearok	curs_outopts(3X)
clrtobot	curs_clear(3X)
clrtoeol	curs_clear(3X)
color_content	curs_color(3X)
copywin	curs_overlay(3X)
curs_set	curs_kernel(3X)
def_prog_mode	curs_kernel(3X)
def_shell_mode	curs_kernel(3X)
del_curterm	curs_terminfo(4)
delay_output	curs_util(3X)
*delch	curs_delch(3X)
deleteln	curs_deleteln(3X)
delscreen	curs_initscr(3X)
delwin	curs_window(3X)
derwin	curs_window(3X)
doupdate	curs_refresh(3X)
dupwin	curs_window(3X)
echo	curs_inopts(3X)
echochar	curs_addch(3X)
*echowchar	curs_addwch(3X)
endwin	curs_initscr(3X)
erase	curs_clear(3X)
erasechar	curs_termattrs(3X)
filter	curs_util(3X)
flash	curs_beep(3X)
flushinp	curs_util(3X)
getbegyx	curs_getyx(3X)
getch	curs_getch(3X)
getmaxyx	curs_getyx(3X)
*getnstr	curs_getstr(3X)
*getnwstr	curs_getwstr(3X)
getparyx	curs_getyx(3X)
getstr	curs_getstr(3X)
getsyx	curs_kernel(3X)
*getwch	curs_getwch(3X)
getwin	curs_util(3X)
*getwstr	curs_getwstr(3X)
getyx	curs_getyx(3X)
halfdelay	curs_inopts(3X)
has_colors	curs_color(3X)
has_ic	curs_termattrs(3X)
has_il	curs_termattrs(3X)

idcok	curs_outopts(3X)
idlok	curs_outopts(3X)
immedok	curs_outopts(3X)
inch	curs_inch(3X)
inchnstr	curs_inchstr(3X)
inchstr	curs_inchstr(3X)
init_color	curs_color(3X)
init_pair	curs_color(3X)
initscr	curs_initscr(3X)
innstr	curs_instr(3X)
*innwstr	curs_inwstr(3X)
insch	curs_insch(3X)
insdelln	curs_deleteln(3X)
insertln	curs_deleteln(3X)
insnstr	curs_insstr(3X)
*insnwstr	curs_inswstr(3X)
insstr	curs_insstr(3X)
instr	curs_instr(3X)
*inswch	curs_inswch(3X)
*inswstr	curs_inswstr(3X)
intrflush	curs_inopts(3X)
*inwch	curs_inwch(3X)
*inwchnstr	curs_inwchstr(3X)
*inwchstr	curs_inwchstr(3X)
*inwstr	curs_inwstr(3X)
is_linetouched	curs_touch(3X)
is_wintouched	curs_touch(3X)
isendwin	curs_initscr(3X)
keyname	curs_util(3X)
keypad	curs_inopts(3X)
killchar	curs_termattrs(3X)
leaveok	curs_outopts(3X)
longname	curs_termattrs(3X)
meta	curs_inopts(3X)
move	curs_move(3X)
mvaddch	curs_addch(3X)
mvaddchnstr	curs_addchstr(3X)
mvaddchstr	curs_addchstr(3X)
mvaddnstr	curs_addstr(3X)
*mvaddnwstr	curs_addwstr(3X)
mvaddstr	curs_addstr(3X)
*mvaddwch	curs_addwch(3X)
*mvaddwchnstr	curs_addwchstr(3X)
*mvaddwchstr	curs_addwchstr(3X)
*mvaddwstr	curs_addwstr(3X)
mvcur	curs_terminfo(4)
*mvdelch	curs_delch(3X)
mvderwin	curs_window(3X)

mvgetch	curs_getch(3X)
*mvgetnstr	curs_getstr(3X)
*mvgetnwstr	curs_getwstr(3X)
mvgetstr	curs_getstr(3X)
*mvgetwch	curs_getwch(3X)
*mvgetwstr	curs_getwstr(3X)
mvinch	curs_inch(3X)
mvinchnstr	curs_inchstr(3X)
mvinchstr	curs_inchstr(3X)
mvinnstr	curs_instr(3X)
*mvinnwstr	curs_inwstr(3X)
mvinsch	curs_insch(3X)
mvinsnstr	curs_insstr(3X)
*mvinsnwstr	curs_inswstr(3X)
mvinsstr	curs_insstr(3X)
mvinstr	curs_instr(3X)
*mvinswch	curs_inswch(3X)
*mvinswstr	curs_inswstr(3X)
*mvinwch	curs_inwch(3X)
*mvinwchnstr	curs_inwchstr(3X)
*mvinwchstr	curs_inwchstr(3X)
*mvinwstr	curs_inwstr(3X)
*mvprintw	curs_printw(3X)
*mvscanw	curs_scanw(3X)
mvwaddch	curs_addch(3X)
mvwaddchnstr	curs_addchstr(3X)
mvwaddchstr	curs_addchstr(3X)
mvwaddnstr	curs_addstr(3X)
*mvwaddnwstr	curs_addwstr(3X)
mvwaddstr	curs_addstr(3X)
*mvwaddwch	curs_addwch(3X)
*mvwaddwchnstr	curs_addwchstr(3X)
*mvwaddwchstr	curs_addwchstr(3X)
*mvwaddwstr	curs_addwstr(3X)
*mvwdelch	curs_delch(3X)
mvwgetch	curs_getch(3X)
*mvwgetnstr	curs_getstr(3X)
*mvwgetnwstr	curs_getwstr(3X)
mvwgetstr	curs_getstr(3X)
*mvwgetwch	curs_getwch(3X)
*mvwgetwstr	curs_getwstr(3X)
mvwin	curs_window(3X)
mvwinch	curs_inch(3X)
mvwinchnstr	curs_inchstr(3X)
mvwinchstr	curs_inchstr(3X)
mvwinnstr	curs_instr(3X)
*mvwinnwstr	curs_inwstr(3X)
mvwinsch	curs_insch(3X)

mvwinsnstr	curs_insstr(3X)
*mvwinsnwstr	curs_inswstr(3X)
mvwinsstr	curs_insstr(3X)
mvwinstr	curs_instr(3X)
*mvwinswch	curs_inswch(3X)
*mvwinswstr	curs_inswstr(3X)
*mvwinwch	curs_inwch(3X)
*mvwinwchnstr	curs_inwchstr(3X)
*mvwinwchstr	curs_inwchstr(3X)
*mvwinwstr	curs_inwstr(3X)
*mvwprintw	curs_printw(3X)
*mvwscanw	curs_scanw(3X)
napms	curs_kernel(3X)
newpad	curs_pad(3X)
newterm	curs_initscr(3X)
newwin	curs_window(3X)
nl	curs_outopts(3X)
nocbreak	curs_inopts(3X)
nodelay	curs_inopts(3X)
noecho	curs_inopts(3X)
nonl	curs_outopts(3X)
noqiflush	curs_inopts(3X)
noraw	curs_inopts(3X)
notimeout	curs_inopts(3X)
overlay	curs_overlay(3X)
overwrite	curs_overlay(3X)
pair_content	curs_color(3X)
pechochar	curs_pad(3X)
*pechowchar	curs_pad(3X)
pnoutrefresh	curs_pad(3X)
prefresh	curs_pad(3X)
*printw	curs_printw(3X)
putp	curs_terminfo(4)
putwin	curs_util(3X)
qiflush	curs_inopts(3X)
raw	curs_inopts(3X)
redrawwin	curs_refresh(3X)
refresh	curs_refresh(3X)
reset_prog_mode	curs_kernel(3X)
reset_shell_mode	curs_kernel(3X)
resetty	curs_kernel(3X)
restartterm	curs_terminfo(4)
ripoffline	curs_kernel(3X)
savetty	curs_kernel(3X)
*scanw	curs_scanw(3X)
scr_dump	curs_scr_dump(3X)
scr_init	curs_scr_dump(3X)
scr_restore	curs_scr_dump(3X)

scr_set	curs_scr_dump(3X)
scroll	curs_scroll(3X)
scrollok	curs_outopts(3X)
set_curterm	curs_terminfo(4)
set_term	curs_initscr(3X)
setscrreg	curs_outopts(3X)
setsyx	curs_kernel(3X)
setterm	curs_terminfo(4)
setupterm	curs_terminfo(4)
slk_attroff	curs_slk(3X)
slk_attron	curs_slk(3X)
slk_attrset	curs_slk(3X)
slk_clear	curs_slk(3X)
slk_init	curs_slk(3X)
slk_label	curs_slk(3X)
slk_noutrefresh	curs_slk(3X)
slk_refresh	curs_slk(3X)
slk_restore	curs_slk(3X)
slk_set	curs_slk(3X)
slk_touch	curs_slk(3X)
srcl	curs_scroll(3X)
standend	curs_attr(3X)
standout	curs_attr(3X)
start_color	curs_color(3X)
subpad	curs_pad(3X)
subwin	curs_window(3X)
syncok	curs_window(3X)
termattrs	curs_termattrs(3X)
termname	curs_termattrs(3X)
tgetent	curs_termcap(3X)
tgetflag	curs_termcap(3X)
tgetnum	curs_termcap(3X)
tgetstr	curs_termcap(3X)
tgoto	curs_termcap(3X)
tigetflag	curs_terminfo(4)
tigetnum	curs_terminfo(4)
tigetstr	curs_terminfo(4)
timeout	curs_inopts(3X)
touchline	curs_touch(3X)
touchwin	curs_touch(3X)
tparm	curs_terminfo(4)
tputs	curs_termcap(3X)
tputs	curs_terminfo(4)
typeahead	curs_inopts(3X)
unctrl	curs_util(3X)
ungetch	curs_getch(3X)
*ungetwch	curs_getwch(3X)
untouchwin	curs_touch(3X)

use_env	curs_util(3X)
vidattr	curs_terminfo(4)
vidputs	curs_terminfo(4)
vwprintw	curs_printw(3X)
vwscanw	curs_scanw(3X)
waddch	curs_addch(3X)
waddchnstr	curs_addchstr(3X)
waddchstr	curs_addchstr(3X)
waddnstr	curs_addstr(3X)
*waddnwstr	curs_addwstr(3X)
waddstr	curs_addstr(3X)
*waddwch	curs_addwch(3X)
*waddwchnstr	curs_addwchstr(3X)
*waddwchstr	curs_addwchstr(3X)
*waddwstr	curs_addwstr(3X)
wattroff	curs_attr(3X)
wattron	curs_attr(3X)
wattrset	curs_attr(3X)
wbkgd	curs_bkgd(3X)
wbkgdset	curs_bkgd(3X)
wborder	curs_border(3X)
wclear	curs_clear(3X)
wclrtobot	curs_clear(3X)
wclrtoeol	curs_clear(3X)
wcursyncup	curs_window(3X)
*wdelch	curs_delch(3X)
wdeleteln	curs_deleteln(3X)
wechochar	curs_addch(3X)
*wechowchar	curs_addwch(3X)
werase	curs_clear(3X)
wgetch	curs_getch(3X)
wgetnstr	curs_getstr(3X)
*wgetnwstr	curs_getwstr(3X)
wgetstr	curs_getstr(3X)
*wgetwch	curs_getwch(3X)
*wgetwstr	curs_getwstr(3X)
whline	curs_border(3X)
winch	curs_inch(3X)
winchnstr	curs_inchstr(3X)
winchstr	curs_inchstr(3X)
winnstr	curs_instr(3X)
*winnwstr	curs_inwstr(3X)
winsch	curs_insch(3X)
winsdelln	curs_deleteln(3X)
winsertln	curs_deleteln(3X)
winsnstr	curs_insstr(3X)
*winsnwstr	curs_inswstr(3X)
winsstr	curs_insstr(3X)

winstr	curs_instr(3X)
*winswch	curs_inswch(3X)
*winswstr	curs_inswstr(3X)
*winwch	curs_inwch(3X)
*winwchnstr	curs_inwchstr(3X)
*winwchstr	curs_inwchstr(3X)
*winwstr	curs_inwstr(3X)
wmove	curs_move(3X)
wnoutrefresh	curs_refresh(3X)
*wprintw	curs_printw(3X)
wredrawln	curs_refresh(3X)
wrefresh	curs_refresh(3X)
*wscanw	curs_scanw(3X)
wscrl	curs_scroll(3X)
wsetscrreg	curs_outopts(3X)
wstandend	curs_attr(3X)
wstandout	curs_attr(3X)
wsyncdown	curs_window(3X)
wsyncup	curs_window(3X)
wtimeout	curs_inopts(3X)
wtouchln	curs_touch(3X)
wvline	curs_border(3X)

RETURN VALUE

Routines that return an integer return ERR upon failure and an integer value other than ERR upon successful completion, unless otherwise noted in the routine descriptions.

All macros return the value of the w version, except setscrreg, wsetscrreg, getyx, getbegyx, getmaxyx. The return values of setscrreg, wsetscrreg, getyx, getbegyx, and getmaxyx are undefined (that is, these should not be used as the right-hand side of assignment statements).

Routines that return pointers return NULL on error.

SEE ALSO

terminfo(4) and 3X pages whose names begin "curs_" for detailed routine descriptions.
curs_addch(3X), curs_addchstr(3X), curs_addstr(3X), curs_attr(3X), curs_beep(3X), curs_bkgd(3X), curs_border(3X), curs_clear(3X), curs_color(3X), curs_delch(3X), curs_deleteln(3X), curs_getch(3X), curs_getyx(3X), curs_inch(3X), curs_inchstr(3X), curs_initscr(3X), curs_inopts(3X), curs_insch(3X), curs_insstr(3X), curs_instr(3X), curs_kernel(3X), curs_move(3X), curs_outopts(3X), curs_overlay(3X), curs_refresh(3X), curs_scr_dmp(3X), curs_scroll(3X), curs_slk(3X), curs_termattr(3X), curs_termcap(3X), curs_terminfo(3X), curs_touch(3X), curs_util(3X), curs_window(3X) in the *System V Release 4.0 Programmer's Guide: Character User Interface*.

NOTES

The header file <curses.h> automatically includes the header files <stdio.h> and <unctrl.h>.

NAME

cuserid – get character login name or user name associated with effective UID

SYNOPSIS

```
#include <stdio.h>

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

DESCRIPTION

cuserid generates either

- a character-string representation of the login name that the owner of the current process is logged in under (the default).

  or

- a character-string representation for the user name associated with the effective user ID of the process (the POSIX 1003.1-1988 defined behavior obtained at link time with -lposix).

If *s* is a NULL pointer, this representation is generated in an internal static area, the address of which is returned. Otherwise, *s* is assumed to point to an array of at least L_cuserid characters; the representation is left in this array. The constant L_cuserid is defined in the stdio.h header file.

DIAGNOSTICS

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

SEE ALSO

getlogin(3C), getpwent(3C).

NAME

`dbminit, 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`(3C), 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`.

*Key*s and *content*s are described by the *datum* typedef. A *datum* specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The 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))
```

DIAGNOSTICS

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

SEE ALSO

ndbm(3C).

NOTES

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

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

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 1024 bytes). Moreover all key/content pairs that hash together must fit on a single block. *Store* 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.

NAME

dg_flock – apply or remove an advisory lock on an open DG/UX file

SYNOPSIS

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

#define     LOCK_SH     1     /* shared lock */
#define     LOCK_EX     2     /* exclusive lock */
#define     LOCK_NB     4     /* don't block when locking */
#define     LOCK_UN     8     /* unlock */

dg_flock(fildes, operation)
int fildes, operation;
```

DESCRIPTION

Dg_flock applies or removes an advisory lock on the file associated with the file descriptor *fildes*, depending on the *operation* specified. A lock is applied by specifying an operation parameter that is the 'exclusive or' of LOCK_SH or LOCK_EX and, possibly, LOCK_NB. *Operation* should be LOCK_UN to unlock an existing lock.

Advisory locks allow cooperating processes to perform consistent operations on files, but do not guarantee consistency (i.e., processes may still access files without using advisory locks, possibly resulting in inconsistencies).

The locking mechanism allows two types of locks: shared locks and exclusive locks. A shared lock is similar to a read lock as described in fcntl(2). That is, an exclusive lock cannot be applied as long as a shared lock is in effect. An exclusive lock is similar to a write lock in that no other lock can be applied as long as the exclusive lock is in effect.

At any time, multiple shared locks may be applied to a file, but at no time are multiple exclusive, or both shared and exclusive, locks allowed simultaneously on a file.

A shared lock may be upgraded to an exclusive lock, and vice versa, simply by specifying the appropriate lock. This is an atomic operation; that is, the lock is not released during the change.

The child of a process acts independently of its parent process in regards to locks. More specifically, the child of a process does not inherit locks; and if a child unlocks a file, it does not release the lock of the parent.

Requesting a lock on an object that is already locked normally causes the caller to be blocked until the lock may be acquired. If LOCK_NB is included in *operation*, then this block will not happen; instead the call will fail and the errno EACCES will be set.

Fildes must be open for reading in order to obtain a shared lock, open for writing to obtain an exclusive lock.

RETURN VALUE

If the operation was successful, 0 is returned; on an error, a –1 is returned and an error code is left in the global location errno.

DIAGNOSTICS

The dg_flock call fails if:

[EACCES] The file is locked and the LOCK_NB option was specified.

[EBADF] The argument *fildes* is an invalid descriptor, or its mode does not allow a request of the given lock type.

[EINTR]	An interrupt was received before the lock was obtained.
[EINVAL]	The argument *fildes* refers to an object other than a file.
[EDEADLK]	Awaiting the lock would cause a deadlock.

SEE ALSO

`open(2)`, `close(2)`, `dup(2)`, `exec(2)`, `fork(2)`, `fcntl(2)`.

NOTES

Locks are on files, not file descriptors. That is, file descriptors duplicated through dup(2) do not result in multiple instances of a lock.

Blocked processes awaiting a lock may be awakened by signals.

Note the following departures from Berkeley 4.2 specifications for `dg_flock` compared to the BSD `flock` call:

> `Dg_flock` can return EDEADLK whereas 4.2 BSD would simply deadlock.
>
> Upgrading a lock will be an atomic operation. That is, a lock is not released to upgrade that lock.
>
> `Dg_flock` will return EACCES instead of EWOULDBLOCK if using LOCK_NB and a lock is already held.
>
> A child process does not inherit locks from its parent process.
>
> *Fildes* must be open for reading to obtain a shared lock, open for writing to obtain an exclusive lock.
>
> `Dg_flock` and `lockf` are mutually compatible until `lockf` has a mandatory locking option.

NAME

dg_seek, dg_block_seek – extended seek functions

SYNOPSIS

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

boolean_type dg_seek (int fildes, uint high, uint low);

boolean_type dg_block_seek (int fildes, ulong block);
```

DESCRIPTION

Dg_seek performs an extended seek operation to a device whose offsets are too large to be handled by lseek(2). Dg_seek will use the values high and low to form a 64 bit file position value with respect to position zero in the file. The current file position for fildes is then set to this value. High is the top 32 bits of the device address and low is the bottom 32 bits of the device address.

Dg_block_seek will use block to create a 64 bit file position value, using 512-byte blocks as the block size. The current file position for fildes is then set to this value.

SEE ALSO

lseek(2).

DIAGNOSTICS

Both functions return TRUE (1 as defined in <sys/dg_c_generics.h>) when the seek is successful, and FALSE (0 as defined in <sys/dg_c_generics.h>) when unsuccessful. Calling this function on an invalid file descriptor will also result in a FALSE result.

NAME

dial – establish an out-going terminal line connection

SYNOPSIS

```
#include <dial.h>

int dial (call)
CALL call;

void undial (fd)
int fd;
```

DESCRIPTION

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

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

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

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

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

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

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

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

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

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

FILES

`/etc/uucp/Devices`
`/etc/uucp/Systems`
`/usr/spool/uucp/LCK..`*tty-device*

DIAGNOSTICS

On failure, a negative value indicating the reason for the failure will be returned. Mnemonics for these negative indices as listed here are defined in the `dial.h` header file.

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

SEE ALSO

`alarm`(2), `read`(2), `write`(2).
`acu`(7), `termio`(7) in the *System Manager's Reference for the DG/UX System*.
`uucp`(1C) in the *User's Reference for the DG/UX System*.

CAUTIONS

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

The above routine uses `stdio.h`, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

BUGS

An `alarm`(2) system call for 3600 seconds is made (and caught) within the `dial` module for the purpose of "touching" the *LCK..* file and constitutes the device allocation semaphore for the terminal device. Otherwise, `uucp`(1C) may simply delete the `LCK..` entry on its 90-minute clean-up rounds. The alarm may go off while the user program is in a `read`(2) or `write`(2) system call, causing an apparent error return. If the user program expects to be around for an hour or more, error returns from *read*s should be checked for `(errno==EINTR)`, and the *read* possibly reissued.

NAME

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

SYNOPSIS

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

DIR *opendir (const char *filename);

struct dirent *readdir (DIR *dirp);

long telldir (DIR *dirp);

void seekdir (DIR *dirp, long loc);

void rewinddir (DIR *dirp);

int closedir (DIR *dirp);
```

Alternate Syntax

```
#include <sys/dir.h>
struct direct *readdir (DIR *dirp);
```

DESCRIPTION

opendir opens the directory named by *filename* and associates a directory stream with it. opendir returns a pointer to be used to identify the directory stream in subsequent operations. The directory stream is positioned at the first entry. A null pointer is returned if *filename* cannot be accessed or is not a directory, or if it cannot malloc(3C) enough memory to hold a DIR structure or a buffer for the directory entries.

readdir returns a pointer to the next active directory entry and positions the directory stream at the next entry. No inactive entries are returned. It returns NULL upon reaching the end of the directory or upon detecting an invalid location in the directory. readdir buffers several directory entries per actual read operation; readdir marks for update the st_atime field of the directory each time the directory is actually read.

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 position associated with the directory stream at the time the telldir operation that provides *loc* was performed. Values returned by telldir are valid only if the directory has not changed because of compaction or expansion. This situation is not a problem with System V, but it can be with DG/UX and some file system types.

rewinddir resets the position of the named directory stream to the beginning of the directory. It also causes the directory stream to refer to the current state of the corresponding directory, as a call to opendir would.

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

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

opendir returns NULL on failure and sets errno to one of the following values:

ENOTDIR	A component of *filename* is not a directory.
EACCES	A component of *filename* denies search permission.

EACCES Read permission is denied on the specified directory.

EMFILE The maximum number of file descriptors are currently open.

ENFILE The system file table is full.

EFAULT *filename* points outside the allocated address space.

ELOOP Too many symbolic links were encountered in translating *filename*.

ENAMETOOLONG The length of the *filename* argument exceeds {PATH_MAX}, or the length of a *filename* component exceeds {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

ENOENT A component of *filename* does not exist or is a null pathname.

readdir returns NULL on failure and sets errno to one of the following values:

ENOENT The current file pointer for the directory is not located at a valid entry.

EBADF The file descriptor determined by the DIR stream is no longer valid. This result occurs if the DIR stream has been closed.

telldir, seekdir, and closedir return −1 on failure and set errno to the following value:

EBADF The file descriptor determined by the DIR stream is no longer valid. This results if the DIR stream has been closed.

EXAMPLE

Here is a sample program that prints the names of all the files in the current directory:

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

main()
{
      DIR *dirp;
      struct dirent *direntp;

      dirp = opendir( "." );
      while ( (direntp = readdir( dirp )) != NULL )
            (void)printf( "%s\n", direntp->d_name );
      closedir( dirp );
      return (0);
}
```

SEE ALSO

getdents(2), dirent(4).

NOTES

rewinddir is implemented as a macro, so its function address cannot be taken.

NAME

dirname – report the parent directory name of a file path name

SYNOPSIS

cc [*flag* ...] *file* ... -lgen [*library* ...]

```
#include <libgen.h>
```

```
char *dirname (char *path);
```

DESCRIPTION

Given a pointer to a null-terminated character string that contains a file system path name, dirname returns a pointer to a static constant string that is the parent directory of that file. In doing this, it sometimes places a null byte in the path name after the next to last element, so the content of *path* must be disposable. Trailing "/" characters in the path are not counted as part of the path.

If *path* or **path* is zero, a pointer to a static constant "." is returned.

dirname and basename together yield a complete path name. dirname (*path*) is the directory where basename (*path*) is found.

EXAMPLES

A simple file name and the strings "." and ".." all have "." as their return value.

Input string	Output pointer
/usr/lib	/usr
/usr/	/
usr	.
/	/
.	.
..	.

The following code reads a path name, changes directory to the appropriate directory [see chdir(2)], and opens the file.

```
char path[100], *pathcopy;
int fd;
gets (path);
pathcopy = strdup (path);
chdir (dirname (pathcopy) );
fd = open (basename (path), O_RDONLY);
```

SEE ALSO

chdir(2), basename(3G).
basename(1) in the *User's Reference Manual*.

NAME

div, ldiv – compute the quotient and remainder

SYNOPSIS

```
#include <stdlib.h>

div_t div (int numer, int denom);

ldiv_t ldiv (long int numer, long int denom);
```

DESCRIPTION

div computes the quotient and remainder of the division of the numerator *numer* by the denominator *denom*. This function provides a well-defined semantics for the signed integral division and remainder operations, unlike the implementation-defined semantics of the built-in operations. The sign of the resulting quotient is that of the algebraic quotient, and, if the division is inexact, the magnitude of the resulting quotient is the largest integer less than the magnitude of the algebraic quotient. If the result cannot be represented, the behavior is undefined; otherwise, *quotient* * *denom* + *remainder* will equal *numer*.

div returns a structure of type div_t, comprising both the quotient and remainder:

```
typedef struct div_t {
      int   quot; /*quotient*/
      int   rem;  /*remainder*/
} div_t;
```

ldiv is similar to div, except that the arguments and the members of the returned structure (which has type ldiv_t) all have type long int.

SEE ALSO

mp(3X).

NAME

doconfig – execute a configuration script

SYNOPSIS

```
# include <sac.h> int doconfig(int fd, char *script, long rflag);
```

DESCRIPTION

doconfig is a Service Access Facility library function that interprets the configuration scripts contained in the files /etc/saf/*pmtag/*_config, /etc/saf/_sysconfig, and /etc/saf/*pmtag*/*svctag*.

script is the name of the configuration script; *fd* is a file descriptor that designates the stream to which stream manipulation operations are to be applied; *rflag* is a bitmask that indicates the mode in which script is to be interpreted. *rflag* may take two values, NORUN and NOASSIGN, which may be or'd. If *rflag* is zero, all commands in the configuration script are eligible to be interpreted. If *rflag* has the NOASSIGN bit set, the assign command is considered illegal and will generate an error return. If *rflag* has the NORUN bit set, the run and runwait commands are considered illegal and will generate error returns.

The configuration language in which script is written consists of a sequence of commands, each of which is interpreted separately. The following reserved keywords are defined: assign, push, pop, runwait, and run. The comment character is #; when a # occurs on a line, everything from that point to the end of the line is ignored. Blank lines are not significant. No line in a command script may exceed 1024 characters.

assign *variable=value*

Used to define environment variables. *variable* is the name of the environment variable and *value* is the value to be assigned to it. The value assigned must be a string constant; no form of parameter substitution is available. *value* may be quoted. The quoting rules are those used by the shell for defining environment variables. assign will fail if space cannot be allocated for the new variable or if any part of the specification is invalid.

push *module1*[, *module2*, *module3*, . . .]

Used to push STREAMS modules onto the stream designated by *fd*. *module1* is the name of the first module to be pushed, *module2* is the name of the second module to be pushed, etc. The command will fail if any of the named modules cannot be pushed. If a module cannot be pushed, the subsequent modules on the same command line will be ignored and modules that have already been pushed will be popped.

pop [*module*]

Used to pop STREAMS modules off the designated stream. If pop is invoked with no arguments, the top module on the stream is popped. If an argument is given, modules will be popped one at a time until the named module is at the top of the stream. If the named module is not on the designated stream, the stream is left as it was and the command fails. If *module* is the special. keyword ALL, then all modules on the stream will be popped. Note that only modules above the topmost driver are affected.

runwait *command*

The runwait command runs a command and waits for it to complete. *command* is the pathname of the command to be run. The command is run with /usr/bin/sh -c prepended to it; shell scripts may thus be executed from configuration scripts. The `runwait` command will fail if *command* cannot be found or cannot be executed, or if *command* exits with a non-zero status.

`run` *command*

The `run` command is identical to `runwait` except that it does not wait for *command* to complete. *command* is the pathname of the command to be run. `run` will not fail unless it is unable to create a child process to execute the command.

Although they are syntactically indistinguishable, some of the commands available to `run` and `runwait` are interpreter built-in commands. Interpreter built-ins are used when it is necessary to alter the state of a process within the context of that process. The `doconfig` interpreter built-in commands are similar to the shell special commands and, like these, they do not spawn another process for execution. See `sh`(1). The initial set of built-in commands is:

```
cd
ulimit
umask
```

DIAGNOSTICS

`doconfig` returns `0` if the script was interpreted successfully. If a command in the script fails, the interpretation of the script ceases at that point and a positive number is returned; this number indicates which line in the script failed. If a system error occurs, a value of `-1` is returned. When a script fails, the process whose environment was being established should *not* be started.

SEE ALSO

`pmadm`(1M), `sacadm`(1M), `sh`(1).

NAME

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

SYNOPSIS

```
#include <stdlib.h>

double drand48 (void);

double erand48 (unsigned short xsubi[3] );

long lrand48 (void);

long nrand48 (unsigned short xsubi[3] );

long mrand48 (void);

long jrand48 (unsigned short xsubi[3] );

void srand48 (long seedval);

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

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

DESCRIPTION

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

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

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

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

Functions srand48, seed48, and lcong48 are initialization entry points, one of which should be invoked before either drand48, lrand48, or mrand48 is called. (Although it is not recommended practice, constant default initializer values will be supplied automatically if drand48, lrand48, or mrand48 is called without a prior call to an initialization entry point.) Functions erand48, nrand48, and jrand48 do not require an initialization entry point to be called first.

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

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

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.

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

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

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

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

SEE ALSO

`rand`(3C).

NAME

drem – IEEE floating-point remainder

SYNOPSIS

```
#include <ieeefp.h> /* for IEEE environment */
```

or

```
#include <math.h>    /* for System V environment */
```

```
double drem (x,y)
double x, y;
```

DESCRIPTION

drem returns the remainder of *x/y* as specified by IEEE Standard 754 for Binary Floating-Point Arithmetic. The remainder *r* is calculated as

$$r = x - n * y$$

where *n* is the integer nearest the exact value of *x/y*. If the absolute value of *x/y* is .5, then *n* is the even integer nearest the result of *x/y*. Since the IEEE standard requires that the exact value of *x/y* be used in the calculation, the 'round nearest' rounding mode is in effect throughout the execution of the drem function. The remainder is always considered to be exact, so inexact exceptions are never raised.

DIAGNOSTICS

The IEEE standard defines drem(*x*, 0) and drem(*infinity*, *y*) to be invalid operations. In addition, this implementation considers drem(*x*, *y*) to be an invalid operation when *x/y* results in infinity or double-precision overflow. These operations raise an 'invalid operation' exception, which results in signal SIGFPE if traps are enabled and a NaN otherwise.

SEE ALSO

fpgetround(3C), isnan(3C).

BUGS

This implementation of drem does not allow exceptional values to be fixed by a trap handler.

NAME

ecvt, fcvt, gcvt – convert floating-point number to string

SYNOPSIS

```
#include <stdlib.h>

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

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

char *gcvt (double value, int ndigit, char *buf);
```

DESCRIPTION

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

fcvt is identical to ecvt, except that the correct digit has been rounded for printf %f output of the number of digits specified by *ndigit*. So, where ecvt(12.3456, 3, decpt, sign) returns a pointer to character string 123\0, fcvt(12.3456, 3, decpt, sign) returns a pointer to character string 123456\0.

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

SEE ALSO

printf(3S).

NOTES

The values returned by ecvt and fcvt point to a single static data array whose content is overwritten by each call.

NAME

`elf` – object file access library

SYNOPSIS

cc [*flag* ...] *file* ... `-lelf` [*library* ...]

```
#include <libelf.h>
```

DESCRIPTION

Functions in the ELF access library let a program manipulate ELF (Executable and Linking Format) object files, archive files, and archive members. The header file provides type and function declarations for all library services.

Programs communicate with many of the higher-level routines using an *ELF descriptor*. That is, when the program starts working with a file, `elf_begin` creates an ELF descriptor through which the program manipulates the structures and information in the file. These ELF descriptors can be used both to read and to write files. After the program establishes an ELF descriptor for a file, it may then obtain *section descriptors* to manipulate the sections of the file [see `elf_getscn`(3E)]. Sections hold the bulk of an object file's real information, such as text, data, the symbol table, and so on. A section descriptor "belongs" to a particular ELF descriptor, just as a section belongs to a file. Finally, *data descriptors* are available through section descriptors, allowing the program to manipulate the information associated with a section. A data descriptor "belongs" to a section descriptor.

Descriptors provide private handles to a file and its pieces. In other words, a data descriptor is associated with one section descriptor, which is associated with one ELF descriptor, which is associated with one file. Although descriptors are private, they give access to data that may be shared. Consider programs that combine input files, using incoming data to create or update another file. Such a program might get data descriptors for an input and an output section. It then could update the output descriptor to reuse the input descriptor's data. That is, the descriptors are distinct, but they could share the associated data bytes. This sharing avoids the space overhead for duplicate buffers and the performance overhead for copying data unnecessarily.

File Classes

ELF provides a framework in which to define a family of object files, supporting multiple processors and architectures. An important distinction among object files is the *class*, or capacity, of the file. The 32-bit class supports architectures in which a 32-bit object can represent addresses, file sizes, etc., as in the following.

Name	Purpose
`Elf32_Addr`	Unsigned address
`Elf32_Half`	Unsigned medium integer
`Elf32_Off`	Unsigned file offset
`Elf32_Sword`	Signed large integer
`Elf32_Word`	Unsigned large integer
`unsigned char`	Unsigned small integer

Other classes will be defined as necessary, to support larger (or smaller) machines. Some library services deal only with data objects for a specific class, while others are class-independent. To make this distinction clear, library function names reflect their status, as described below.

Data Representations

Conceptually, two parallel sets of objects support cross compilation environments.

One set corresponds to file contents, while the other set corresponds to the native memory image of the program manipulating the file. Type definitions supplied by the header files work on the native machine, which may have different data encodings (size, byte order, etc.) than the target machine. Although native memory objects should be at least as big as the file objects (to avoid information loss), they may be bigger if that is more natural for the host machine.

Translation facilities exist to convert between file and memory representations. Some library routines convert data automatically, while others leave conversion as the program's responsibility. Either way, programs that create object files must write file-typed objects to those files; programs that read object files must take a similar view. See `elf_xlate`(3E) and `elf_fsize`(3E) for more information.

Programs may translate data explicitly, taking full control over the object file layout and semantics. If the program prefers not to have and exercise complete control, the library provides a higher-level interface that hides many object file details. `elf_begin` and related functions let a program deal with the native memory types, converting between memory objects and their file equivalents automatically when reading or writing an object file.

Elf Versions

Object file versions allow ELF to adapt to new requirements. Three—independent—versions can be important to a program. First, an application program knows about a particular version by virtue of being compiled with certain header files. Second, the access library similarly is compiled with header files that control what versions it understands. Third, an ELF object file holds a value identifying its version, determined by the ELF version known by the file's creator. Ideally, all three versions would be the same, but they may differ.

> If a program's version is newer than the access library, the program might use information unknown to the library. Translation routines might not work properly, leading to undefined behavior. This condition merits installing a new library.
>
> The library's version might be newer than the program's and the file's. The library understands old versions, thus avoiding compatibility problems in this case.
>
> Finally, a file's version might be newer than either the program or the library understands. The program might or might not be able to process the file properly, depending on whether the file has extra information and whether that information can be safely ignored. Again, the safe alternative is to install a new library that understands the file's version.

To accommodate these differences, a program must use `elf_version` to pass its version to the library, thus establishing the *working version* for the process. Using this, the library accepts data from and presents data to the program in the proper representations. When the library reads object files, it uses each file's version to interpret the data. When writing files or converting memory types to the file equivalents, the library uses the program's working version for the file data.

System Services

As mentioned above, `elf_begin` and related routines provide a higher-level interface to ELF files, performing input and output on behalf of the application program. These routines assume a program can hold entire files in memory, without explicitly using temporary files. When reading a file, the library routines bring the data into memory and perform subsequent operations on the memory copy. Programs that wish to read or write large object files with this model must execute on a machine with a large process virtual address space. If the underlying operating system limits the number of open files, a program can use `elf_cntl` to retrieve all necessary data from the file, allowing the program to close the file descriptor and reuse it.

Although the `elf_begin` interfaces are convenient and efficient for many programs, they might be inappropriate for some. In those cases, an application may invoke the `elf_xlate` data translation routines directly. These routines perform no input or output, leaving that as the application's responsibility. By assuming a larger share of the job, an application controls its input and output model.

Library Names

Names associated with the library take several forms.

`elf_`*name* These class-independent names perform some service, *name*, for the program.

`elf32_`*name* Service names with an embedded class, `32` here, indicate they work only for the designated class of files.

`Elf_`*Type* Data types can be class-independent as well, distinguished by *Type*.

`Elf32_`*Type* Class-dependent data types have an embedded class name, `32` here.

`ELF_C_`*CMD* Several functions take commands that control their actions. These values are members of the `Elf_Cmd` enumeration; they range from zero through `ELF_C_NUM-1`.

`ELF_F_`*FLAG* Several functions take flags that control library status and/or actions. Flags are bits that may be combined.

`ELF32_FSZ_`*TYPE*

These constants give the file sizes in bytes of the basic ELF types for the 32-bit class of files. See `elf_fsize` for more information.

`ELF_K_`*KIND* The function `elf_kind` identifies the *KIND* of file associated with an ELF descriptor. These values are members of the `Elf_Kind` enumeration; they range from zero through `ELF_K_NUM-1`.

`ELF_T_`*TYPE* When a service function, such as `elf_xlate`, deals with multiple types, names of this form specify the desired *TYPE*. Thus, for example, `ELF_T_EHDR` is directly related to `Elf32_Ehdr`. These values are members of the `Elf_Type` enumeration; they range from zero through `ELF_T_NUM-1`.

SEE ALSO

`cof2elf`(1), `elf_begin`(3E), `elf_cntl`(3E), `elf_end`(3E), `elf_error`(3E), `elf_fill`(3E), `elf_flag`(3E), `elf_fsize`(3E), `elf_getarhdr`(3E), `elf_getarsym`(3E), `elf_getbase`(3E), `elf_getdata`(3E), `elf_getehdr`(3E), `elf_getident`(3E), `elf_getphdr`(3E), `elf_getscn`(3E), `elf_getshdr`(3E), `elf_hash`(3E), `elf_kind`(3E), `elf_next`(3E), `elf_rand`(3E), `elf_rawfile`(3E), `elf_strptr`(3E), `elf_update`(3E), `elf_version`(3E), `elf_xlate`(3E), `a.out`(4) `ar`(4)

The "Object Files" chapterin the *Programmer's Guide: ANSI C and Programming Support Tools*.

NOTES

Information in the ELF header files is separated into common parts and processor-specific parts. A program can make a processor's information available by including the appropriate header file: <sys/elf_*NAME*.h> where *NAME* matches the processor name as used in the ELF file header.

Symbol	Processor
M32	AT&T WE 32100
SPARC	SPARC
386	Intel 80386
486	Intel 80486
860	Intel 80860
68K	Motorola 68000
88K	Motorola 88000

Other processors will be added to the table as necessary. To illustrate, a program could use the following code to "see" the processor-specific information for the Motorola 88000:

```
#include <libelf.h>
#include <sys/elf_88K.h>
```

Without the <sys/elf_88K.h> definition, only the common ELF information would be visible.

NAME

elf_begin – make a file descriptor

SYNOPSIS

cc [*flag* ...] *file* ... -lelf [*library* ...]

```
#include <libelf.h>

Elf *elf_begin(int fildes, Elf_Cmd cmd, Elf *ref);
```

DESCRIPTION

elf_begin, elf_next, elf_rand, and elf_end work together to process ELF object files, either individually or as members of archives. After obtaining an ELF descriptor from elf_begin, the program may read an existing file, update an existing file, or create a new file. *fildes* is an open file descriptor that elf_begin uses for reading or writing. The initial file offset [see lseek(2)] is unconstrained, and the resulting file offset is undefined.

cmd may have the following values.

ELF_C_NULL When a program sets *cmd* to this value, elf_begin returns a null pointer, without opening a new descriptor. *ref* is ignored for this command. See elf_next(3E) and the examples below for more information.

ELF_C_READ When a program wishes to examine the contents of an existing file, it should set *cmd* to this value. Depending on the value of *ref*, this command examines archive members or entire files. Three cases can occur.

First, if *ref* is a null pointer, elf_begin allocates a new ELF descriptor and prepares to process the entire file. If the file being read is an archive, elf_begin also prepares the resulting descriptor to examine the initial archive member on the next call to elf_begin, as if the program had used elf_next or elf_rand to "move" to the initial member.

Second, if *ref* is a non-null descriptor associated with an archive file, elf_begin lets a program obtain a separate ELF descriptor associated with an individual member. The program should have used elf_next or elf_rand to position *ref* appropriately (except for the initial member, which elf_begin prepares; see the example below). In this case, *fildes* should be the same file descriptor used for the parent archive.

Finally, if *ref* is a non-null ELF descriptor that is not an archive, elf_begin increments the number of activations for the descriptor and returns *ref*, without allocating a new descriptor and without changing the descriptor's read/write permissions. To terminate the descriptor for *ref*, the program must call elf_end once for each activation. See elf_next(3E) and the examples below for more information.

ELF_C_RDWR This command duplicates the actions of ELF_C_READ and additionally allows the program to update the file image [see elf_update(3E)]. That is, using ELF_C_READ gives a read-only view of the file, while ELF_C_RDWR lets the program read *and* write the file. ELF_C_RDWR is not valid for archive members. If *ref* is non-null, it must have been created with the ELF_C_RDWR command.

ELF_C_WRITE If the program wishes to ignore previous file contents, presumably to create a new file, it should set *cmd* to this value. *ref* is ignored for this command.

elf_begin "works" on all files (including files with zero bytes), providing it can allocate memory for its internal structures and read any necessary information from the file. Programs reading object files thus may call elf_kind or elf_getehdr to determine the file type (only object files have an ELF header). If the file is an archive with no more members to process, or an error occurs, elf_begin returns a null pointer. Otherwise, the return value is a non-null ELF descriptor.

Before the first call to elf_begin, a program must call elf_version to coordinate versions.

SYSTEM SERVICES

When processing a file, the library decides when to read or write the file, depending on the program's requests. Normally, the library assumes the file descriptor remains usable for the life of the ELF descriptor. If, however, a program must process many files simultaneously and the underlying operating system limits the number of open files, the program can use elf_cntl to let it reuse file descriptors. After calling elf_cntl with appropriate arguments, the program may close the file descriptor without interfering with the library.

All data associated with an ELF descriptor remain allocated until elf_end terminates the descriptor's last activation. After the descriptors have been terminated, the storage is released; attempting to reference such data gives undefined behavior. Consequently, a program that deals with multiple input (or output) files must keep the ELF descriptors active until it finishes with them.

EXAMPLES

A prototype for reading a file appears below. If the file is a simple object file, the program executes the loop one time, receiving a null descriptor in the second iteration. In this case, both elf and arf will have the same value, the activation count will be two, and the program calls elf_end twice to terminate the descriptor. If the file is an archive, the loop processes each archive member in turn, ignoring those that are not object files.

```
if (elf_version(EV_CURRENT) == EV_NONE)
{
        /* library out of date */
        /* recover from error */
}
cmd = ELF_C_READ;
arf = elf_begin(fildes, cmd, (Elf *)0);
while ((elf = elf_begin(fildes, cmd, arf)) != 0)
{
        if ((ehdr = elf32_getehdr(elf)) != 0)
        {
                /* process the file ... */
        }
        cmd = elf_next(elf);
        elf_end(elf);
}
elf_end(arf);
```

Alternatively, the next example illustrates random archive processing. After identifying the file as an archive, the program repeatedly processes archive members of interest. For clarity, this example omits error checking and ignores simple object files. Additionally, this fragment preserves the ELF descriptors for all archive members, because it does not call elf_end to terminate them.

```
elf_version(EV_CURRENT);
arf = elf_begin(fildes, ELF_C_READ, (Elf *)0);
if (elf_kind(arf) != ELF_K_AR)
{
        /* not an archive */
}
/* initial processing */
/* set offset = ... for desired member header */
while (elf_rand(arf, offset) == offset)
{
        if ((elf = elf_begin(fildes, ELF_C_READ, arf)) == 0)
                break;
        if ((ehdr = elf32_getehdr(elf)) != 0)
        {
                /* process archive member ... */
        }
        /* set offset = ... for desired member header */
}
```

The following outline shows how one might create a new ELF file. This example is simplified to show the overall flow.

```
elf_version(EV_CURRENT);
fildes = open("path/name", O_RDWR|O_TRUNC|O_CREAT, 0666);
if ((elf = elf_begin(fildes, ELF_C_WRITE, (Elf *)0)) == 0)
        return;
ehdr = elf32_newehdr(elf);
phdr = elf32_newphdr(elf, count);
scn = elf_newscn(elf);
shdr = elf32_getshdr(scn);
data = elf_newdata(scn);
elf_update(elf, ELF_C_WRITE);
elf_end(elf);
```

Finally, the following outline shows how one might update an existing ELF file. Again, this example is simplified to show the overall flow.

```
elf_version(EV_CURRENT);
fildes = open("path/name", O_RDWR);
elf = elf_begin(fildes, ELF_C_RDWR, (Elf *)0);

/* add new or delete old information ... */

close(creat("path/name", 0666));
elf_update(elf, ELF_C_WRITE);
elf_end(elf);
```

In the example above, the call to creat truncates the file, thus ensuring the resulting file will have the "right" size. Without truncation, the updated file might be as big as the original, even if information were deleted. The library truncates the file, if it can, with ftruncate [see truncate(2)]. Some systems, however, do not support ftruncate, and the call to creat protects against this.

Notice that both file creation examples open the file with write *and* read permissions. On systems that support mmap, the library uses it to enhance performance, and mmap requires a readable file descriptor. Although the library can use a write-only file descriptor, the application will not obtain the performance advantages of mmap.

SEE ALSO

cof2elf(1), creat(2), lseek(2), mmap(2), open(2), truncate(2), elf(3E), elf_cntl(3E), elf_end(3E), elf_getarhdr(3E), elf_getbase(3E), elf_getdata(3E), elf_getehdr(3E), elf_getphdr(3E), elf_getscn(3E), elf_kind(3E), elf_next(3E), elf_rand(3E), elf_rawfile(3E), elf_update(3E), elf_version(3E), ar(4).

NOTES

COFF is an object file format that preceded ELF . When a program calls elf_begin on a COFF file, the library translates COFF structures to their ELF equivalents, allowing programs to read (but not to write) a COFF file as if it were ELF . This conversion happens only to the memory image and *not* to the file itself. After the initial elf_begin, file offsets and addresses in the ELF header, the program headers, and the section headers retain the original COFF values [see elf_getehdr, elf_getphdr, and elf_getshdr]. A program may call elf_update to adjust these values (without writing the file), and the library will then present a consistent, ELF view of the file. Data obtained through elf_getdata are translated (the COFF symbol table is presented as ELF, etc.). Data viewed through elf_rawdata undergo no conversion, allowing the program to view the bytes from the file itself.

Some COFF debugging information is not translated, though this does not affect the semantics of a running program.

Although the ELF library supports COFF, programmers are strongly encouraged to recompile their programs, obtaining ELF object files.

NAME

elf_cntl – control a file descriptor

SYNOPSIS

cc [*flag* ...] *file* ... -lelf [*library* ...]

```
#include <libelf.h>

int elf_cntl(Elf *elf, Elf_Cmd cmd);
```

DESCRIPTION

elf_cntl instructs the library to modify its behavior with respect to an ELF descriptor, *elf*. As elf_begin(3E) describes, an ELF descriptor can have multiple activations, and multiple ELF descriptors may share a single file descriptor. Generally, elf_cntl commands apply to all activations of *elf*. Moreover, if the ELF descriptor is associated with an archive file, descriptors for members within the archive will also be affected as described below. Unless stated otherwise, operations on archive members do not affect the descriptor for the containing archive.

The *cmd* argument tells what actions to take and may have the following values.

ELF_C_FDDONE This value tells the library not to use the file descriptor associated with *elf*. A program should use this command when it has requested all the information it cares to use and wishes to avoid the overhead of reading the rest of the file. The memory for all completed operations remains valid, but later file operations, such as the initial elf_getdata for a section, will fail if the data are not in memory already.

ELF_C_FDREAD This command is similar to ELF_C_FDDONE, except it forces the library to read the rest of the file. A program should use this command when it must close the file descriptor but has not yet read everything it needs from the file. After elf_cntl completes the ELF_C_FDREAD command, future operations, such as elf_getdata, will use the memory version of the file without needing to use the file descriptor.

If elf_cntl succeeds, it returns zero. Otherwise *elf* was null or an error occurred, and the function returns -1.

SEE ALSO

elf(3E), elf_begin(3E), elf_getdata(3E), elf_rawfile(3E).

NOTE

If the program wishes to use the "raw" operations [see elf_rawdata, which elf_getdata(3E) describes, and elf_rawfile(3E)] after disabling the file descriptor with ELF_C_FDDONE or ELF_C_FDREAD, it must execute the raw operations explicitly beforehand. Otherwise, the raw file operations will fail. Calling elf_rawfile makes the entire image available, thus supporting subsequent elf_rawdata calls.

NAME

elf_end – finish using an object file

SYNOPSIS

```
cc [flag ...] file ...  -lelf [library ...]

#include <libelf.h>

int elf_end(Elf *elf);
```

DESCRIPTION

A program uses elf_end to terminate an ELF descriptor, *elf*, and to deallocate data associated with the descriptor. Until the program terminates a descriptor, the data remain allocated. *elf* should be a value previously returned by elf_begin; a null pointer is allowed as an argument, to simplify error handling. If the program wishes to write data associated with the ELF descriptor to the file, it must use elf_update before calling elf_end.

As elf_begin(3E) explains, a descriptor can have more than one activation. Calling elf_end removes one activation and returns the remaining activation count. The library does not terminate the descriptor until the activation count reaches zero. Consequently, a zero return value indicates the ELF descriptor is no longer valid.

SEE ALSO

elf(3E), elf_begin(3E), elf_update(3E).

NAME

elf_errmsg, elf_errno – error handling

SYNOPSIS

cc [*flag* ...] *file* ... -lelf [*library* ...]

```
#include <libelf.h>

const char *elf_errmsg(int err);
int elf_errno(void);
```

DESCRIPTION

If an ELF library function fails, a program may call elf_errno to retrieve the library's internal error number. As a side effect, this function resets the internal error number to zero, which indicates no error.

elf_errmsg takes an error number, *err*, and returns a null-terminated error message (with no trailing new-line) that describes the problem. A zero *err* retrieves a message for the most recent error. If no error has occurred, the return value is a null pointer (not a pointer to the null string). Using *err* of –1 also retrieves the most recent error, except it guarantees a non-null return value, even when no error has occurred. If no message is available for the given number, elf_errmsg returns a pointer to an appropriate message. This function does not have the side effect of clearing the internal error number.

EXAMPLE

The following fragment clears the internal error number and checks it later for errors. Unless an error occurs after the first call to elf_errno, the next call will return zero.

```
(void)elf_errno();
while (more_to_do)
{
        /* processing ... */
        if ((err = elf_errno()) != 0)
        {
                msg = elf_errmsg(err);
                /* print msg */
        }
}
```

SEE ALSO

elf(3E), elf_version(3E).

NAME

`elf_fill` – set fill byte

SYNOPSIS

`cc` [*flag* ...] *file* ... `-lelf` [*library* ...]

```
#include <libelf.h>

void elf_fill(int fill);
```

DESCRIPTION

Alignment constraints for ELF files sometimes require the presence of "holes." For example, if the data for one section are required to begin on an eight-byte boundary, but the preceding section is too "short," the library must fill the intervening bytes. These bytes are set to the *fill* character. The library uses zero bytes unless the application supplies a value. See `elf_getdata`(3E) for more information about these holes.

SEE ALSO

`elf`(3E), `elf_getdata`(3E), `elf_flag`(3E), `elf_update`(3E).

NOTE

An application can assume control of the object file organization by setting the `ELF_F_LAYOUT` bit [see `elf_flag`(3E)]. When this is done, the library does *not* fill holes.

NAME

`elf_flagdata`, `elf_flagehdr`, `elf_flagelf`, `elf_flagphdr`, `elf_flagscn`, `elf_flagshdr` – manipulate flags

SYNOPSIS

cc [*flag* ...] *file* ... `-lelf` [*library* ...]

```
#include <libelf.h>

unsigned elf_flagdata(Elf_Data *data, Elf_Cmd cmd, unsigned flags);

unsigned elf_flagehdr(Elf *elf, Elf_Cmd cmd, unsigned flags);

unsigned elf_flagelf(Elf *elf, Elf_Cmd cmd, unsigned flags);

unsigned elf_flagphdr(Elf *elf, Elf_Cmd cmd, unsigned flags);

unsigned elf_flagscn(Elf_Scn *scn, Elf_Cmd cmd, unsigned flags);

unsigned elf_flagshdr(Elf_Scn *scn, Elf_Cmd cmd, unsigned flags);
```

DESCRIPTION

These functions manipulate the flags associated with various structures of an ELF file. Given an ELF descriptor (*elf*), a data descriptor (*data*), or a section descriptor (*scn*), the functions may set or clear the associated status bits, returning the updated bits. A null descriptor is allowed, to simplify error handling; all functions return zero for this degenerate case.

cmd may have the following values:

`ELF_C_CLR` The functions clear the bits that are asserted in *flags*. Only the non-zero bits in *flags* are cleared; zero bits do not change the status of the descriptor.

`ELF_C_SET` The functions set the bits that are asserted in *flags*. Only the non-zero bits in *flags* are set; zero bits do not change the status of the descriptor.

Descriptions of the defined *flags* bits appear below.

`ELF_F_DIRTY` When the program intends to write an ELF file, this flag asserts the associated information needs to be written to the file. Thus, for example, a program that wished to update the ELF header of an existing file would call `elf_flagehdr` with this bit set in *flags* and *cmd* equal to `ELF_C_SET`. A later call to `elf_update` would write the marked header to the file.

`ELF_F_LAYOUT` Normally, the library decides how to arrange an output file. That is, it automatically decides where to place sections, how to align them in the file, etc. If this bit is set for an ELF descriptor, the program assumes responsibility for determining all file positions. This bit is meaningful only for `elf_flagelf` and applies to the entire file associated with the descriptor.

When a flag bit is set for an item, it affects all the subitems as well. Thus, for example, if the program sets the `ELF_F_DIRTY` bit with `elf_flagelf`, the entire logical file is "dirty."

EXAMPLE

The following fragment shows how one might mark the ELF header to be written to the output file.

```
        ehdr = elf32_getehdr(elf);
        /* dirty ehdr ... */
        elf_flagehdr(elf, ELF_C_SET, ELF_F_DIRTY);
```

SEE ALSO

elf(3E), elf_end(3E), elf_getdata(3E), elf_getehdr(3E), elf_update(3E).

NAME

elf_fsize: elf32_fsize – return the size of an object file type

SYNOPSIS

cc [*flag* ...] *file* ... -lelf [*library* ...]

```
#include <libelf.h>
```

```
size_t elf32_fsize(Elf_Type type, size_t count, unsigned ver);
```

DESCRIPTION

elf32_fsize gives the size in bytes of the 32-bit file representation of *count* data objects with the given *type*. The library uses version *ver* to calculate the size [see elf(3E) and elf_version(3E)].

Constant values are available for the sizes of fundamental types.

Elf_Type	File Size	Memory Size
ELF_T_ADDR	ELF32_FSZ_ADDR	sizeof(Elf32_Addr)
ELF_T_BYTE	1	sizeof(unsigned char)
ELF_T_HALF	ELF32_FSZ_HALF	sizeof(Elf32_Half)
ELT_T_OFF	ELF32_FSZ_OFF	sizeof(Elf32_Off)
ELF_T_SWORD	ELF32_FSZ_SWORD	sizeof(Elf32_Sword)
ELF_T_WORD	ELF32_FSZ_WORD	sizeof(Elf32_Word)

elf32_fsize returns zero if the value of *type* or *ver* is unknown. See elf_xlate(3E) for a list of the *type* values.

SEE ALSO

elf(3E), elf_version(3E), elf_xlate(3E).

NAME

elf_getarhdr – retrieve archive member header

SYNOPSIS

cc [*flag* ...] *file* ... -lelf [*library* ...]

```
#include <libelf.h>

Elf_Arhdr *elf_getarhdr(Elf *elf);
```

DESCRIPTION

elf_getarhdr returns a pointer to an archive member header, if one is available for the ELF descriptor *elf*. Otherwise, no archive member header exists, an error occurred, or *elf* was null; elf_getarhdr then returns a null value. The header includes the following members.

```
char            *ar_name;
time_t          ar_date;
long            ar_uid;
long            ar_gid;
unsigned long   ar_mode;
off_t           ar_size;
char            *ar_rawname;
```

An archive member name, available through ar_name, is a null-terminated string, with the ar format control characters removed. The ar_rawname member holds a null-terminated string that represents the original name bytes in the file, including the terminating slash and trailing blanks as specified in the archive format.

In addition to "regular" archive members, the archive format defines some special members. All special member names begin with a slash (/), distinguishing them from regular members (whose names may not contain a slash). These special members have the names (ar_name) defined below.

/ This is the archive symbol table. If present, it will be the first archive member. A program may access the archive symbol table through elf_getarsym. The information in the symbol table is useful for random archive processing [see elf_rand(3E)].

// This member, if present, holds a string table for long archive member names. An archive member's header contains a 16-byte area for the name, which may be exceeded in some file systems. The library automatically retrieves long member names from the string table, setting ar_name to the appropriate value.

Under some error conditions, a member's name might not be available. Although this causes the library to set ar_name to a null pointer, the ar_rawname member will be set as usual.

SEE ALSO

elf(3E), elf_begin(3E), elf_getarsym(3E), elf_rand(3E), ar(4).

NAME

elf_getarsym – retrieve archive symbol table

SYNOPSIS

cc [*flag* ...] *file* ... -lelf [*library* ...]

```
#include <libelf.h>

Elf_Arsym *elf_getarsym(Elf *elf, size_t *ptr);
```

DESCRIPTION

elf_getarsym returns a pointer to the archive symbol table, if one is available for the ELF descriptor *elf*. Otherwise, the archive doesn't have a symbol table, an error occurred, or *elf* was null; elf_getarsym then returns a null value. The symbol table is an array of structures that include the following members.

```
char            *as_name;
size_t          as_off;
unsigned long   as_hash;
```

These members have the following semantics.

as_name A pointer to a null-terminated symbol name resides here.

as_off This value is a byte offset from the beginning of the archive to the member's header. The archive member residing at the given offset defines the associated symbol. Values in as_off may be passed as arguments to elf_rand to access the desired archive member.

as_hash This is a hash value for the name, as computed by elf_hash.

If *ptr* is non-null, the library stores the number of table entries in the location to which *ptr* points. This value is set to zero when the return value is null. The table's last entry, which is included in the count, has a null as_name, a zero value for as_off, and ~0UL for as_hash.

SEE ALSO

elf(3E), elf_getarhdr(3E), elf_hash(3E), elf_rand(3E), ar(4).

NAME

elf_getbase – get the base offset for an object file

SYNOPSIS

cc [*flag* ...] *file* ... -lelf [*library* ...]

```
#include <libelf.h>

off_t elf_getbase(Elf *elf);
```

DESCRIPTION

elf_getbase returns the file offset of the first byte of the file or archive member associated with *elf*, if it is known or obtainable, and –1 otherwise. A null *elf* is allowed, to simplify error handling; the return value in this case is –1. The base offset of an archive member is the beginning of the member's information, *not* the beginning of the archive member header.

SEE ALSO

elf(3E), elf_begin(3E), ar(4).

NAME

elf_getdata, elf_newdata, elf_rawdata – get section data

SYNOPSIS

cc [*flag* ...] *file* ... -lelf [*library* ...]

```
#include <libelf.h>

Elf_Data *elf_getdata(Elf_Scn *scn, Elf_Data *data);

Elf_Data *elf_newdata(Elf_Scn *scn);

Elf_Data *elf_rawdata(Elf_Scn *scn, Elf_Data *data);
```

DESCRIPTION

These functions access and manipulate the data associated with a section descriptor, *scn*. When reading an existing file, a section will have a single data buffer associated with it. A program may build a new section in pieces, however, composing the new data from multiple data buffers. For this reason, "the" data for a section should be viewed as a list of buffers, each of which is available through a data descriptor.

elf_getdata lets a program step through a section's data list. If the incoming data descriptor, *data*, is null, the function returns the first buffer associated with the section. Otherwise, *data* should be a data descriptor associated with *scn*, and the function gives the program access to the next data element for the section. If *scn* is null or an error occurs, elf_getdata returns a null pointer.

elf_getdata translates the data from file representations into memory representations [see elf_xlate(3E)] and presents objects with memory data types to the program, based on the file's *class* [see elf(3E)]. The working library version [see elf_version(3E)] specifies what version of the memory structures the program wishes elf_getdata to present.

elf_newdata creates a new data descriptor for a section, appending it to any data elements already associated with the section. As described below, the new data descriptor appears empty, indicating the element holds no data. For convenience, the descriptor's type (d_type below) is set to ELF_T_BYTE, and the version (d_version below) is set to the working version. The program is responsible for setting (or changing) the descriptor members as needed. This function implicitly sets the ELF_F_DIRTY bit for the section's data [see elf_flag(3E)]. If *scn* is null or an error occurs, elf_newdata returns a null pointer.

elf_rawdata differs from elf_getdata by returning only uninterpreted bytes, regardless of the section type. This function typically should be used only to retrieve a section image from a file being read, and then only when a program must avoid the automatic data translation described below. Moreover, a program may not close or disable [see elf_cntl(3E)] the file descriptor associated with *elf* before the initial raw operation, because elf_rawdata might read the data from the file to ensure it doesn't interfere with elf_getdata. See elf_rawfile(3E) for a related facility that applies to the entire file. When elf_getdata provides the right translation, its use is recommended over elf_rawdata. If *scn* is null cr an error occurs, elf_rawdata returns a null pointer.

The Elf_Data structure includes the following members:

```
void            *d_buf;
Elf_Type        d_type;
size_t          d_size;
off_t           d_off;
size_t          d_align;
unsigned        d_version;
```

These members are available for direct manipulation by the program. Descriptions appear below.

d_buf A pointer to the data buffer resides here. A data element with no data has a null pointer.

d_type This member's value specifies the type of the data to which d_buf points. A section's type determines how to interpret the section contents, as summarized below.

d_size This member holds the total size, in bytes, of the memory occupied by the data. This may differ from the size as represented in the file. The size will be zero if no data exist. [See the discussion of SHT_NOBITS below for more information.]

d_off This member gives the offset, within the section, at which the buffer resides. This offset is relative to the file's section, not the memory object's.

d_align This member holds the buffer's required alignment, from the beginning of the section. That is, d_off will be a multiple of this member's value. For example, if this member's value is four, the beginning of the buffer will be four-byte aligned within the section. Moreover, the entire section will be aligned to the maximum of its constituents, thus ensuring appropriate alignment for a buffer within the section and within the file.

d_version This member holds the version number of the objects in the buffer. When the library originally read the data from the object file, it used the working version to control the translation to memory objects.

Data Alignment

As mentioned above, data buffers within a section have explicit alignment constraints. Consequently, adjacent buffers sometimes will not abut, causing "holes" within a section. Programs that create output files have two ways of dealing with these holes.

First, the program can use elf_fill to tell the library how to set the intervening bytes. When the library must generate gaps in the file, it uses the fill byte to initialize the data there. The library's initial fill value is zero, and elf_fill lets the application change that.

Second, the application can generate its own data buffers to occupy the gaps, filling the gaps with values appropriate for the section being created. A program might even use different fill values for different sections. For example, it could set text sections' bytes to no-operation instructions, while filling data section holes with zero. Using this technique, the library finds no holes to fill, because the application eliminated them.

Section and Memory Types

elf_getdata interprets sections' data according to the section type, as noted in the section header available through elf_getshdr. The following table shows the section types and how the library represents them with memory data types for the 32-bit file class. Other classes would have similar tables. By implication, the memory data types control translation by elf_xlate.

Section Type	Elf_Type	32-Bit Type
SHT_DYNAMIC	ELF_T_DYN	Elf32_Dyn
SHT_DYNSYM	ELF_T_SYM	Elf32_Sym
SHT_HASH	ELF_T_WORD	Elf32_Word
SHT_NOBITS	ELF_T_BYTE	unsigned char
SHT_NOTE	ELF_T_BYTE	unsigned char
SHT_NULL	*none*	*none*
SHT_PROGBITS	ELF_T_BYTE	unsigned char
SHT_REL	ELF_T_REL	Elf32_Rel
SHT_RELA	ELF_T_RELA	Elf32_Rela
SHT_STRTAB	ELF_T_BYTE	unsigned char
SHT_SYMTAB	ELF_T_SYM	Elf32_Sym
other	ELF_T_BYTE	unsigned char

elf_rawdata creates a buffer with type ELF_T_BYTE.

As mentioned above, the program's working version controls what structures the library creates for the application. The library similarly interprets section types according to the versions. If a section type "belongs" to a version newer than the application's working version, the library does not translate the section data. Because the application cannot know the data format in this case, the library presents an untranslated buffer of type ELF_T_BYTE, just as it would for an unrecognized section type.

A section with a special type, SHT_NOBITS, occupies no space in an object file, even when the section header indicates a non-zero size. elf_getdata and elf_rawdata "work" on such a section, setting the *data* structure to have a null buffer pointer and the type indicated above. Although no data are present, the d_size value is set to the size from the section header. When a program is creating a new section of type SHT_NOBITS, it should use elf_newdata to add data buffers to the section. These "empty" data buffers should have the d_size members set to the desired size and the d_buf members set to null.

EXAMPLE

The following fragment obtains the string table that holds section names (ignoring error checking). See elf_strptr(3E) for a variation of string table handling.

```
ehdr = elf32_getehdr(elf);
scn = elf_getscn(elf, (size_t)ehdr->e_shstrndx);
shdr = elf32_getshdr(scn);
if (shdr->sh_type != SHT_STRTAB)
{
        /* not a string table */
}
data = 0;
if ((data = elf_getdata(scn, data)) == 0 || data->d_size == 0)
{
        /* error or no data */
}
```

The e_shstrndx member in an ELF header holds the section table index of the string table. The program gets a section descriptor for that section, verifies it is a string table, and then retrieves the data. When this fragment finishes, data->d_buf points at the first byte of the string table, and data->d_size holds the string table's size in bytes.

SEE ALSO

elf(3E), elf_cntl(3E), elf_fill(3E), elf_flag(3E), elf_getehdr(3E), elf_getscn(3E), elf_getshdr(3E), elf_rawfile(3E), elf_version(3E), elf_xlate(3E).

NAME

elf_getehdr: elf32_getehdr, elf32_newehdr – retrieve class-dependent object file header

SYNOPSIS

cc [*flag* ...] *file* ... -lelf [*library* ...]

```
#include <libelf.h>

Elf32_Ehdr *elf32_getehdr(Elf *elf);

Elf32_Ehdr *elf32_newehdr(Elf *elf);
```

DESCRIPTION

For a 32-bit class file, elf32_getehdr returns a pointer to an ELF header, if one is available for the ELF descriptor *elf*. If no header exists for the descriptor, elf32_newehdr allocates a "clean" one, but it otherwise behaves the same as elf32_getehdr. It does not allocate a new header if one exists already. If no header exists (for elf_getehdr), one cannot be created (for elf_newehdr), a system error occurs, the file is not a 32-bit class file, or *elf* is null, both functions return a null pointer.

The header includes the following members.

```
unsigned char   e_ident[EI_NIDENT];
Elf32_Half      e_type;
Elf32_Half      e_machine;
Elf32_Word      e_version;
Elf32_Addr      e_entry;
Elf32_Off       e_phoff;
Elf32_Off       e_shoff;
Elf32_Word      e_flags;
Elf32_Half      e_ehsize;
Elf32_Half      e_phentsize;
Elf32_Half      e_phnum;
Elf32_Half      e_shentsize;
Elf32_Half      e_shnum;
Elf32_Half      e_shstrndx;
```

elf32_newehdr automatically sets the ELF_F_DIRTY bit [see elf_flag(3E)]. A program may use elf_getident to inspect the identification bytes from a file.

SEE ALSO

elf(3E), elf_begin(3E), elf_flag(3E), elf_getident(3E).

NAME

elf_getident – retrieve file identification data

SYNOPSIS

cc [*flag* ...] *file* ... -lelf [*library* ...]

```
#include <libelf.h>

char *elf_getident(Elf *elf, size_t *ptr);
```

DESCRIPTION

As elf(3E) explains, ELF provides a framework for various classes of files, where basic objects may have 32 bits, 64 bits, etc. To accommodate these differences, without forcing the larger sizes on smaller machines, the initial bytes in an ELF file hold identification information common to all file classes. Every ELF header's e_ident has EI_NIDENT bytes with the following interpretation.

e_ident Index	Value	Purpose
EI_MAG0 EI_MAG1 EI_MAG2 EI_MAG3	ELFMAG0 ELFMAG1 ELFMAG2 ELFMAG3	File identification
EI_CLASS	ELFCLASSNONE ELFCLASS32 ELFCLASS64	File class
EI_DATA	ELFDATANONE ELFDATA2LSB ELFDATA2MSB	Data encoding
EI_VERSION	EV_CURRENT	File version
7-15	0	Unused, set to zero

Other kinds of files [see elf_kind(3E)] also may have identification data, though they would not conform to e_ident.

elf_getident returns a pointer to the file's "initial bytes." If the library recognizes the file, a conversion from the file image to the memory image may occur. In any case, the identification bytes are guaranteed not to have been modified, though the size of the unmodified area depends on the file type. If *ptr* is non-null, the library stores the number of identification bytes in the location to which *ptr* points. If no data are present, *elf* is null, or an error occurs, the return value is a null pointer, with zero optionally stored through *ptr*.

SEE ALSO

elf(3E), elf_begin(3E), elf_getehdr(3E), elf_kind(3E), elf_rawfile(3E).

NAME

elf_getphdr: elf32_getphdr, elf32_newphdr – retrieve class-dependent program header table

SYNOPSIS

cc [*flag* ...] *file* ... -lelf [*library* ...]

```
#include <libelf.h>

Elf32_Phdr *elf32_getphdr(Elf *elf);

Elf32_Phdr *elf32_newphdr(Elf *elf, size_t count);
```

DESCRIPTION

For a 32-bit class file, elf32_getphdr returns a pointer to the program execution header table, if one is available for the ELF descriptor *elf*.

elf32_newphdr allocates a new table with *count* entries, regardless of whether one existed previously, and sets the ELF_F_DIRTY bit for the table [see elf_flag(3E)]. Specifying a zero *count* deletes an existing table. Note this behavior differs from that of elf32_newehdr [see elf32_getehdr(3E)], allowing a program to replace or delete the program header table, changing its size if necessary.

If no program header table exists, the file is not a 32-bit class file, an error occurs, or *elf* is null, both functions return a null pointer. Additionally, elf32_newphdr returns a null pointer if *count* is zero.

The table is an array of Elf32_Phdr structures, each of which includes the following members.

```
Elf32_Word      p_type;
Elf32_Off       p_offset;
Elf32_Addr      p_vaddr;
Elf32_Addr      p_paddr;
Elf32_Word      p_filesz;
Elf32_Word      p_memsz;
Elf32_Word      p_flags;
Elf32_Word      p_align;
```

The ELF header's e_phnum member tells how many entries the program header table has [see elf_getehdr(3E)]. A program may inspect this value to determine the size of an existing table; elf32_newphdr automatically sets the member's value to *count*. If the program is building a new file, it is responsible for creating the file's ELF header before creating the program header table.

SEE ALSO

elf(3E), elf_begin(3E), elf_flag(3E), elf_getehdr(3E).

NAME

elf_getscn, elf_ndxscn, elf_newscn, elf_nextscn – get section information

SYNOPSIS

cc [*flag* ...] *file* ... -lelf [*library* ...]

```
#include <libelf.h>

Elf_Scn *elf_getscn(Elf *elf, size_t index);

size_t elf_ndxscn(Elf_Scn *scn);

Elf_Scn *elf_newscn(Elf *elf);

Elf_Scn *elf_nextscn(Elf *elf, Elf_Scn *scn);
```

DESCRIPTION

These functions provide indexed and sequential access to the sections associated with the ELF descriptor *elf*. If the program is building a new file, it is responsible for creating the file's ELF header before creating sections; see elf_getehdr(3E).

elf_getscn returns a section descriptor, given an *index* into the file's section header table. Note the first "real" section has index 1. Although a program can get a section descriptor for the section whose *index* is 0 (SHN_UNDEF, the undefined section), the section has no data and the section header is "empty" (though present). If the specified section does not exist, an error occurs, or *elf* is null, elf_getscn returns a null pointer.

elf_newscn creates a new section and appends it to the list for *elf*. Because the SHN_UNDEF section is required and not "interesting" to applications, the library creates it automatically. Thus the first call to elf_newscn for an ELF descriptor with no existing sections returns a descriptor for section 1. If an error occurs or *elf* is null, elf_newscn returns a null pointer.

After creating a new section descriptor, the program can use elf_getshdr to retrieve the newly created, "clean" section header. The new section descriptor will have no associated data [see elf_getdata(3E)]. When creating a new section in this way, the library updates the e_shnum member of the ELF header and sets the ELF_F_DIRTY bit for the section [see elf_flag(3E)]. If the program is building a new file, it is responsible for creating the file's ELF header [see elf_getehdr(3E)] before creating new sections.

elf_nextscn takes an existing section descriptor, *scn*, and returns a section descriptor for the next higher section. One may use a null *scn* to obtain a section descriptor for the section whose index is 1 (skipping the section whose index is SHN_UNDEF). If no further sections are present or an error occurs, elf_nextscn returns a null pointer.

elf_ndxscn takes an existing section descriptor, *scn*, and returns its section table index. If *scn* is null or an error occurs, elf_ndxscn returns SHN_UNDEF.

EXAMPLE

An example of sequential access appears below. Each pass through the loop processes the next section in the file; the loop terminates when all sections have been processed.

```
        scn = 0;
        while ((scn = elf_nextscn(elf, scn)) != 0)
        {
                /* process section */
        }
```

SEE ALSO

elf(3E), elf_begin(3E), elf_flag(3E), elf_getdata(3E), elf_getehdr(3E), elf_getshdr(3E).

NAME

elf_getshdr: elf32_getshdr – retrieve class-dependent section header

SYNOPSIS

cc [*flag* ...] *file* ... -lelf [*library* ...]

```
#include <libelf.h>

Elf32_Shdr *elf32_getshdr(Elf_Scn *scn);
```

DESCRIPTION

For a 32-bit class file, elf32_getshdr returns a pointer to a section header for the section descriptor *scn*. Otherwise, the file is not a 32-bit class file, *scn* was null, or an error occurred; elf32_getshdr then returns null.

The header includes the following members.

```
Elf32_Word      sh_name;
Elf32_Word      sh_type;
Elf32_Word      sh_flags;
Elf32_Addr      sh_addr;
Elf32_Off       sh_offset;
Elf32_Word      sh_size;
Elf32_Word      sh_link;
Elf32_Word      sh_info;
Elf32_Word      sh_addralign;
Elf32_Word      sh_entsize;
```

If the program is building a new file, it is responsible for creating the file's ELF header before creating sections.

SEE ALSO

elf(3E), elf_flag(3E), elf_getscn(3E), elf_strptr(3E).

NAME

elf_hash – compute hash value

SYNOPSIS

cc [*flag* ...] *file* ... -lelf [*library* ...]

```
#include <libelf.h>
```

```
unsigned long elf_hash(const char *name);
```

DESCRIPTION

elf_hash computes a hash value, given a null terminated string, *name*. The returned hash value, *h*, can be used as a bucket index, typically after computing *h* mod *x* to ensure appropriate bounds.

Hash tables may be built on one machine and used on another because elf_hash uses unsigned arithmetic to avoid possible differences in various machines' signed arithmetic. Although *name* is shown as char* above, elf_hash treats it as unsigned char* to avoid sign extension differences. Using char* eliminates type conflicts with expressions such as elf_hash("name").

ELF files' symbol hash tables are computed using this function [see elf_getdata(3E) and elf_xlate(3E)]. The hash value returned is guaranteed not to be the bit pattern of all ones (~0UL).

SEE ALSO

elf(3E), elf_getdata(3E), elf_xlate(3E).

NAME

elf_kind – determine file type

SYNOPSIS

cc [*flag* ...] *file* ... -lelf [*library* ...]

```
#include <libelf.h>

Elf_Kind elf_kind(Elf *elf);
```

DESCRIPTION

This function returns a value identifying the kind of file associated with an ELF descriptor (*elf*). Currently defined values appear below.

ELF_K_AR The file is an archive [see ar(4)]. An ELF descriptor may also be associated with an archive *member*, not the archive itself, and then elf_kind identifies the member's type.

ELF_K_COFF The file is a COFF object file. elf_begin(3E) describes the library's handling for COFF files.

ELF_K_ELF The file is an ELF file. The program may use elf_getident to determine the class. Other functions, such as elf_getehdr, are available to retrieve other file information.

ELF_K_NONE This indicates a kind of file unknown to the library.

Other values are reserved, to be assigned as needed to new kinds of files. *elf* should be a value previously returned by elf_begin. A null pointer is allowed, to simplify error handling, and causes elf_kind to return ELF_K_NONE.

SEE ALSO

elf(3E), elf_begin(3E), elf_getehdr(3E), elf_getident(3E), ar(4).

NAME

elf_next – sequential archive member access

SYNOPSIS

cc [*flag* ...] *file* ... -lelf [*library* ...]

```
#include <libelf.h>
```

```
Elf_Cmd elf_next(Elf *elf);
```

DESCRIPTION

elf_next, elf_rand, and elf_begin manipulate simple object files and archives. *elf* is an ELF descriptor previously returned from elf_begin.

elf_next provides sequential access to the next archive member. That is, having an ELF descriptor, *elf*, associated with an archive member, elf_next prepares the containing archive to access the following member when the program calls elf_begin. After successfully positioning an archive for the next member, elf_next returns the value ELF_C_READ. Otherwise, the open file was not an archive, *elf* was null, or an error occurred, and the return value is ELF_C_NULL. In either case, the return value may be passed as an argument to elf_begin, specifying the appropriate action.

SEE ALSO

elf(3E), elf_begin(3E), elf_getarsym(3E), elf_rand(3E), ar(4).

NAME

elf_rand – random archive member access

SYNOPSIS

cc [*flag* ...] *file* ... -lelf [*library* ...]

```
#include <libelf.h>

size_t elf_rand(Elf *elf, size_t offset);
```

DESCRIPTION

elf_rand, elf_next, and elf_begin manipulate simple object files and archives. *elf* is an ELF descriptor previously returned from elf_begin.

elf_rand provides random archive processing, preparing *elf* to access an arbitrary archive member. *elf* must be a descriptor for the archive itself, not a member within the archive. *offset* gives the byte offset from the beginning of the archive to the archive header of the desired member. See elf_getarsym(3E) for more information about archive member offsets. When elf_rand works, it returns *offset*. Otherwise it returns 0, because an error occurred, *elf* was null, or the file was not an archive (no archive member can have a zero offset). A program may mix random and sequential archive processing.

EXAMPLE

An archive starts with a "magic string" that has SARMAG bytes; the initial archive member follows immediately. An application could thus provide the following function to rewind an archive (the function returns −1 for errors and 0 otherwise).

```
#include <ar.h>
#include <libelf.h>

int
rewindelf(Elf *elf)
{
        if (elf_rand(elf, (size_t)SARMAG) == SARMAG)
                return 0;
        return -1;
}
```

SEE ALSO

elf(3E), elf_begin(3E), elf_getarsym(3E), elf_next(3E), ar(4).

NAME

elf_rawfile – retrieve uninterpreted file contents

SYNOPSIS

cc [*flag* ...] *file* ... -lelf [*library* ...]

```
#include <libelf.h>

char *elf_rawfile(Elf *elf, size_t *ptr);
```

DESCRIPTION

elf_rawfile returns a pointer to an uninterpreted byte image of the file. This function should be used only to retrieve a file being read. For example, a program might use elf_rawfile to retrieve the bytes for an archive member.

A program may not close or disable [see elf_cntl(3E)] the file descriptor associated with *elf* before the initial call to elf_rawfile, because elf_rawfile might have to read the data from the file if it does not already have the original bytes in memory. Generally, this function is more efficient for unknown file types than for object files. The library implicitly translates object files in memory, while it leaves unknown files unmodified. Thus asking for the uninterpreted image of an object file may create a duplicate copy in memory.

elf_rawdata [see elf_getdata(3E)] is a related function, providing access to sections within a file.

If *ptr* is non-null, the library also stores the file's size, in bytes, in the location to which *ptr* points. If no data are present, *elf* is null, or an error occurs, the return value is a null pointer, with zero optionally stored through *ptr*.

SEE ALSO

elf(3E), elf_begin(3E), elf_cntl(3E), elf_getdata(3E), elf_getehdr(3E), elf_getident(3E), elf_kind(3E).

NOTE

A program that uses elf_rawfile and that also interprets the same file as an object file potentially has two copies of the bytes in memory. If such a program requests the raw image first, before it asks for translated information (through such functions as elf_getehdr, elf_getdata, and so on), the library "freezes" its original memory copy for the raw image. It then uses this frozen copy as the source for creating translated objects, without reading the file again. Consequently, the application should view the raw file image returned by elf_rawfile as a read-only buffer, unless it wants to alter its own view of data subsequently translated. In any case, the application may alter the translated objects without changing bytes visible in the raw image.

Multiple calls to elf_rawfile with the same ELF descriptor return the same value; the library does not create duplicate copies of the file.

NAME

elf_strptr − make a string pointer

SYNOPSIS

cc [*flag* ...] *file* ... -lelf [*library* ...]

```
#include <libelf.h>

char *elf_strptr(Elf *elf, size_t section, size_t offset);
```

DESCRIPTION

This function converts a string section *offset* to a string pointer. *elf* identifies the file in which the string section resides, and *section* gives the section table index for the strings. elf_strptr normally returns a pointer to a string, but it returns a null pointer when *elf* is null, *section* is invalid or is not a section of type SHT_STRTAB, the section data cannot be obtained, *offset* is invalid, or an error occurs.

EXAMPLE

A prototype for retrieving section names appears below. The file header specifies the section name string table in the e_shstrndx member. The following code loops through the sections, printing their names.

```
if ((ehdr = elf32_getehdr(elf)) == 0)
{
        /* handle the error */
        return;
}
ndx = ehdr->e_shstrndx;
scn = 0;
while ((scn = elf_nextscn(elf, scn)) != 0)
{
        char    *name = 0;
        if ((shdr = elf32_getshdr(scn)) != 0)
          name = elf_strptr(elf, ndx, (size_t)shdr->sh_name);
        printf("'%s'\n", name? name: "(null)");
}
```

SEE ALSO

elf(3E), elf_getdata(3E), elf_getshdr(3E), elf_xlate(3E).

NOTE

A program may call elf_getdata to retrieve an entire string table section. For some applications, that would be both more efficient and more convenient than using elf_strptr.

NAME

elf_update – update an ELF descriptor

SYNOPSIS

cc [*flag* ...] *file* ... -lelf [*library* ...]

```
#include <libelf.h>
```

```
off_t elf_update(Elf *elf, Elf_Cmd cmd);
```

DESCRIPTION

elf_update causes the library to examine the information associated with an ELF descriptor, *elf*, and to recalculate the structural data needed to generate the file's image.

cmd may have the following values.

ELF_C_NULL
This value tells elf_update to recalculate various values, updating only the ELF descriptor's memory structures. Any modified structures are flagged with the ELF_F_DIRTY bit. A program thus can update the structural information and then reexamine them without changing the file associated with the ELF descriptor. Because this does not change the file, the ELF descriptor may allow reading, writing, or both reading and writing [see elf_begin(3E)].

ELF_C_WRITE
If *cmd* has this value, elf_update duplicates its ELF_C_NULL actions and also writes any "dirty" information associated with the ELF descriptor to the file. That is, when a program has used elf_getdata or the elf_flag facilities to supply new (or update existing) information for an ELF descriptor, those data will be examined, coordinated, translated if necessary [see elf_xlate(3E)], and written to the file. When portions of the file are written, any ELF_F_DIRTY bits are reset, indicating those items no longer need to be written to the file [see elf_flag(3E)]. The sections' data are written in the order of their section header entries, and the section header table is written to the end of the file.

When the ELF descriptor was created with elf_begin, it must have allowed writing the file. That is, the elf_begin command must have been either ELF_C_RDWR or ELF_C_WRITE.

If elf_update succeeds, it returns the total size of the file image (not the memory image), in bytes. Otherwise an error occurred, and the function returns -1.

When updating the internal structures, elf_update sets some members itself. Members listed below are the application's responsibility and retain the values given by the program.

	Member	Notes
ELF Header	e_ident[EI_DATA]	Library controls other e_ident values
	e_type	
	e_machine	
	e_version	
	e_entry	
	e_phoff	Only when ELF_F_LAYOUT asserted
	e_shoff	Only when ELF_F_LAYOUT asserted
	e_flags	
	e_shstrndx	

	Member	Notes
Program Header	p_type	The application controls all program header entries
	p_offset	
	p_vaddr	
	p_paddr	
	p_filesz	
	p_memsz	
	p_flags	
	p_align	

	Member	Notes
Section Header	sh_name	
	sh_type	
	sh_flags	
	sh_addr	
	sh_offset	Only when ELF_F_LAYOUT asserted
	sh_size	Only when ELF_F_LAYOUT asserted
	sh_link	
	sh_info	
	sh_addralign	Only when ELF_F_LAYOUT asserted
	sh_entsize	

	Member	Notes
Data Descriptor	d_buf	
	d_type	
	d_size	
	d_off	Only when ELF_F_LAYOUT asserted
	d_align	
	d_version	

Note that the program is responsible for two particularly important members (among others) in the ELF header. The e_version member controls the version of data structures written to the file. If the version is EV_NONE, the library uses its own internal version. The e_ident[EI_DATA] entry controls the data encoding used in the file. As a special case, the value may be ELFDATANONE to request the native data encoding for the host machine. An error occurs in this case if the native encoding doesn't match a file encoding known by the library.

Further note that the program is responsible for the sh_entsize section header member. Although the library sets it for sections with known types, it cannot reliably know the correct value for all sections. Consequently, the library relies on the program to provide the values for unknown section type. If the entry size is unknown or not applicable, the value should be set to zero.

When deciding how to build the output file, elf_update obeys the alignments of individual data buffers to create output sections. A section's most strictly aligned data buffer controls the section's alignment. The library also inserts padding between buffers, as necessary, to ensure the proper alignment of each buffer.

SEE ALSO

elf(3E), elf_begin(3E), elf_flag(3E), elf_fsize(3E), elf_getdata(3E), elf_getehdr(3E), elf_getshdr(3E), elf_xlate(3E).

NOTE

As mentioned above, the ELF_C_WRITE command translates data as necessary, before writing them to the file. This translation is *not* always transparent to the application program. If a program has obtained pointers to data associated with a file [for example, see elf_getehdr(3E) and elf_getdata(3E)], the program should reestablish the pointers after calling elf_update.

As elf_begin(3E) describes, a program may "update" a COFF file to make the image consistent for ELF . The ELF_C_NULL command updates only the memory image; one can use the ELF_C_WRITE command to modify the file as well. Absolute executable files (a.out files) require special alignment, which cannot normally be preserved between COFF and ELF . Consequently, one may not update an executable COFF file with the ELF_C_WRITE command (though ELF_C_NULL is allowed).

NAME

elf_version – coordinate ELF library and application versions

SYNOPSIS

cc [*flag* ...] *file* ... -lelf [*library* ...]

```
#include <libelf.h>

unsigned elf_version(unsigned ver);
```

DESCRIPTION

As elf(3E) explains, the program, the library, and an object file have independent notions of the "latest" ELF version. elf_version lets a program determine the ELF library's *internal version*. It further lets the program specify what memory types it uses by giving its own *working version*, *ver*, to the library. Every program that uses the ELF library must coordinate versions as described below.

The header file <libelf.h> supplies the version to the program with the macro EV_CURRENT. If the library's internal version (the highest version known to the library) is lower than that known by the program itself, the library may lack semantic knowledge assumed by the program. Accordingly, elf_version will not accept a working version unknown to the library.

Passing *ver* equal to EV_NONE causes elf_version to return the library's internal version, without altering the working version. If *ver* is a version known to the library, elf_version returns the previous (or initial) working version number. Otherwise, the working version remains unchanged and elf_version returns EV_NONE.

EXAMPLE

The following excerpt from an application program protects itself from using an older library.

```
if (elf_version(EV_CURRENT) == EV_NONE)
{
        /* library out of date */
        /* recover from error */
}
```

NOTES

The working version should be the same for all operations on a particular elf descriptor. Changing the version between operations on a descriptor will probably not give the expected results.

SEE ALSO

elf(3E), elf_begin(3E), elf_xlate(3E).

NAME

elf_xlate: elf32_xlatetof, elf32_xlatetom – class-dependent data translation

SYNOPSIS

cc [*flag* ...] *file* ... -lelf [*library* ...]

```
#include <libelf.h>
```

```
Elf_Data *elf32_xlatetof(Elf_Data *dst, const Elf_Data *src,
      unsigned encode);
```

```
Elf_Data *elf32_xlatetom(Elf_Data *dst, const Elf_Data *src,
      unsigned encode);
```

DESCRIPTION

elf32_xlatetom translates various data structures from their 32-bit class file representations to their memory representations; elf32_xlatetof provides the inverse. This conversion is particularly important for cross development environments. *src* is a pointer to the source buffer that holds the original data; *dst* is a pointer to a destination buffer that will hold the translated copy. *encode* gives the byte encoding in which the file objects are (to be) represented and must have one of the encoding values defined for the ELF header's e_ident[EI_DATA] entry [see elf_getident(3E)]. If the data can be translated, the functions return *dst*. Otherwise, they return null because an error occurred, such as incompatible types, destination buffer overflow, etc.

elf_getdata(3E) describes the Elf_Data descriptor, which the translation routines use as follows.

d_buf Both the source and destination must have valid buffer pointers.

d_type This member's value specifies the type of the data to which d_buf points and the type of data to be created in the destination. The program supplies a d_type value in the source; the library sets the destination's d_type to the same value. These values are summarized below.

d_size This member holds the total size, in bytes, of the memory occupied by the source data and the size allocated for the destination data. If the destination buffer is not large enough, the routines do not change its original contents. The translation routines reset the destination's d_size member to the actual size required, after the translation occurs. The source and destination sizes may differ.

d_version This member holds version number of the objects (desired) in the buffer. The source and destination versions are independent.

Translation routines allow the source and destination buffers to coincide. That is, dst->d_buf may equal src->d_buf. Other cases where the source and destination buffers overlap give undefined behavior.

Elf_Type	32-Bit Memory Type
ELF_T_ADDR	Elf32_Addr
ELF_T_BYTE	unsigned char
ELF_T_DYN	Elf32_Dyn
ELF_T_EHDR	Elf32_Ehdr
ELF_T_HALF	Elf32_Half
ELT_T_OFF	Elf32_Off
ELF_T_PHDR	Elf32_Phdr
ELF_T_REL	Elf32_Rel
ELF_T_RELA	Elf32_Rela
ELF_T_SHDR	Elf32_Shdr
ELF_T_SWORD	Elf32_Sword
ELF_T_SYM	Elf32_Sym
ELF_T_WORD	Elf32_Word

"Translating" buffers of type ELF_T_BYTE does not change the byte order.

SEE ALSO

elf(3E), elf_fsize(3E), elf_getdata(3E), elf_getident(3E).

NAME

end, etext, edata – last locations in program

SYNOPSIS

```
extern etext;
extern _etext;
extern __etext;

extern edata;
extern _edata;
extern __edata;

extern end;
extern _end;
extern __end;
```

DESCRIPTION

These names refer neither to routines nor to locations with interesting contents; only their addresses are meaningful.

etext, _etext, __etext
: The address of these symbols is the first address above the program text.

edata, _edata, __edata
: The address of these symbols is the first address above the initialized data region.

end, _end, __end
: The address of these symbols is the first address above the uninitialized data region.

SEE ALSO

cc(1), brk(2), malloc(3C), stdio(3S).

NOTE

When execution begins, the program break (the first location beyond the data) coincides with end, but the program break may be reset by the routines brk and malloc, by standard input/output routines [see stdio(3S)], by the profile (-p) option of cc, and so on. Thus, the current value of the program break should be determined by sbrk ((char *)0) [see brk(2)].

NAME

erf, erfc – error function and complementary error function

SYNOPSIS

cc [*flag* ...] *file* ... -lm [*library* ...]

```
#include <math.h>

double erf (double x);

double erfc (double x);
```

DESCRIPTION

erf returns the error function of x, defined as

$$\frac{2}{\sqrt{\pi}}\int_{0}^{x} e^{-t^2}dt$$

erfc, which returns 1.0 – erf(x), is provided because of the extreme loss of relative accuracy if erf(x) is called for large x and the result subtracted from 1.0 (e.g., for $x = 5$, 12 places are lost).

SEE ALSO

exp(3M).

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`(4).

FILES

`/etc/ethers` (or the Network Information Services maps `ethers.byaddr` and `ethers.byname`)

SEE ALSO

`ethers`(4).

NAME

exp, expf, cbrt, log, logf, log10, log10f, pow, powf, sqrt, sqrtf – exponential, logarithm, power, square root functions

SYNOPSIS

cc [*flag* ...] *file* ... -lm [*library* ...]

```
#include <math.h>

double exp (double x);

float expf (float x);

double cbrt (double x);

double log (double x);

float logf (float x);

double log10 (double x);

float log10f (float x);

double pow (double x, double y);

float powf (float x, float y);

double sqrt (double x);

float sqrtf (float x);
```

DESCRIPTION

exp and expf return e^x.

cbrt returns the cube root of x.

log and logf return the natural logarithm of x. The value of x must be positive.

log10 and log10f return the base ten logarithm of x. The value of x must be positive.

pow and powf return x^y. If x is 0, y must be positive. If x is negative, y must be an integer.

sqrt and sqrtf return the non-negative square root of x. The value of x may not be negative.

DIAGNOSTICS

exp and expf return HUGE when the correct value would overflow, or 0 when the correct value would underflow, and set errno to ERANGE.

log, logf, log10, and log10f return -HUGE and set errno to EDOM when x is non-positive. A message indicating DOMAIN error is printed on standard error.

pow and powf return 0 and set errno to EDOM when x is 0 and y is non-positive, or when x is negative and y is not an integer. In these cases, a message indicating DOMAIN error is printed on standard error. When the correct value for pow or powf would overflow or underflow, these functions return ±HUGE or 0, respectively, and set errno to ERANGE.

sqrt and sqrtf return 0 and set errno to EDOM when x is negative. A message indicating DOMAIN error is printed on standard error.

Except when the -Xc compilation option is used, these error-handling procedures may be changed with the function matherr. When the -Xa or -Xc compilation options are used, HUGE_VAL is returned instead of HUGE and no error messages are printed. In these compilation modes, pow and powf return 1, with no error, when both *x* and *y* are 0; when *x* is 0 and *y* is negative, they return -HUGE_VAL and set errno to EDOM. Under -Xc, log and logf return -HUGE_VAL and set errno to ERANGE when *x* is 0. Under -Xc, sqrt and sqrtf return NaN when *x* is negative.

SEE ALSO

hypot(3M), matherr(3M), sinh(3M).

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 (creating it if it does not already exist) and returns a file pointer to use with getexportent, addexportent, remexportent, and endexportent. You must be superuser to call setexportent. 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:

```
#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 */
};
```

For more information about the fields, see exports(5).

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.

ACCESS

You must be superuser to execute setexportent.

FILES

```
/etc/exports
/etc/xtab
```

SEE ALSO

`exports`(5), `xtab`(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

extended_perror – print an error message to standard error

SYNOPSIS

```
void extended_perror();
char *string;
. . .
extended_perror(string);
```

where:

string A pointer to a null-terminated string that is printed (along with a colon) before the error message is printed

DESCRIPTION

The extended_perror function prints to the standard error file a string and an error message corresponding to the last DG/UX extended error with the dg_ext_errno function. If no extended error message is available, nothing is printed.

RETURNS

The extended_perror function returns no value.

EXAMPLE

The following program copies data from the tape drive to the standard output. If an error occurs, the extended_perror function prints an error message.

```
/* Program test for the extended_perror() function */

#include <fcntl.h>


#define MBUF   32768
#define STDOUT 1

char buf[MBUF];
void perror(), extended_perror();
int read(), write();

main() {
    int len, fd = open("/dev/rmt/0", O_RDONLY);

    if (fd == -1) {
        perror("Open of /dev/rmt/0");
        return 1;
    }
    while ((len = read(fd, buf, MBUF)) > 0) {
        if (write(STDOUT, buf, len) < 0) {
            perror("Write to stdout");
            return 1;
        }
    }
    if (len < 0) {
        extended_perror("Read of /dev/rmt/0");
        return 1;
    }
```

```
        return 0;
    }
```

FILES

/usr/lib/nls/msg/*locale*/exterr.cat — message catalog.

SEE ALSO

See also the perror function.
extended_strerror(3C).

NAME

extended_strerror – get extended error message string

SYNOPSIS

```
char *extended_strerror (int extended_errnum);
```

DESCRIPTION

extended_strerror maps the error number in *extended_errnum* to an error message string, and returns a pointer to that string. extended_strerror uses the same set of error messages as extended_perror. The returned string should not be overwritten.

Many system calls return an extended error number in dg_ext_errno.

FILES

/usr/lib/nls/msg/*locale*/exterr.cat — message catalog.

SEE ALSO

extended_perror(3C), strerror(3C).

NAME

`fattach` – attach STREAMS-based file descriptor to object in file system name space

SYNOPSIS

```
#include <unistd.h>

int fattach(int fildes, const char *path)
```

DESCRIPTION

The `fattach` routine attaches a STREAMS-based file descriptor to an object in the file system name space, effectively associating a name with *fildes*. *fildes* must be a valid open file descriptor representing a STREAMS file. *path* is a path name of an existing object and the user must have appropriate privileges or be the owner of the file and have write permissions. All subsequent operations on *path* will operate on the STREAMS file until the STREAMS file is detached from the node. *fildes* can be attached to more than one *path*, i.e., a stream can have several names associated with it.

The attributes of the named stream [see `stat(2)`], are initialized as follows: the permissions, user ID, group ID, and times are set to those of *path*, the number of links is set to 1, and the size and device identifier are set to those of the streams device associated with *fildes*. If any attributes of the named stream are subsequently changed [e.g., `chmod(2)`], the attributes of the underlying object are not affected.

RETURN VALUE

If successful, `fattach` returns 0; otherwise it returns -1 and sets `errno` to indicate an error.

DIAGNOSTICS

Under the following conditions, the function `fattach` fails and sets `errno` to:

`EACCES` The user is the owner of *path* but does not have write permissions on *path* or *fildes* is locked.

`EBADF` *fildes* is not a valid open file descriptor.

`ENOENT` *path* does not exist.

`ENOTDIR` A component of a path prefix is not a directory.

`EINVAL` *fildes* does not represent a STREAMS file.

`EPERM` The effective user ID is not the owner of *path* or a user with the appropriate privileges.

`EBUSY` *path* is currently a mount point or has a STREAMS file descriptor attached it.

`ENAMETOOLONG`
The size of *path* exceeds `{PATH_MAX}`, or the component of a path name is longer than `{NAME_MAX}` while `{_POSIX_NO_TRUNC}` is in effect.

`ELOOP` Too many symbolic links were encountered in translating *path*.

`EREMOTE` *path* is a file in a remotely mounted directory.

SEE ALSO

`fdetach(1M)`,`fdetach(3C)`, `isastream(3C)`, `streamio(7)`
in the *Programmer's Guide: STREAMS*

NAME

fclose, fflush – close or flush a stream

SYNOPSIS

```
#include <stdio.h>

int fclose (FILE *stream);

int fflush (FILE *stream);
```

DESCRIPTION

fclose causes any buffered data waiting to be written for the named *stream* [see intro(3)] to be written out, and the *stream* to be closed. If the underlying file pointer is not already at end of file, and the file is one capable of seeking, the file pointer is adjusted so that the next operation on the open file pointer deals with the byte after the last one read from or written to the file being closed.

fclose is performed automatically for all open files upon calling exit.

If *stream* points to an output stream or an update stream on which the most recent operation was not input, fflush causes any buffered data waiting to be written for the named *stream* to be written to that file. Any unread data buffered in *stream* is discarded. The *stream* remains open. If *stream* is open for reading, the underlying file pointer is not already at end of file, and the file is one capable of seeking, the file pointer is adjusted so that the next operation on the open file pointer deals with the byte after the last one read from or written to the stream.

When calling fflush, if *stream* is a null pointer, all files open for writing are flushed.

SEE ALSO

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

DIAGNOSTICS

Upon successful completion these functions return a value of zero. Otherwise EOF is returned.

NAME

fdetach – detach a name from a STREAMS-based file descriptor

SYNOPSIS

```
#include <unistd.h>

int fdetach(const char *path)
```

DESCRIPTION

The fdetach routine detaches a STREAMS-based file descriptor from a name in the file system. *path* is the path name of the object in the file system name space, which was previously attached [see fattach(3C)]. The user must be the owner of the file or a user with the appropriate privileges. All subsequent operations on *path* will operate on the file system node and not on the STREAMS file. The permissions and status of the node are restored to the state the node was in before the STREAMS file was attached to it.

RETURN VALUE

If successful, fdetach returns 0; otherwise it returns -1 and sets errno to indicate an error.

DIAGNOSTICS

Under the following conditions, the function fdetach fails and sets errno to:

EPERM — The effective user ID is not the owner of *path* or is not a user with appropriate permissions.

ENOTDIR — A component of the path prefix is not a directory.

ENOENT — *path* does not exist.

EINVAL — *path* is not attached to a STREAMS file.

ENAMETOOLONG — The size of *path* exceeds {PATH_MAX}, or a path name component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

ELOOP — Too many symbolic links were encountered in translating *path*.

SEE ALSO

fdetach(1M), fattach(3C), streamio(7).
in the *Programmer's Guide: STREAMS*

NAME

ferror, feof, clearerr, fileno – stream status inquiries

SYNOPSIS

```
#include <stdio.h>

int ferror (FILE *stream);

int feof (FILE *stream);

void clearerr (FILE *stream);

int fileno (FILE *stream);
```

DESCRIPTION

ferror returns non-zero when an error has previously occurred reading from or writing to the named *stream* [see intro(3)], otherwise zero.

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

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

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

SEE ALSO

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

NAME

ffs – find first set bit

SYNOPSIS

```
#include <string.h>

int ffs(const int i);
```

DESCRIPTION

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 low order bit. A return value of zero indicates that the value passed is zero.

SEE ALSO

string(3C).

NAME

floor, floorf, ceil, ceilf, copysign, fmod, fmodf, fabs, fabsf, rint, remainder – floor, ceiling, remainder, absolute value functions

SYNOPSIS

cc [*flag* ...] *file* ... -lm [*library* ...]

```
#include <math.h>

double floor (double x);

float floorf (float x);

double ceil (double x);

float ceilf (float x);

double copysign (double x, double y);

double fmod (double x, double y);

float fmodf (float x, float y);

double fabs (double x);

float fabsf (float x);

double rint (double x);

double remainder (double x, double y);
```

DESCRIPTION

floor and floorf return the largest integer not greater than *x*. ceil and ceilf return the smallest integer not less than *x*.

copysign returns *x* but with the sign of *y*.

fmod and fmodf return the floating point remainder of the division of *x* by *y*. More precisely, they return the number *f* with the same sign as *x*, such that $x = iy + f$ for some integer *i*, and $|f| < |y|$.

fabs and fabsf return the absolute value of *x*, $|x|$.

rint returns the nearest integer value to its floating point argument *x* as a double-precision floating point number. The returned value is rounded according to the currently set machine rounding mode. If round-to-nearest (the default mode) is set and the difference between the function argument and the rounded result is exactly 0.5, then the result will be rounded to the nearest even integer.

remainder returns the floating point remainder of the division of *x* by *y*. More precisely, it returns the value $r = x - yn$, where *n* is the integer nearest the exact value x/y. Whenever $|n - x/y| = 1/2$, then *n* is even.

SEE ALSO

abs(3C), matherr(3M).

DIAGNOSTICS

fmod and fmodf return *x* when *y* is 0 and set errno to EDOM. remainder returns NaN when *y* is 0 and sets errno to EDOM. In both cases, except in compilation modes -Xa or -Xc, a message indicating DOMAIN error is printed on standard error. Except under -Xc, these error-handling procedures may be changed with the function matherr.

NAME

fmtmsg – display a message on stderr or system console

SYNOPSIS

```
#include <fmtmsg.h>

int fmtmsg(long classification, const  char *label, int severity,
    const char *text, const char *action, const char *tag);
```

DESCRIPTION

Based on a message's classification component, fmtmsg writes a formatted message to stderr, to the console, or to both.

fmtmsg can be used instead of the traditional printf interface to display messages to stderr. fmtmsg, in conjunction with gettxt, provides a simple interface for producing language-independent applications.

A formatted message consists of up to five standard components as defined below. The component, *classification*, is not part of the standard message displayed to the user, but rather defines the source of the message and directs the display of the formatted message.

classification
Contains identifiers from the following groups of major classifications and subclassifications. Any one identifier from a subclass may be used in combination by ORing the values together with a single identifier from a different subclass. Two or more identifiers from the same subclass should not be used together, with the exception of identifiers from the display subclass. (Both display subclass identifiers may be used so that messages can be displayed to both stderr and the system console).

"Major classifications" identify the source of the condition. Identifiers are: MM_HARD (hardware), MM_SOFT (software), and MM_FIRM (firmware).

"Message source subclassifications" identify the type of software in which the problem is spotted. Identifiers are: MM_APPL (application), MM_UTIL (utility), and MM_OPSYS (operating system).

"Display subclassifications" indicate where the message is to be displayed. Identifiers are: MM_PRINT to display the message on the standard error stream, MM_CONSOLE to display the message on the system console. Neither, either, or both identifiers may be used.

"Status subclassifications" indicate whether the application will recover from the condition. Identifiers are: MM_RECOVER (recoverable) and MM_NRECOV (non-recoverable).

An additional identifier, MM_NULLMC, indicates that no classification component is supplied for the message.

label Identifies the source of the message. The format of this component is two fields separated by a colon. The first field is up to 10 characters long; the second is up to 14 characters. Suggested usage is that *label* identifies the package in which the application resides as well as the program or application name. For example, the *label* UX:cat indicates the UNIX System V package and the cat application.

severity
Indicates the seriousness of the condition. Identifiers for the standard levels of

severity are:

MM_HALT indicates that the application has encountered a severe fault and is halting. Produces the print string **HALT**.

MM_ERROR indicates that the application has detected a fault. Produces the print string **ERROR**.

MM_WARNING indicates a condition out of the ordinary that might be a problem and should be watched. Produces the print string **WARNING**.

MM_INFO provides information about a condition that is not in error. Produces the print string **INFO**.

MM_NOSEV indicates that no severity level is supplied for the message.

Other severity levels may be added by using the addseverity routine.

text Describes the condition that produced the message. The *text* string is not limited to a specific size.

action
Describes the first step to be taken in the error recovery process. fmtmsg precedes each action string with the prefix: **TO FIX:**. The *action* string is not limited to a specific size.

tag An identifier which references on-line documentation for the message. Suggested usage is that *tag* includes the *label* and a unique identifying number. A sample *tag* is UX:cat:146.

Environment Variables

There are two environment variables that control the behavior of fmtmsg: MSGVERB and SEV_LEVEL.

MSGVERB tells fmtmsg which message components it is to select when writing messages to stderr. The value of MSGVERB is a colon-separated list of optional keywords. MSGVERB can be set as follows:

```
MSGVERB=[keyword[:keyword[:...]]]
export MSGVERB
```

Valid *keywords* are: label, severity, text, action, and tag. If MSGVERB contains a keyword for a component and the component's value is not the component's null value, fmtmsg includes that component in the message when writing the message to stderr. If MSGVERB does not include a keyword for a message component, that component is not included in the display of the message. The keywords may appear in any order. If MSGVERB is not defined, if its value is the null-string, if its value is not of the correct format, or if it contains keywords other than the valid ones listed above, fmtmsg selects all components.

The first time fmtmsg is called, it examines the MSGVERB environment variable to see which message components it is to select when generating a message to write to the standard error stream, stderr. The values accepted on the initial call are saved for future calls.

MSGVERB affects only which components are selected for display to the standard error stream. All message components are included in console messages.

SEV_LEVEL defines severity levels and associates print strings with them for use by fmtmsg. The standard severity levels shown below cannot be modified. Additional severity levels can also be defined, redefined, and removed using addseverity [see addseverity(3C)]. If the same severity level is defined by both SEV_LEVEL and addseverity, the definition by addseverity is controlling.

0	(no severity is used)
1	**HALT**
2	**ERROR**
3	**WARNING**
4	**INFO**

SEV_LEVEL can be set as follows:

SEV_LEVEL=[*description*[:*description*[:...]]]
export SEV_LEVEL

description is a comma-separated list containing three fields:

description=severity_keyword,level,printstring

severity_keyword is a character string that is used as the keyword on the -s *severity* option to the fmtmsg command. (This field is not used by the fmtmsg function.)

level is a character string that evaluates to a positive integer (other than 0, 1, 2, 3, or 4, which are reserved for the standard severity levels). If the keyword *severity_keyword* is used, *level* is the severity value passed on to the fmtmsg function.

printstring is the character string used by fmtmsg in the standard message format whenever the severity value *level* is used.

If a *description* in the colon list is not a three-field comma list, or, if the second field of a comma list does not evaluate to a positive integer, that *description* in the colon list is ignored.

The first time fmtmsg is called, it examines the SEV_LEVEL environment variable, if defined, to see whether the environment expands the levels of severity beyond the five standard levels and those defined using addseverity. The values accepted on the initial call are saved for future calls.

Use in Applications

One or more message components may be systematically omitted from messages generated by an application by using the null value of the argument for that component.

The table below indicates the null values and identifiers for fmtmsg arguments.

Argument	Type	Null-Value	Identifier
label	char*	(char*) NULL	MM_NULLLBL
severity	int	0	MM_NULLSEV
class	long	0L	MM_NULLMC
text	char*	(char*) NULL	MM_NULLTXT
action	char*	(char*) NULL	MM_NULLACT
tag	char*	(char*) NULL	MM_NULLTAG

Another means of systematically omitting a component is by omitting the component keyword(s) when defining the MSGVERB environment variable (see the "Environment Variables" section).

EXAMPLES

Example 1:

The following example of fmtmsg:

```
fmtmsg(MM_PRINT, "UX:cat", MM_ERROR, "invalid syntax",
"refer to manual", "UX:cat:001")
```

produces a complete message in the standard message format:

```
UX:cat: ERROR: invalid syntax
        TO FIX: refer to manual   UX:cat:001
```

Example 2:

When the environment variable `MSGVERB` is set as follows:

```
MSGVERB=severity:text:action
```

and the Example 1 is used, `fmtmsg` produces:

```
ERROR: invalid syntax
TO FIX: refer to manual
```

Example 3:

When the environment variable `SEV_LEVEL` is set as follows:

```
SEV_LEVEL=note,5,NOTE
```

the following call to `fmtmsg`:

```
fmtmsg(MM_UTIL | MM_PRINT, "UX:cat", 5, "invalid syntax",
"refer to manual", "UX:cat:001")
```

produces:

```
UX:cat: NOTE: invalid syntax
        TO FIX: refer to manual   UX:cat:001
```

SEE ALSO

`addseverity`(3C), `gettxt`(3C), `printf`(3S).
`fmtmsg`(1) in the *User's Reference Manual.*

DIAGNOSTICS

The exit codes for `fmtmsg` are the following:

MM_OK	The function succeeded.
MM_NOTOK	The function failed completely.
MM_NOMSG	The function was unable to generate a message on the standard error stream, but otherwise succeeded.
MM_NOCON	The function was unable to generate a console message, but otherwise succeeded.

NAME

fopen, freopen, fdopen – open a stream

SYNOPSIS

```
#include <stdio.h>

FILE *fopen (const char *filename, const char *type);

FILE *freopen (const char *filename, const char *type, FILE
    *stream);

FILE *fdopen (int fildes, const char *type);
```

DESCRIPTION

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

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

type is a character string beginning with one of the following sequences:

"r" or "rb"
: open for reading

"w" or "wb"
: truncate to zero length or create for writing

"a" or "ab"
: append; open for writing at end of file, or create for writing

"r+", "r+b" or "rb+"
: open for update (reading and writing)

"w+", "w+b" or "wb+"
: truncate or create for update

"a+", "a+b" or "ab+"
: append; open or create for update at end-of-file

The "b" is ignored in the above *types*. The "b" exists to distinguish binary files from text files. However, there is no distinction between these types of files on a UNIX system.

freopen substitutes the named file in place of the open *stream*. A flush is first attempted, and then the original *stream* is closed, regardless of whether the open ultimately succeeds. Failure to flush or close *stream* successfully is ignored. freopen returns a pointer to the FILE structure associated with *stream*.

freopen is typically used to attach the preopened *streams* associated with stdin, stdout, and stderr to other files. stderr is by default unbuffered, but the use of freopen will cause it to become buffered or line-buffered.

fdopen associates a *stream* with a file descriptor. File descriptors are obtained from open, dup, creat, or pipe, which open files but do not return pointers to a FILE structure *stream*. Streams are necessary input for almost all of the Section 3S library routines. The *type* of *stream* must agree with the mode of the open file. The file position indicator associated with *stream* is set to the position indicated by the file offset associated with *fildes*.

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 fflush, fseek, fsetpos, or rewind, and input may not be directly followed by output without an intervening fseek, fsetpos, or rewind, or an input operation that encounters end-of-file.

When a file is opened for append (i.e., when *type* is "a", "ab", "a+", or "ab+"), 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.

When opened, a *stream* is fully buffered if and only if it can be determined not to refer to an interactive device. The error and end-of-file indicators are cleared for the *stream*.

SEE ALSO

close(2), creat(2), dup(2), open(2), pipe(2), write(2), fclose(3S), fseek(3S), setbuf(3S), stdio(3S).

DIAGNOSTICS

The functions fopen and freopen return a null pointer if *path* cannot be accessed, or if *type* is invalid, or if the file cannot be opened. In addition, errno is set to ENOENT if *filename* is a NULL pointer or the string it points to is a null string.

The function fdopen returns a null pointer if *fildes* is not an open file descriptor, or if *type* is invalid, or if the file cannot be opened.

The functions fopen or fdopen may fail and not set errno if there are no free stdio streams.

File descriptors used by fdopen must be less than 255.

NAME

form_cursor: pos_form_cursor – position forms window cursor

SYNOPSIS

```
#include <form.h>

int pos_form_cursor(FORM *form);
```

DESCRIPTION

pos_form_cursor moves the form window cursor to the location required by the form driver to resume form processing. This may be needed after the application calls a curses library I/O routine.

RETURN VALUE

pos_form_cursor returns one of the following:

E_OK	– The function returned successfully.
E_SYSTEM_ERROR	– System error.
E_BAD_ARGUMENT	– An argument is incorrect.
E_NOT_POSTED	– The form is not posted.

NOTES

The header file <form.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), forms(3X).

NAME

form_data: data_ahead, data_behind – tell if forms field has off-screen data ahead or behind

SYNOPSIS

```
#include <form.h>

int data_ahead(FORM *form);

int data_behind(FORM *form);
```

DESCRIPTION

data_ahead returns TRUE (1) if the current field has more off-screen data ahead; otherwise it returns FALSE (0).

data_behind returns TRUE (1) if the current field has more off-screen data behind; otherwise it returns FALSE (0).

NOTES

The header file <form.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), forms(3X).

NAME

form_driver – command processor for the forms subsystem

SYNOPSIS

```
#include <form.h>

int form_driver(FORM *form, int c);
```

DESCRIPTION

form_driver is the workhorse of the forms subsystem; it checks to determine whether the character *c* is a forms request or data. If it is a request, the form driver executes the request and reports the result. If it is data (a printable ASCII character), it enters the data into the current position in the current field. If it is not recognized, the form driver assumes it is an application-defined command and returns E_UNKNOWN_COMMAND. Application defined commands should be defined relative to MAX_COMMAND, the maximum value of a request listed below.

Form driver requests:

REQ_NEXT_PAGE	Move to the next page.
REQ_PREV_PAGE	Move to the previous page.
REQ_FIRST_PAGE	Move to the first page.
REQ_LAST_PAGE	Move to the last page.
REQ_NEXT_FIELD	Move to the next field.
REQ_PREV_FIELD	Move to the previous field.
REQ_FIRST_FIELD	Move to the first field.
REQ_LAST_FIELD	Move to the last field.
REQ_SNEXT_FIELD	Move to the sorted next field.
REQ_SPREV_FIELD	Move to the sorted prev field.
REQ_SFIRST_FIELD	Move to the sorted first field.
REQ_SLAST_FIELD	Move to the sorted last field.
REQ_LEFT_FIELD	Move left to field.
REQ_RIGHT_FIELD	Move right to field.
REQ_UP_FIELD	Move up to field.
REQ_DOWN_FIELD	Move down to field.
REQ_NEXT_CHAR	Move to the next character in the field.
REQ_PREV_CHAR	Move to the previous character in the field.
REQ_NEXT_LINE	Move to the next line in the field.
REQ_PREV_LINE	Move to the previous line in the field.
REQ_NEXT_WORD	Move to the next word in the field.
REQ_PREV_WORD	Move to the previous word in the field.
REQ_BEG_FIELD	Move to the first char in the field.
REQ_END_FIELD	Move after the last char in the field.
REQ_BEG_LINE	Move to the beginning of the line.
REQ_END_LINE	Move after the last char in the line.
REQ_LEFT_CHAR	Move left in the field.
REQ_RIGHT_CHAR	Move right in the field.
REQ_UP_CHAR	Move up in the field.
REQ_DOWN_CHAR	Move down in the field.

`REQ_NEW_LINE`	Insert/overlay a new line.
`REQ_INS_CHAR`	Insert the blank character at the cursor.
`REQ_INS_LINE`	Insert a blank line at the cursor.
`REQ_DEL_CHAR`	Delete the character at the cursor.
`REQ_DEL_PREV`	Delete the character before the cursor.
`REQ_DEL_LINE`	Delete the line at the cursor.
`REQ_DEL_WORD`	Delete the word at the cursor.
`REQ_CLR_EOL`	Clear to the end of the line.
`REQ_CLR_EOF`	Clear to the end of the field.
`REQ_CLR_FIELD`	Clear the entire field.
`REQ_OVL_MODE`	Enter overlay mode.
`REQ_INS_MODE`	Enter insert mode.

`REQ_SCR_FLINE`	Scroll the field forward a line.
`REQ_SCR_BLINE`	Scroll the field backward a line.
`REQ_SCR_FPAGE`	Scroll the field forward a page.
`REQ_SCR_BPAGE`	Scroll the field backward a page.
`REQ_SCR_FHPAGE`	Scroll the field forward half a page.
`REQ_SCR_BHPAGE`	Scroll the field backward half a page.

`REQ_SCR_FCHAR`	Horizontal scroll forward a character.
`REQ_SCR_BCHAR`	Horizontal scroll backward a character.
`REQ_SCR_HFLINE`	Horizontal scroll forward a line.
`REQ_SCR_HBLINE`	Horizontal scroll backward a line.
`REQ_SCR_HFHALF`	Horizontal scroll forward half a line.
`REQ_SCR_HBHALF`	Horizontal scroll backward half a line.

`REQ_VALIDATION`	Validate field.
`REQ_PREV_CHOICE`	Display the previous field choice.
`REQ_NEXT_CHOICE`	Display the next field choice.

RETURN VALUE

`form_driver` returns one of the following:

`E_OK`	– The function returned successfully.
`E_SYSTEM_ERROR`	– System error.
`E_BAD_ARGUMENT`	– An argument is incorrect.
`E_NOT_POSTED`	– The form is not posted.
`E_INVALID_FIELD`	– The field contents are invalid.
`E_BAD_STATE`	– The routine was called from an initialization or termination function.
`E_REQUEST_DENIED`	– The form driver request failed.
`E_UNKNOWN_COMMAND`	– An unknown request was passed to the the form driver.

NOTES

The header file `<form.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

`curses`(3X), `forms`(3X).

NAME

form_field: set_form_fields, form_fields, field_count, move_field – connect fields to forms

SYNOPSIS

```
#include <form.h>

int set_form_fields(FORM *form, FIELD **field);

FIELD **form_fields(FORM *form);

int field_count(FORM *form);

int move_field(FIELD *field, int frow, int fcol);
```

DESCRIPTION

set_form_fields changes the fields connected to *form* to *fields*. The original fields are disconnected.

form_fields returns a pointer to the field pointer array connected to *form*.

field_count returns the number of fields connected to *form*.

move_field moves the disconnected *field* to the location *frow, fcol* in the forms subwindow.

RETURN VALUE

form_fields returns NULL on error.

field_count returns -1 on error.

set_form_fields and move_field return one of the following:

E_OK	– The function returned successfully.
E_CONNECTED	– The field is already connected to a form.
E_SYSTEM_ERROR	– System error.
E_BAD_ARGUMENT	– An argument is incorrect.
E_POSTED	– The form is posted.

NOTES

The header file <form.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), forms(3X).

NAME

form_field_attributes: set_field_fore, field_fore, set_field_back, field_back, set_field_pad, field_pad – format the general display attributes of forms

SYNOPSIS

```
#include <form.h>

int set_field_fore(FIELD *field, chtype attr);
chtype field_fore(FIELD *field);

int set_field_back(FIELD *field, chtype attr);
chtype field_back(FIELD *field);

int set_field_pad(FIELD *field, int pad);
int field_pad(FIELD *field);
```

DESCRIPTION

set_field_fore sets the foreground attribute of *field*. The foreground attribute is the low-level curses display attribute used to display the field contents. field_fore returns the foreground attribute of *field*.

set_field_back sets the background attribute of *field*. The background attribute is the low-level curses display attribute used to display the extent of the field. field_back returns the background attribute of *field*.

set_field_pad sets the pad character of *field* to *pad*. The pad character is the character used to fill within the field. field_pad returns the pad character of *field*.

RETURN VALUE

field_fore, field_back and field_pad return default values if *field* is NULL. If *field* is not NULL and is not a valid FIELD pointer, the return value from these routines is undefined.

set_field_fore, set_field_back and set_field_pad return one of the following:

E_OK	– The function returned successfully.
E_SYSTEM_ERROR	– System error.
E_BAD_ARGUMENT	– An argument is incorrect.

NOTES

The header file <form.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), forms(3X).

NAME

form_field_buffer: set_field_buffer, field_buffer, set_field_status, field_status, set_max_field – set and get forms field attributes

SYNOPSIS

```
#include <form.h>

int set_field_buffer(FIELD *field, int buf, char *value);
char *field_buffer(FIELD *field, int buf);

int set_field_status(FIELD *field, int status);
int field_status(FIELD *field);

int set_max_field(FIELD *field, int max);
```

DESCRIPTION

set_field_buffer sets buffer *buf* of *field* to *value*. Buffer 0 stores the displayed contents of the field. Buffers other than 0 are application specific and not used by the forms library routines. field_buffer returns the value of *field* buffer *buf*.

Every field has an associated status flag that is set whenever the contents of field buffer 0 changes. set_field_status sets the status flag of *field* to *status*. field_status returns the status of *field*.

set_max_field sets a maximum growth on a dynamic field, or if *max*=0 turns off any maximum growth.

RETURN VALUE

field_buffer returns NULL on error.

field_status returns TRUE or FALSE.

set_field_buffer, set_field_status and set_max_field return one of the following:

E_OK	– The function returned successfully.
E_SYSTEM_ERROR	– System error.
E_BAD_ARGUMENT	– An argument is incorrect.

NOTES

The header file <form.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), forms(3X).

NAME

form_field_info: field_info, dynamic_field_info – get forms field characteristics

SYNOPSIS

```
#include <form.h>

int field_info(FIELD *field, int *rows, int *cols,
      int *frow, int *fcol, int *nrow, int *nbuf);

int dynamic_field_info(FIELD *field, int *drows, int *dcols,
      int *max);
```

DESCRIPTION

field_info returns the size, position, and other named field characteristics, as defined in the original call to new_field, to the locations pointed to by the arguments *rows*, *cols*, *frow*, *fcol*, *nrow*, and *nbuf*.

dynamic_field_info returns the actual size of the *field* in the pointer arguments *drows*, *dcols* and returns the maximum growth allowed for *field* in *max*. If no maximum growth limit is specified for *field*, *max* will contain 0. A field can be made dynamic by turning off the field option O_STATIC.

RETURN VALUE

These routines return one of the following:

E_OK	–	The function returned successfully.
E_SYSTEM_ERROR	–	System error.
E_BAD_ARGUMENT	–	An argument is incorrect.

NOTES

The header file <form.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), forms(3X).

NAME

form_field_just: set_field_just, field_just – format the general appearance of forms

SYNOPSIS

```
#include <form.h>

int set_field_just(FIELD *field, int justification);

int field_just(FIELD *field);
```

DESCRIPTION

set_field_just sets the justification for *field*. Justification may be one of:

NO_JUSTIFICATION, JUSTIFY_RIGHT, JUSTIFY_LEFT, or JUSTIFY_CENTER.

The field justification will be ignored if *field* is a dynamic field.

field_just returns the type of justification assigned to *field*.

RETURN VALUE

field_just returns the one of:

NO_JUSTIFICATION, JUSTIFY_RIGHT, JUSTIFY_LEFT, or JUSTIFY_CENTER.

set_field_just returns one of the following:

E_OK	– The function returned successfully.
E_SYSTEM_ERROR	– System error.
E_BAD_ARGUMENT	– An argument is incorrect.

NOTES

The header file <form.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), forms(3X).

NAME

form_field_new: new_field, dup_field, link_field, free_field, – create and destroy forms fields

SYNOPSIS

```
#include <form.h>

FIELD *new_field(int r, int c, int frow, int fcol,
      int nrow, int ncol);

FIELD *dup_field(FIELD *field, int frow, int fcol);

FIELD *link_field(FIELD *field, int frow, int fcol);

int free_field(FIELD *field);
```

DESCRIPTION

new_field creates a new field with *r* rows and *c* columns, starting at *frow*, *fcol*, in the subwindow of a form. *nrow* is the number of off-screen rows and *nbuf* is the number of additional working buffers. This routine returns a pointer to the new field.

dup_field duplicates *field* at the specified location. All field attributes are duplicated, including the current contents of the field buffers.

link_field also duplicates *field* at the specified location. However, unlike dup_field, the new field shares the field buffers with the original field. After creation, the attributes of the new field can be changed without affecting the original field.

free_field frees the storage allocated for *field*.

RETURN VALUE

Routines that return pointers return NULL on error. free_field returns one of the following:

E_OK	– The function returned successfully.
E_CONNECTED	– The field is already connected to a form.
E_SYSTEM_ERROR	– System error.
E_BAD_ARGUMENT	– An argument is incorrect.

NOTES

The header file <form.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

forms(3X).

NAME

form_field_opts: set_field_opts, field_opts_on, field_opts_off, field_opts – forms field option routines

SYNOPSIS

```
#include <form.h>

int set_field_opts(FIELD *field, OPTIONS opts);
int field_opts_on(FIELD *field, OPTIONS opts);
int field_opts_off(FIELD *field, OPTIONS opts);
OPTIONS field_opts(FIELD *field);
```

DESCRIPTION

set_field_opts turns on the named options of *field* and turns off all remaining options. Options are boolean values that can be OR-ed together.

field_opts_on turns on the named options; no other options are changed.

field_opts_off turns off the named options; no other options are changed.

field_opts returns the options set for *field*.
Field Options:

O_VISIBLE	The field is displayed.
O_ACTIVE	The field is visited during processing.
O_PUBLIC	The field contents are displayed as data is entered.
O_EDIT	The field can be edited.
O_WRAP	Words not fitting on a line are wrapped to the next line.
O_BLANK	The whole field is cleared if a character is entered in the first position.
O_AUTOSKIP	Skip to the next field when the current field becomes full.
O_NULLOK	A blank field is considered valid.
O_STATIC	The field buffers are fixed in size.
O_PASSOK	Validate field only if modified by user.

RETURN VALUE

set_field_opts, field_opts_on and field_opts_off return one of the following:

E_OK	– The function returned successfully.
E_SYSTEM_ERROR	– System error.
E_CURRENT	– The field is the current field.

NOTES

The header file <form.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), forms(3X).

NAME

form_field_userptr: set_field_userptr, field_userptr – associate application data with forms

SYNOPSIS

```
#include <form.h>

int set_field_userptr(FIELD *field, char *ptr);
char *field_userptr(FIELD *field);
```

DESCRIPTION

Every field has an associated user pointer that can be used to store pertinent data. set_field_userptr sets the user pointer of *field*. field_userptr returns the user pointer of *field*.

RETURN VALUE

field_userptr returns NULL on error. set_field_userptr returns one of the following:

E_OK	– The function returned successfully.
E_SYSTEM_ERROR	– System error.

NOTES

The header file <form.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), forms(3X).

NAME

form_field_validation: set_field_type, field_type, field_arg – forms field data type validation

SYNOPSIS

```
#include <form.h>

int set_field_type(FIELD *field, FIELDTYPE *type, ...);

FIELDTYPE *field_type(FIELD *field);

char *field_arg(FIELD *field);
```

DESCRIPTION

set_field_type associates the specified field type with *field*. Certain field types take additional arguments. TYPE_ALNUM, for instance, requires one, the minimum width specification for the field. The other predefined field types are: TYPE_ALPHA, TYPE_ENUM, TYPE_INTEGER, TYPE_NUMERIC, TYPE_REGEXP.

field_type returns a pointer to the field type of *field*. NULL is returned if no field type is assigned.

field_arg returns a pointer to the field arguments associated with the field type of *field*. NULL is returned if no field type is assigned.

RETURN VALUE

field_type and field_arg return NULL on error.

set_field_type returns one of the following:

E_OK	– The function returned successfully.
E_SYSTEM_ERROR	– System error.

NOTES

The header file <form.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), forms(3X).

NAME

form_fieldtype: new_fieldtype, free_fieldtype, set_fieldtype_arg, set_fieldtype_choice, link_fieldtype – forms fieldtype routines

SYNOPSIS

```
#include <form.h>

FIELDTYPE *new_fieldtype(int (* field_check)(FIELD *, char *),
      int (* char_check)(int, char *));

int free_fieldtype(FIELDTYPE *fieldtype);

int set_fieldtype_arg(FIELDTYPE *fieldtype,
      char *(* mak_arg)(va_list *),
      char *(* copy_arg)(char *), void (* free_arg)(char *));

int set_fieldtype_choice(FIELDTYPE *fieldtype,
      int (* next_choice)(FIELD *, char *),
      int (* prev_choice)(FIELD *, char *));

FIELDTYPE *link_fieldtype(FIELDTYPE *type1, FIELDTYPE *type2);
```

DESCRIPTION

new_fieldtype creates a new field type. The application programmer must write the function *field_check*, which validates the field value, and the function *char_check*, which validates each character. free_fieldtype frees the space allocated for the field type.

By associating function pointers with a field type, set_fieldtype_arg connects to the field type additional arguments necessary for a set_field_type call. Function *mak_arg* allocates a structure for the field specific parameters to set_field_type and returns a pointer to the saved data. Function *copy_arg* duplicates the structure created by *make_arg*. Function *free_arg* frees any storage allocated by *make_arg* or *copy_arg*.

The form_driver requests REQ_NEXT_CHOICE and REQ_PREV_CHOICE let the user request the next or previous value of a field type comprising an ordered set of values. set_fieldtype_choice allows the application programmer to implement these requests for the given field type. It associates with the given field type those application-defined functions that return pointers to the next or previous choice for the field.

link_fieldtype returns a pointer to the field type built from the two given types. The constituent types may be any application-defined or pre-defined types.

RETURN VALUE

Routines that return pointers always return NULL on error. Routines that return an integer return one of the following:

E_OK	– The function returned successfully.
E_SYSTEM_ERROR	– System error.
E_BAD_ARGUMENT	–. An argument is incorrect.
E_CONNECTED	– Type is connected to one or more fields.

NOTES

The header file <form.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), forms(3X).

NAME

form_hook: set_form_init, form_init, set_form_term, form_term, set_field_init, field_init, set_field_term, field_term – assign application-specific routines for invocation by forms

SYNOPSIS

```
#include <form.h>

int set_form_init(FORM *form, void (*func)(FORM *));
void (*)(FORM *) form_init(FORM *form);

int set_form_term(FORM *form, void (*func)(FORM *));
void (*)(FORM *) form_term(FORM *form);

int set_field_init(FORM *form, void (*func)(FORM *));
void (*)(FORM *) field_init(FORM *form);

int set_field_term(FORM *form, void (*func)(FORM *));
void (*)(FORM *) field_term(FORM *form);
```

DESCRIPTION

These routines allow the programer to assign application specific routines to be executed automatically at initialization and termination points in the forms application. The user need not specify any application-defined initialization or termination routines at all, but they may be helpful for displaying messages or page numbers and other chores.

set_form_init assigns an application-defined initialization function to be called when the *form* is posted and just after a page change. form_init returns a pointer to the initialization function, if any.

set_form_term assigns an application-defined function to be called when the *form* is unposted and just before a page change. form_term returns a pointer to the function, if any.

set_field_init assigns an application-defined function to be called when the *form* is posted and just after the current field changes. field_init returns a pointer to the function, if any.

set_field_term assigns an application-defined function to be called when the *form* is unposted and just before the current field changes. field_term returns a pointer to the function, if any.

RETURN VALUE

Routines that return pointers always return NULL on error. Routines that return an integer return one of the following:

E_OK	– The function returned successfully.
E_SYSTEM_ERROR	– System error.

NOTES

The header file <form.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), forms(3X).

NAME

form_new: new_form, free_form – create and destroy forms

SYNOPSIS

```
#include <form.h>

FORM *new_form(FIELD **fields);

int free_form(FORM *form);
```

DESCRIPTION

new_form creates a new form connected to the designated fields and returns a pointer to the form.

free_form disconnects the *form* from its associated field pointer array and deallocates the space for the form.

RETURN VALUE

new_form always returns NULL on error. free_form returns one of the following:

E_OK	– The function returned successfully.
E_BAD_ARGUMENT	– An argument is incorrect.
E_POSTED	– The form is posted.

NOTES

The header file <form.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), forms(3X).

NAME

form_new_page: set_new_page, new_page – forms pagination

SYNOPSIS

```
#include <form.h>

int set_new_page(FIELD *field, int bool);

int new_page(FIELD *field);
```

DESCRIPTION

set_new_page marks *field* as the beginning of a new page on the form.

new_page returns a boolean value indicating whether or not *field* begins a new page of the form.

RETURN VALUE

new_page returns TRUE or FALSE.

set_new_page returns one of the following:

E_OK	– The function returned successfully.
E_CONNECTED	– The field is already connected to a form.
E_SYSTEM_ERROR	– System error.

NOTES

The header file <form.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), forms(3X).

NAME

form_opts: set_form_opts, form_opts_on, form_opts_off, form_opts – forms option routines

SYNOPSIS

```
#include <form.h>

int set_form_opts(FORM *form, OPTIONS opts);
int form_opts_on(FORM *form, OPTIONS opts);
int form_opts_off(FORM *form, OPTIONS opts);
OPTIONS form_opts(FORM *form);
```

DESCRIPTION

set_form_opts turns on the named options for *form* and turns off all remaining options. Options are boolean values which can be OR-ed together.

form_opts_on turns on the named options; no other options are changed.

form_opts_off turns off the named options; no other options are changed.

form_opts returns the options set for *form*.
Form Options:

O_NL_OVERLOAD	Overload the REQ_NEW_LINE form driver request.
O_BS_OVERLOAD	Overload the REQ_DEL_PREV form driver request.

RETURN VALUE

set_form_opts, form_opts_on and form_opts_off return one of the following:

E_OK	– The function returned successfully.
E_SYSTEM_ERROR	– System error.

NOTES

The header file <form.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), forms(3X).

NAME

form_page: set_form_page, form_page, set_current_field, current_field, field_index – set forms current page and field

SYNOPSIS

```
#include <form.h>

int set_form_page(FORM *form, int page);
int form_page(FORM *form);

int set_current_field(FORM *form, FIELD *field);
FIELD *current_field(FORM *form);

int field_index(FIELD *field);
```

DESCRIPTION

set_form_page sets the page number of *form* to *page*. form_page returns the current page number of *form*.

set_current_field sets the current field of *form* to *field*. current_field returns a pointer to the current field of *form*.

field_index returns the index in the field pointer array of *field*.

RETURN VALUE

form_page returns -1 on error.

current_field returns NULL on error.

field_index returns -1 on error.

set_form_page and set_current_field return one of the following:

E_OK	– The function returned successfully.
E_SYSTEM_ERROR	– System error.
E_BAD_ARGUMENT	– An argument is incorrect.
E_BAD_STATE	– The routine was called from an initialization or termination function.
E_INVALID_FIELD	– The field contents are invalid.
E_REQUEST_DENIED	– The form driver request failed.

NOTES

The header file <form.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), forms(3X).

NAME

form_post: post_form, unpost_form – write or erase forms from associated subwindows

SYNOPSIS

```
#include <form.h>

int post_form(FORM *form);

int unpost_form(FORM *form);
```

DESCRIPTION

post_form writes *form* into its associated subwindow. The application programmer must use curses library routines to display the form on the physical screen or call update_panels if the panels library is being used.

unpost_form erases *form* from its associated subwindow.

RETURN VALUE

These routines return one of the following:

E_OK	– The function returned successfully.
E_SYSTEM_ERROR	– System error.
E_BAD_ARGUMENT	– An argument is incorrect.
E_POSTED	– The form is posted.
E_NOT_POSTED	– The form is not posted.
E_NO_ROOM	– The form does not fit in the subwindow.
E_BAD_STATE	– The routine was called from an initialization or termination function.
E_NOT_CONNECTED	– The field is not connected to a form.

NOTES

The header file <form.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), forms(3X), panels(3X), panel_update(3X).

NAME

form_userptr: set_form_userptr, form_userptr – associate application data with forms

SYNOPSIS

```
#include <form.h>

int set_form_userptr(FORM *form, char *ptr);
char *form_userptr(FORM *form);
```

DESCRIPTION

Every form has an associated user pointer that can be used to store pertinent data. set_form_userptr sets the user pointer of *form*. form_userptr returns the user pointer of *form*.

RETURN VALUE

form_userptr returns NULL on error. set_form_userptr returns one of the following:

E_OK	– The function returned successfully.
E_SYSTEM_ERROR	– System error.

NOTES

The header file <form.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), forms(3X).

NAME

form_win: set_form_win, form_win, set_form_sub, form_sub, scale_form – forms window and subwindow association routines

SYNOPSIS

```
#include <form.h>

int set_form_win(FORM *form, WINDOW *win);
WINDOW *form_win(FORM *form);

int set_form_sub(FORM *form, WINDOW *sub);
WINDOW *form_sub(FORM *form);

int scale_form(FORM *form, int *rows, int *cols);
```

DESCRIPTION

set_form_win sets the window of *form* to *win*. form_win returns a pointer to the window associated with *form*.

set_form_sub sets the subwindow of *form* to *sub*. form_sub returns a pointer to the subwindow associated with *form*.

scale_form returns the smallest window size necessary for the subwindow of *form*. *rows* and *cols* are pointers to the locations used to return the number of rows and columns for the form.

RETURN VALUE

Routines that return pointers always return NULL on error. Routines that return an integer return one of the following:

```
E_OK             – The function returned successfully.
E_SYSTEM_ERROR   – System error.
E_BAD_ARGUMENT   – An argument is incorrect.
E_NOT_CONNECTED  – The field is not connected to a form.
E_POSTED         – The form is posted.
```

NOTES

The header file <form.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), forms(3X).

NAME

forms – character based forms package

SYNOPSIS

#include <form.h>

DESCRIPTION

The form library is built using the curses library, and any program using forms routines must call one of the curses initialization routines such as initscr. A program using these routines must be compiled with -lform and -lcurses on the cc command line.

The forms package gives the applications programmer a terminal-independent method of creating and customizing forms for user-interaction. The forms package includes: field routines, which are used to create and customize fields, link fields and assign field types; fieldtype routines, which are used to create new field types for validating fields; and form routines, which are used to create and customize forms, assign pre/post processing functions, and display and interact with forms.

Current Default Values for Field Attributes

The forms package establishes initial current default values for field attributes. During field initialization, each field attribute is assigned the current default value for that attribute. An application can change or retrieve a current default attribute value by calling the appropriate set or retrieve routine with a NULL field pointer. If an application changes a current default field attribute value, subsequent fields created using new_field will have the new default attribute value. (The attributes of previously created fields are not changed if a current default attribute value is changed.)

Routine Name Index

The following table lists each forms routine and the name of the manual page on which it is described.

forms Routine Name	Manual Page Name
current_field	form_page(3X)
data_ahead	form_data(3X)
data_behind	form_data(3X)
dup_field	form_field_new(3X)
dynamic_field_info	form_field_info(3X)
field_arg	form_field_validation(3X)
field_back	form_field_attributes(3X)
field_buffer	form_field_buffer(3X)
field_count	form_field(3X)
field_fore	form_field_attributes(3X)
field_index	form_page(3X)
field_info	form_field_info(3X)
field_init	form_hook(3X)
field_just	form_field_just(3X)
field_opts	form_field_opts(3X)
field_opts_off	form_field_opts(3X)
field_opts_on	form_field_opts(3X)
field_pad	form_field_attributes(3X)

field_status	form_field_buffer(3X)
field_term	form_hook(3X)
field_type	form_field_validation(3X)
field_userptr	form_field_userptr(3X)
form_driver	form_driver(3X)
form_fields	form_field(3X)
form_init	form_hook(3X)
form_opts	form_opts(3X)
form_opts_off	form_opts(3X)
form_opts_on	form_opts(3X)
form_page	form_page(3X)
form_sub	form_win(3X)
form_term	form_hook(3X)
form_userptr	form_userptr(3X)
form_win	form_win(3X)
free_field	form_field_new(3X)
free_fieldtype	form_fieldtype(3X)
free_form	form_new(3X)
link_field	form_field_new(3X)
link_fieldtype	form_fieldtype(3X)
move_field	form_field(3X)
new_field	form_field_new(3X)
new_fieldtype	form_fieldtype(3X)
new_form	form_new(3X)
new_page	form_new_page(3X)
pos_form_cursor	form_cursor(3X)
post_form	form_post(3X)
scale_form	form_win(3X)
set_current_field	form_page(3X)
set_field_back	form_field_attributes(3X)
set_field_buffer	form_field_buffer(3X)
set_field_fore	form_field_attributes(3X)
set_field_init	form_hook(3X)
set_field_just	form_field_just(3X)
set_field_opts	form_field_opts(3X)
set_field_pad	form_field_attributes(3X)
set_field_status	form_field_buffer(3X)
set_field_term	form_hook(3X)
set_field_type	form_field_validation(3X)
set_field_userptr	form_field_userptr(3X)
set_fieldtype_arg	form_fieldtype(3X)
set_fieldtype_choice	form_fieldtype(3X)
set_form_fields	form_field(3X)
set_form_init	form_hook(3X)
set_form_opts	form_opts(3X)
set_form_page	form_page(3X)
set_form_sub	form_win(3X)
set_form_term	form_hook(3X)

`set_form_userptr`	form_userptr(3X)
`set_form_win`	form_win(3X)
`set_max_field`	form_field_buffer(3X)
`set_new_page`	form_new_page(3X)
`unpost_form`	form_post(3X)

RETURN VALUE

Routines that return a pointer always return `NULL` on error. Routines that return an integer return one of the following:

`E_OK`	–	The function returned successfully.
`E_CONNECTED`	–	The field is already connected to a form.
`E_SYSTEM_ERROR`	–	System error.
`E_BAD_ARGUMENT`	–	An argument is incorrect.
`E_CURRENT`	–	The field is the current field.
`E_POSTED`	–	The form is posted.
`E_NOT_POSTED`	–	The form is not posted.
`E_INVALID_FIELD`	–	The field contents are invalid.
`E_NOT_CONNECTED`	–	The field is not connected to a form.
`E_NO_ROOM`	–	The form does not fit in the subwindow.
`E_BAD_STATE`	–	The routine was called from an initialization or termination function.
`E_REQUEST_DENIED`	–	The form driver request failed.
`E_UNKNOWN_COMMAND`	–	An unknown request was passed to the the form driver.

NOTES

The header file `<form.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

`curses`(3X), and 3X pages whose names begin "form_" for detailed routine descriptions.

NAME

fpgetround, fpsetround, fpgetmask, fpsetmask, fpgetsticky, fpsetsticky – IEEE floating-point environment control

SYNOPSIS

```
#include <ieeefp.h>

fp_rnd fpgetround ();

fp_rnd fpsetround (rnd_dir)
fp_rnd rnd_dir;

fp_except fpgetmask ();

fp_except fpsetmask (mask)
fp_except mask;

fp_except fpgetsticky ();

fp_except fpsetsticky (sticky)
fp_except sticky;
```

DESCRIPTION

There are five floating-point exceptions: divide-by-zero, overflow, underflow, imprecise (inexact) result, and invalid operation. When a floating-point exception is detected, a trap (SIGFPE) occurs only when the corresponding mask bit is enabled. Otherwise, the corresponding sticky bit is set and the standard IEEE-specified fixup is performed on the result. These routines let the user control the behavior on occurrence of any of these exceptions, as well as the rounding mode for floating-point operations.

fpgetround returns the current rounding mode. fpsetround sets the rounding mode and returns the previous rounding mode. The enumeration type fp_rnd (defined in <ieeefp.h>) comprises the following rounding modes:

FP_RN	/* round to nearest */
FP_RP	/* round to plus */
FP_RM	/* round to minus */
FP_RZ	/* round to zero (truncate) */

fpgetmask returns the current exception masks.

fpsetmask sets the exception masks and returns the previous setting.

fpgetsticky returns the current exception sticky flags.

fpsetsticky sets (clears) the exception sticky flags and returns the previous setting.

The type fp_except is defined in <ieeefp.h>, along with the following exception masks:

FP_X_INV	/* invalid operation exception */
FP_X_OFL	/* overflow exception */
FP_X_UFL	/* underflow exception */
FP_X_DZ	/* divide-by-zero exception */
FP_X_IMP	/* imprecise (loss of precision) */

The following defaults are in effect *unless* your program includes the file <ieeefp.h>:

Rounding mode set to nearest (FP_RN).
Divide-by-zero,
floating-point overflow, and
invalid operation traps enabled.

If your program includes the file <ieeefp.h>, all traps are disabled.

SEE ALSO

isnan(3C).

CAUTIONS

fpsetsticky modifies all sticky flags. fpsetmask changes all mask bits.

Both C and F77 require truncation (round to zero) for floating-point to integral conversions. The rounding mode has no effect on these conversions.

The sticky bit is never set when the trap for the exception is enabled. As a result, it is currently impossible to determine what IEEE floating-point exception occurred from a C-coded signal handler.

NAME

fread, fwrite – binary input/output

SYNOPSIS

```
#include <stdio.h>

size_t fread (void *ptr, size_t size, size_t nitems, FILE *stream);

size_t fwrite (const void *ptr, size_t size, size_t nitems, FILE
    *stream);
```

DESCRIPTION

fread reads into an array pointed to by *ptr* up to *nitems* items of data from *stream*, where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. fread stops reading bytes if an end-of-file or error condition is encountered while reading *stream*, or if *nitems* items have been read. fread increments the data pointer in *stream* to point to the byte following the last byte read if there is one. fread does not change the contents of *stream*. fread returns the number of items read.

fwrite writes to the named output *stream* at most *nitems* items of data from the array pointed to by *ptr*, where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. 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 array pointed to by *ptr*. fwrite increments the data-pointer in *stream* by the number of bytes written. fwrite returns the number of items written.

If *size* or *nitems* is zero, then fread and fwrite return a value of 0 and do not effect the state of *stream*.

The ferror or feof routines must be used to distinguish between an error condition and end-of-file condition.

SEE ALSO

exit(2), lseek(2), read(2), write(2), abort(3C), fclose(3S), fopen(3S), getc(3S), gets(3S), printf(3S), putc(3S), puts(3S), scanf(3S), stdio(3S).

DIAGNOSTICS

If an error occurs, the error indicator for *stream* is set. If fread (fwrite) was called with a *stream* that was not opened for reading (writing) then errno will be set to EBADF.

NAME

frexp, ldexp, logb, modf, modff, nextafter, scalb – manipulate parts of floating-point numbers

SYNOPSIS

```
#include <math.h>

double frexp (double value, int *eptr);

double ldexp (double value, int exp);

double logb (double value);

double nextafter (double value1, double value2);

double scalb (double value, double exp);

double modf (double value, double *iptr);

float modff (float value, float *iptr);
```

DESCRIPTION

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

ldexp and scalb return the quantity $value * 2^{exp}$. The only difference between the two is that scalb of a signaling NaN will result in the invalid operation exception being raised.

logb returns the unbiased exponent of its floating-point argument as a double-precision floating-point value.

modf and modff (single-precision version) return the signed fractional part of *value* and store the integral part indirectly in the location pointed to by *iptr*.

nextafter returns the next representable double-precision floating-point value following *value1* in the direction of *value2*. Thus, if *value2* is less than *value1*, nextafter returns the largest representable floating-point number less than *value1*.

DIAGNOSTICS

If ldexp would cause overflow, ±HUGE (defined in math.h) is returned (according to the sign of *value*), and errno is set to ERANGEETI . If ldexp would cause underflow, zero is returned and errno is set to ERANGEETI . If the input *value* to ldexp is NaN or infinity, that input is returned and errno is set to EDOMETI . The same error conditions apply to scalb except that a signaling NaN as input will result in the raising of the invalid operation exception.

logb of NaN returns that NaN, logb of infinity returns positive infinity, and logb of zero returns negative infinity and results in the raising of the divide by zero exception. In each of these conditions errno is set to EDOMETI .

If input *value1* to nextafter is positive or negative infinity, that input is returned and errno is set to EDOMETI . The overflow and inexact exceptions are signalled when input *value1* is finite, but nextafter(*value1*, *value2*) is not. The underflow and inexact exceptions are signalled when nextafter(*value1*, *value2*) lies strictly between $\pm 2^{-1022}$. In both cases errno is set to ERANGEETI .

When the program is compiled with the cc options -Xc or -Xa, HUGE_VAL is returned instead of HUGE.

SEE ALSO

cc(1), intro(3M).

NAME

fseek, rewind, ftell – reposition a file pointer in a stream

SYNOPSIS

```
#include <stdio.h>

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

void rewind (FILE *stream);

long ftell (FILE *stream);
```

DESCRIPTION

fseek sets the position of the next input or output operation on the *stream* [see intro(3)]. The new position is at the signed distance *offset* bytes from the beginning, from the current position, or from the end of the file, according to a *ptrname* value of SEEK_SET, SEEK_CUR, or SEEK_END (defined in stdio.h) as follows:

SEEK_SET set position equal to *offset* bytes.

SEEK_CUR set position to current location plus *offset*.

SEEK_END set position to EOF plus *offset*.

fseek allows the file position indicator to be set beyond the end of the existing data in the file. If data is later written at this point, subsequent reads of data in the gap will return zero until data is actually written into the gap. fseek, by itself, does not extend the size of the file.

rewind (stream) is equivalent to:

```
(void) fseek (stream, 0L, SEEK_SET);
```

except that rewind also clears the error indicator on *stream*.

fseek and rewind clear the EOF indicator and undo any effects of ungetc on *stream*. After fseek or rewind, the next operation on a file opened for update may be either input or output.

If *stream* is writable and buffered data has not been written to the underlying file, fseek and rewind cause the unwritten data to be written to the file.

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

SEE ALSO

lseek(2), write(2), fopen(3S), popen(3S), stdio(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 that has not been opened via fopen; in particular, fseek may not be used on a terminal or on a file opened via popen. If the file has not been opened, errno is set to EBADF along with a non-zero return value. After a stream is closed, no further operations are defined on that stream.

NOTES

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 non-UNIX systems requires that an offset be used by fseek directly. Arithmetic may not meaningfully be performed on such an offset, which is not necessarily measured in bytes.

NAME

ftime – get date and time

SYNOPSIS

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

int ftime (struct timeb *tp)
```

DESCRIPTION

This interface is obsoleted by gettimeofday(2).

ftime fills in a structure pointed to by its argument, as defined by <sys/timeb.h>:

```
/*
 *  Structure returned by ftime system call
 */
struct timeb
{
        time_t time;
        unsigned short millitm;
        short timezone;
        short dstflag;
};
```

The structure contains the time since 00:00:00 GMT, January 1, 1970 (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.

SEE ALSO

date(1), gettimeofday(2), settimeofday(2), time(2), ctime(3C).

NAME

ftw, nftw – walk a file tree

SYNOPSIS

```
#include <ftw.h>

int ftw (const char *path, int (*fn) (const char *, const struct
    stat *, int), int depth);

int nftw (const char *path, int (*fn) (const char *, const struct
    stat *, int, struct FTW*), int depth, int flags);
```

DESCRIPTION

ftw recursively descends the directory hierarchy rooted in *path*. For each object in the hierarchy, ftw calls the user-defined function *fn*, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a stat structure (see stat(2)) containing information about the object, and an integer. Possible values of the integer, defined in the ftw.h header file, are:

FTW_F The object is a file.

FTW_D The object is a directory.

FTW_DNR The object is a directory that cannot be read. Descendants of the directory will not be processed.

FTW_NS stat failed on the object because of lack of appropriate permission or the object is a symbolic link that points to a non-existent file. The stat buffer passed to *fn* is undefined.

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 other than EACCES, it returns −1, and sets the error type in errno.

The function nftw is similar to ftw except that it takes an additional argument, *flags*. The *flags* field is used to specify:

FTW_PHYS Physical walk, does not follow symbolic links. Otherwise, nftw will follow links but will not walk down any path that crosses itself.

FTW_MOUNT The walk will not cross a mount point.

FTW_DEPTH All subdirectories will be visited before the directory itself.

FTW_CHDIR The walk will change to each directory before reading it.

The function nftw calls *fn* with four arguments at each file and directory. The first argument is the pathname of the object, the second is a pointer to the stat buffer, the third is an integer giving additional information, and the fourth is a struct FTW that contains the following members:

```
int base;
int level;
```

base is the offset into the pathname of the base name of the object. level indicates the depth relative to the rest of the walk, where the root level is zero.

The values of the third argument are as follows:

FTW_F The object is a file.

FTW_D The object is a directory.

FTW_DP The object is a directory and subdirectories have been visited.

FTW_SLN The object is a symbolic link that points to a non-existent file.

FTW_DNR The object is a directory that cannot be read. *fn* will not be called for any of its descendants.

FTW_NS stat failed on the object because of lack of appropriate permission. The stat buffer passed to *fn* is undefined. stat failure other than lack of appropriate permission (EACCES) is considered an error and nftw will return −1.

Both ftw and nftw use 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 faster if *depth* is at least as large as the number of levels in the tree. When ftw and nftw return, they close any file descriptors they have opened; they do not close any file descriptors that may have been opened by *fn*.

SEE ALSO

stat(2), malloc(3C).

NOTES

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

ftw uses malloc(3C) to allocate dynamic storage during its operation. If ftw is forcibly terminated, such as by longjmp being executed by *fn* or an interrupt routine, ftw 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

gamma, lgamma – log gamma function

SYNOPSIS

cc [*flag* ...] *file* ... -lm [*library* ...]

```
#include <math.h>

double gamma (double x);

double lgamma (double x);

extern int signgam;
```

DESCRIPTION

gamma and lgamma return

$$\ln(|\Gamma(x)|)$$

where $\Gamma(x)$ is defined as

$$\int_0^\infty e^{-t}t^{x-1}dt$$

The sign of $\Gamma(x)$ is returned in the external integer signgam. The argument x may not be a non-positive integer.

The following C program fragment might be used to calculate Γ:

```
if ((y = gamma(x)) > LN_MAXDOUBLE)
      error( );
y = signgam * exp(y);
```

where LN_MAXDOUBLE is the least value that causes exp to return a range error, and is defined in the values.h header file.

SEE ALSO

exp(3M), matherr(3M), values(5).

DIAGNOSTICS

For non-positive integer arguments HUGE is returned and errno is set to EDOM. A message indicating SING error is printed on the standard error output.

If the correct value would overflow, gamma and lgamma return HUGE and set errno to ERANGE.

Except when the -Xc compilation option is used, these error-handling procedures may be changed with the function matherr. When the -Xa or -Xc compilation options are used, HUGE_VAL is returned instead of HUGE and no error messages are printed.

NAME

getc, getchar, fgetc, getw – get character or word from a stream

SYNOPSIS

```
#include <stdio.h>

int getc (FILE *stream);

int getchar (void);

int fgetc (FILE *stream);

int getw (FILE *stream);
```

DESCRIPTION

getc returns the next character (i.e., byte) from the named input *stream* [see intro(3)] as an unsigned char converted to an int. It also moves the file pointer, if defined, ahead one character in *stream*. getchar is defined as getc(stdin). getc and getchar are macros.

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

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

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), gets(3S), putc(3S), scanf(3S), stdio(3S), ungetc(3S).

DIAGNOSTICS

These functions return the constant EOF at end-of-file or upon an error and set the EOF or error indicator of *stream*, respectively. If the stream was not open for reading, errno will be set to EBADF. Because EOF is a valid integer, ferror should be used to detect getw errors.

NOTES

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 implementation dependent.

The macro version of getc evaluates a *stream* argument more than once and may treat side effects incorrectly. In particular, getc(*f++) does not work sensibly. Use fgetc instead.

Because of possible differences in word length and byte ordering, files written using putw are implementation dependent, and may not be read using getw on a different processor.

Functions exist for all the above-defined macros. To get the function form, the macro name must be undefined (e.g., #undef getc).

NAME

getcwd – get pathname of current working directory

SYNOPSIS

```
#include <unistd.h>

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

DESCRIPTION

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

If *buf* is not NULL, the pathname will be stored in the space pointed to by *buf*.

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

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

EACCES A parent directory cannot be read to get its name.

EINVAL *size* is less than or equal to 0.

ERANGE *size* is greater than 0 and less than the length of the pathname plus 1.

EXAMPLE

Here is a program that prints the current working directory.

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

main()
{
        char *cwd;
        if ((cwd = getcwd(NULL, 64)) == NULL)
        {
                perror("pwd");
                exit(2);
        }
        (void)printf("%s\n", cwd);
        return(0);
}
```

DIAGNOSTICS

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

SEE ALSO

getwd(3C), malloc(3C).

NAME

getdate, getdate_err – convert user format date and time

SYNOPSIS

```
#include <time.h>

struct tm *getdate (const char *string);

extern int getdate_err;
```

DESCRIPTION

getdate converts user-definable date and/or time specifications pointed to by *string* into a tm structure. The structure declaration is in the time.h header file [see also ctime(3C)].

User-supplied templates are used to parse and interpret the input string. The templates are text files created by the user and identified via the environment variable DATEMSK. Each line in the template represents an acceptable date and/or time specification using some of the same field descriptors as the ones used by the date command. The first line in the template that matches the input specification is used for interpretation and conversion into the internal time format. If successful, the function getdate returns a pointer to a tm structure; otherwise, it returns NULL and sets the global variable getdate_err to indicate the error.

The following field descriptors are supported:

%%	same as %
%a	abbreviated weekday name
%A	full weekday name
%b	abbreviated month name
%B	full month name
%c	locale's appropriate date and time representation
%d	day of month (01-31; the leading 0 is optional)
%e	same as %d
%D	date as %m/%d/%y
%h	abbreviated month name
%H	hour (00-23)
%I	hour (01-12)
%m	month number (01-12)
%M	minute (00-59)
%n	same as \n
%p	locale's equivalent of either AM or PM
%r	time as %I:%M:%S %p
%R	time as %H:%M
%S	seconds (00-59)
%t	insert a tab
%T	time as %H:%M:%S
%w	weekday number (0-6; Sunday = 0)
%x	locale's appropriate date representation
%X	locale's appropriate time representation
%y	year with century (00-99)
%Y	year as ccyy (e.g., 1986)
%Z	time zone name or no characters if no time zone exists

The month and weekday names can consist of any combination of upper and lower case letters. The user can request that the input date or time specification be in a specific language by setting the categories LC_TIME and LC_CTYPE of setlocale.

The following example shows the possible contents of a template:

```
%m
%A %B %d %Y, %H:%M:%S
%A
%B
%m/%d/%y %I %p
%d,%m,%Y %H:%M
at %A the %dst of %B in %Y
run job at %I %p,%B %dnd
%A den %d. %B %Y %H.%M Uhr
```

The following are examples of valid input specifications for the above template:

```
getdate("10/1/87 4 PM")
getdate("Friday")
getdate("Friday September 19 1987, 10:30:30")
getdate("24,9,1986 10:30")
getdate("at monday the 1st of december in 1986")
getdate("run job at 3 PM, december %2nd")
```

If the LANG environment variable is set to german, the following is valid:

```
getdate("freitag den 10. oktober 1986 10.30 Uhr")
```

Local time and date specification are also supported. The following examples show how local date and time specification can be defined in the template.

Invocation	Line in Template
getdate("11/27/86")	%m/%d/%y
getdate("27.11.86")	%d.%m.%y
getdate("86-11-27")	%y-%m-%d
getdate("Friday 12:00:00")	%A %H:%M:%S

The following rules are applied for converting the input specification into the internal format:

> If only the weekday is given, today is assumed if the given day is equal to the current day and next week if it is less.
>
> If only the month is given, the current month is assumed if the given month is equal to the current month and next year if it is less and no year is given. (The first day of month is assumed if no day is given.)
>
> If no hour, minute, and second are given, the current hour, minute, and second are assumed.
>
> If no date is given, today is assumed if the given hour is greater than the current hour and tomorrow is assumed if it is less.

The following examples illustrate the above rules. Assume that the current date is Mon Sep 22 12:19:47 EDT 1986 and the LANG environment variable is not set.

Input	Line in Template	Date
`Mon`	`%a`	Mon Sep 22 12:19:48 EDT 1986
`Sun`	`%a`	Sun Sep 28 12:19:49 EDT 1986
`Fri`	`%a`	Fri Sep 26 12:19:49 EDT 1986
`September`	`%B`	Mon Sep 1 12:19:49 EDT 1986
`January`	`%B`	Thu Jan 1 12:19:49 EST 1987
`December`	`%B`	Mon Dec 1 12:19:49 EST 1986
`Sep Mon`	`%b %a`	Mon Sep 1 12:19:50 EDT 1986
`Jan Fri`	`%b %a`	Fri Jan 2 12:19:50 EST 1987
`Dec Mon`	`%b %a`	Mon Dec 1 12:19:50 EST 1986
`Jan Wed 1989`	`%b %a %Y`	Wed Jan 4 12:19:51 EST 1989
`Fri 9`	`%a %H`	Fri Sep 26 09:00:00 EDT 1986
`Feb 10:30`	`%b %H:%S`	Sun Feb 1 10:00:30 EST 1987
`10:30`	`%H:%M`	Tue Sep 23 10:30:00 EDT 1986
`13:30`	`%H:%M`	Mon Sep 22 13:30:00 EDT 1986

FILES

/usr/lib/locale/*locale*/LC_TIME	language specific printable files
/usr/lib/locale/*locale*/LC_CTYPE	code set specific printable files

DIAGNOSTICS

On failure `getdate` returns `NULL` and sets the variable `getdate_err` to indicate the error.

The following is a complete list of the `getdate_err` settings and their meanings.

1 The `DATEMSK` environment variable is null or undefined.

2 The template file cannot be opened for reading.

3 Failed to get file status information.

4 The template file is not a regular file.

5 An error is encountered while reading the template file.

6 `malloc` failed (not enough memory is available).

7 There is no line in the template that matches the input.

8 The input specification is invalid (e.g., `February 31`).

SEE ALSO

`setlocale`(3C), `ctype`(3C), `environ`(5).

NOTES

Subsequent calls to `getdate` alter the contents of `getdate_err`.

Dates before 1970 and after 2037 are illegal.

`getdate` makes explicit use of macros described in `ctype`(3C).

NAME

getenv – return value for environment name

SYNOPSIS

```
#include <stdlib.h>

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

DESCRIPTION

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

SEE ALSO

exec(2), putenv(3C), environ(5).

NAME

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

SYNOPSIS

```
#include <fstab.h>
```

struct fstab *getfsent(void)

struct fstab *getfsspec(char *spec)

struct fstab *getfsfile(char *file)

struct fstab *getfstype(char *type)

int setfsent(void)

int endfsent(void)

DESCRIPTION

These routines are included for compatibility with earlier revisions of the DG/UX System. They have been superseded by the getmntent(3C) 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 filesystem 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(4). Note that new *fs_type* definition strings have been added to these functions and to the <fstab.h> file to describe NFS remote file systems.

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

setfsent 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 filesystem file name is found, or until EOF is encountered. getfstype does likewise, matching on the filesystem type field.

FILES

/etc/fstab

RETURN VALUE

Null pointer (0) returned on EOF or error.

SEE ALSO

fstab(4).

CAVEAT

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

NAME

getgrent, getgrgid, getgrnam, setgrent, endgrent, fgetgrent – get group file entry

SYNOPSIS

```
#include <grp.h>

struct group *getgrent(void)

struct group *getgrgid(gid_t gid)

struct group *getgrnam(const char *name)

void setgrent(void)

void endgrent(void)

struct group *fgetgrent(FILE *f)
```

DESCRIPTION

getgrent, getgrgid, and getgrnam each return pointers to an object with the following structure containing the broken-out fields of a line in the group file.

```
structgroup {
        char   *gr_name;
        char   *gr_passwd;
        gid_t  gr_gid;
        char   **gr_mem;
};
```

The members of this structure are:

gr_name	The name of the group.
gr_passwd	The encrypted password of the group.
gr_gid	The numerical group-ID.
gr_mem	Null-terminated vector of pointers to the individual member names.

getgrent simply reads the next line while getgrgid and getgrnam search until a matching *gid* or *name* is found (or until EOF is encountered). Each routine picks up where the others leave off so successive calls may be used to search the entire file.

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.

FILES

/etc/group

DIAGNOSTICS

A null pointer (0) is returned on EOF or error.

SEE ALSO

bcs_cat(1M), getlogin(3C), getpwent(3C), group(5), ypserv(8).

BUGS

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

STANDARDS

When using m88kbcs as the Software Development Environment target, the functions mentioned above will be implemented on top of the `bcs_cat` command. Because of this, some performance degradation may be noticed in comparison to using these routines in `/lib/libc.a`.

NAME

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

SYNOPSIS

```
#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 describing an Internet host referenced by name or by address, respectively. This structure contains either the information obtained from the name server, named(8), YP (see *Managing NFS and Its Facilities on the DG/UX System*), or broken-out fields from a line in /etc/hosts. gethostbyname and gethostbyaddr read the /etc/svcorder file to determine which host/name address resolution method to use. If /etc/svcorder does not exist or has invalid data, the default order is YP, /etc/hosts, then the name server.

```
struct hostent {
   char   *h_name;        /* official name of host */
   char   **h_aliases;    /* alias list */
   int    h_addrtype;     /* host address type */
   int    h_length;       /* length of address */
   char   **h_addr_list;  /* list of address from the name server */
};
#define   h_addr h_addr_list[0]  /* address for backward
                                    compatibility */
```

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 zero-terminated array of network addresses for the host. Host addresses are returned in network byte order.

h_addr
: The first address in h_addr_list; this is for backward compatibility.

When using the name server, `gethostbyname` will search for the named host in the current domain and its parents unless the name ends in a dot. See `hostname(7)` for the domain search procedure and the alias file format.

`Sethostent` may be used to request the use of a connected TCP socket for queries. If the *stayopen* flag is non-zero, this sets the option to send all queries to the name server using TCP and to retain the connection after each call to `gethostbyname` or `gethostbyaddr`. Otherwise, queries are performed using UDP datagrams.

`gethostent` reads the next line of `/etc/hosts`, opening the file if necessary.

`Sethostent` is redefined to open and rewind the file. If the *stayopen* argument is non-zero, the hosts database will not be closed after each call to `gethostbyname` or `gethostbyaddr`. `Endhostent` is redefined to close the file.

If your system is using Network Information Services (NIS), you may need to see *Managing NFS and Its Facilities on the DG/UX System* for information on how to update the `/etc/hosts` file.

DIAGNOSTICS

Error return status from `gethostbyname` and `gethostbyaddr` is indicated by return of a null pointer. The external integer `h_errno` may then be checked to see whether this is a temporary failure or an invalid or unknown host. The routine `herror` can be used to print an error message describing the failure. If its argument string is non-NULL, it is printed, followed by a colon and a space. The error message is printed with a trailing newline.

`h_errno` can have the following values:

HOST_NOT_FOUND

No such host is known.

TRY_AGAIN

This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time may succeed.

NO_RECOVERY

Some unexpected server failure was encountered. This is a non-recoverable error.

NO_DATA

The requested name is valid but does not have an IP address; this is not a temporary error. This means that the name is known to the name server but there is no address associated with this name. Another type of request to the name server using this domain name will result in an answer; for example, a mail-forwarder may be registered for this domain.

Null pointer (0) returned on EOF or error.

FILES

`/etc/hosts`, `/etc/svcorder`

SEE ALSO

`hosts(4)`.

BUGS

All information is contained in a static area, so you must copy it if you want to save it. Only the Internet address format is currently understood.

NAME

getlogin – get login name

SYNOPSIS

```
#include <stdlib.h>

char *getlogin (void);
```

DESCRIPTION

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

If getlogin is called within a process that is not attached to a terminal, it returns a null pointer. This is the case for processes started at the system console. The correct procedure for determining the login name is to call cuserid, or to call getlogin and if it fails to call getpwuid.

FILES

/etc/utmp

SEE ALSO

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

DIAGNOSTICS

Returns a null pointer if the login name is not found.

NOTES

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

NAME

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

SYNOPSIS

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

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

struct mntent *getmntent(FILE *filep)

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

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

int endmntent(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/mnttab.

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(3S). 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 filesystem description file, <mntent.h>. The fields have meanings described in fstab(4).

```
struct mntent {
        char    *mnt_fsname;      /* file system name */
        char    *mnt_dir;         /* file system path prefix */
        char    *mnt_type;        /* 4.2, nfs, swap, or xx */
        char    *mnt_opts;        /* ro, quota, etc. */
        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. 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.

FILES

/etc/fstab
/etc/mnttab

DIAGNOSTICS

NULL pointer (0) returned on EOF or error.

SEE ALSO

fopen(3S), getfsent(3), fstab(4), mnttab(4).

BUGS

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

NAME

getnetconfig – get network configuration database entry

SYNOPSIS

```
#include <netconfig.h>

void *
setnetconfig()

struct netconfig *
getnetconfig(handlep)
void * handlep

int
endnetconfig(handlep)
void * handlep

struct netconfig *
getnetconfigent(netid)
        char * netid ;

int
freenetconfigent(netconfigp)
      struct netconfig * netconfigp ;
```

DESCRIPTION

The five library routines described on this page are part of the UNIX System V Network Selection component. They provide application access to the system network configuration database, /etc/netconfig. In addition to the netconfig database and the routines for accessing it, Network Selection includes the environment variable NETPATH (see environ(5)) and the NETPATH access routines described in getnetpath(3N).

A call to setnetconfig() has the effect of "binding" or "rewinding" the netconfig database. setnetconfig() must be called before the first call to getnetconfig() and may be called at any other time. setnetconfig() need *not* be called before a call to getnetconfigent(). setnetconfig() returns a unique handle to be used by getnetconfig().

When first called, getnetconfig() returns a pointer to the current entry in the netconfig database, formatted as a struct netconfig. getnetconfig() can thus be used to search the entire netconfig file. getnetconfig() returns NULL at end of file.

endnetconfig() should be called when processing is complete to release resources for reuse. Programmers should be aware, however, that the last call to endnetconfig() frees all memory allocated by getnetconfig() for the struct netconfig data structure. endnetconfig() may not be called before setnetconfig(). endnetconfig() returns 0 on success and -1 on failure (e.g., if setnetconfig() was not called previously).

getnetconfigent(*netid*) returns a pointer to the struct netconfig structure corresponding to *netid*. It returns NULL if *netid* is invalid (i.e., does not name an entry in the netconfig database). It returns NULL and sets *errno* in case of failure (e.g., if setnetconfig() was not called previously).

freenetconfigent(*netconfigp*) frees the netconfig structure pointed to by *netconfigp* (previously returned by getnetconfigent()).

SEE ALSO

netconfig(4), getnetpath(3N), environ(5)
Network Programmer's Guide
System Administrator's Guide

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 */
        unsigned 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 getnetbyname or getnetbyaddr.

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 EOF is encountered. Network numbers are supplied in host order.

FILES

/etc/networks

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

SEE ALSO

networks(5), ypserv(8).

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. Expecting network numbers to fit in no more than 32 bits is probably naive.

NAME

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

SYNOPSIS

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

setnetgrent(netgroup)
char *netgroup

endnetgrent( )

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

DESCRIPTION

inngetgr 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.

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(3C) to allocate space for the name. This space is released when a endnetgrent( ) call is made. getnetgrent( ) returns 1 if it succeeding 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.

FILES

/etc/netgroup

SEE ALSO

malloc(3C).

NAME

getnetpath – get /etc/netconfig entry corresponding to NETPATH component

SYNOPSIS

```
#include <netconfig.h>

void *
setnetpath()

struct netconfig *
getnetpath(handlep);
void * handlep;

int
endnetpath(handlep);
void * handlep;
```

DESCRIPTION

The three routines described on this page are part of the UNIX System V Network Selection component. They provide application access to the system network configuration database, /etc/netconfig, as it is "filtered" by the NETPATH environment variable (see environ(5)). Network Selection also includes routines that access the network configuration database directly (see getnetconfig(3N)).

A call to setnetpath() "binds" or "rewinds" NETPATH. setnetpath() must be called before the first call to getnetpath() and may be called at any other time. It returns a handle that is used by getnetpath. setnetpath() will fail if the netconfig database is not present. If NETPATH is unset, setnetpath() returns the number of "visible" networks in the netconfig file. The set of visible networks constitutes a default NETPATH.

When first called, getnetpath() returns a pointer to the netconfig database entry corresponding to the first valid NETPATH component. The netconfig entry is formatted as a struct netconfig. On each subsequent call, getnetpath returns a pointer to the netconfig entry that corresponds to the next valid NETPATH component. getnetpath() can thus be used to search the netconfig database for all networks included in the NETPATH variable. When NETPATH has been exhausted, getnetpath() returns NULL.

getnetpath() silently ignores invalid NETPATH components. A NETPATH component is invalid if there is no corresponding entry in the netconfig database.

If the NETPATH variable is *unset,* getnetpath() behaves *as if* NETPATH were set to the sequence of "default" or "visible" networks in the netconfig database, in the order in which they are listed.

endnetpath() may be called to "unbind" NETPATH when processing is complete, releasing resources for reuse. Programmer's should be aware, however, that endnetpath() frees all memory allocated by setnetpath(). endnetpath() returns 0 on success and -1 on failure (e.g., if setnetpath() was not called previously).

SEE ALSO

netconfig(4), getnetconfig(3N), environ(5)
Network Programmer's Guide
System Administrator's Guide

NAME

getopt – get option letter from argument vector

SYNOPSIS

```
#include <stdlib.h>

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

extern char *optarg;

extern int optind, opterr, optopt;
```

DESCRIPTION

getopt returns the next option letter in *argv* that matches a letter in *optstring*. It supports all the rules of the command syntax standard [see intro(1)].

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 may 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 (i.e., up to the first non-option argument), getopt returns EOF. The special option "--" (two hyphens) may be used to delimit the end of the options; when it is encountered, EOF is returned and "--" is skipped. This is useful in delimiting non-option arguments that begin with "–" (hyphen).

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 argument:

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

main (int argc, char **argv)
{
      int c;
      extern char *optarg;
      extern int optind;
      int aflg = 0;
      int bflg = 0;
      int errflg = 0;
      char *ofile = NULL;

      while ((c = getopt(argc, argv, "abo:")) != EOF)
            switch (c) {
            case 'a':
                  if (bflg)
                        errflg++;
                  else
                        aflg++;
                  break;
            case 'b':
                  if (aflg)
```

```
                        errflg++;
                  else
                        bflg++;
                  break;
            case 'o':
                  ofile = optarg;
                  (void)printf("ofile = %s\n", ofile);
                  break;
            case '?':
                  errflg++;
            }
      if (errflg) {
            (void)fprintf(stderr,
                  "usage: cmd [-a|-b] [-ofile] files...\n");
            exit (2);
      }
      for ( ; optind < argc; optind++)
            (void)printf("%s\n", argv[optind]);
      return 0;
}
```

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 argument after an option that expects one. This error message may be disabled by setting opterr to 0. The value of the character that caused the error is in optopt.

SEE ALSO

getsubopt(3C).
getopts(1), intro(1) in the *User's Reference Manual.*

NOTES

The library routine getopt does not fully check for mandatory arguments. That is, given an option string a:b and the input -a -b, getopt assumes that -b is the mandatory argument to the option -a and not that -a is missing a mandatory argument.

It is a violation of the command syntax standard [see intro(1)] for options with arguments to be grouped with other options, as in cmd -aboxxx file, where a and b are options, o is an option that requires an argument, and xxx is the argument to o. Although this syntax is permitted in the current implementation, it should not be used because it may not be supported in future releases. The correct syntax is cmd -ab -oxxx file.

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

```
#include <stdlib.h>

char *getpass (const char *prompt);
```

DESCRIPTION

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

FILES

/dev/tty

SEE ALSO

getpwent(3C), passwd(4).

NOTE

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 getprotobyname or getprotobynumber.

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 EOF is encountered.

FILES

/etc/protocols

SEE ALSO

protocols(5), ypserv(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. Only the Internet protocols are currently understood.

NAME

getpw – get name from UID

SYNOPSIS

```
#include <stdio.h>
int getpw (uid_t uid, char *buf);
```

DESCRIPTION

getpw goes through three steps:

1. Search the password file for a user id number that equals *uid*.
2. Copy the line of the password file in which *uid* was found into the array pointed to by *buf*.
3. Return 0.

getpw returns non-zero if *uid* cannot be found.

Do not use this routine in new programs; it is included only for compatibility with prior systems. See getpwent(3C) for routines to use instead.

FILES

/etc/passwd

DIAGNOSTICS

getpw returns non-zero on error.

SEE ALSO

getpwent(3C)
passwd(4) in the *System Manager's Reference for the DG/UX System*.

WARNING

The above routine uses <stdio.h>, which causes it to increase the size of programs that don't otherwise use standard I/O.

NAME

getpwent, getpwuid, getpwnam, setpwent, endpwent, setpwfile, fgetpwent – manipulate password file entry

SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);

struct passwd *getpwnam(const char *name);

struct passwd *getpwent(void);

setpwent(void);

endpwent(void);

setpwfile(char *name);

struct passwd *fgetpwent(FILE *f);
```

DESCRIPTION

getpwent, getpwuid and getpwnam each return a pointer to an object with the following structure containing the broken-out fields of a line in the password stream. The password stream consists of the /etc/passwd file and optionally the Network Information Services (NIS) password database.

```
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;
};

struct comment {
        char    *c_dept;
        char    *c_name;
        char    *c_acct;
        char    *c_bin;
};
```

The *pw_comment* field is not used; the others have meanings described in passwd(5).

setpwent opens the database; endpwent closes it. getpwuid and getpwnam search the database (opening it if necessary) for a matching *uid* or *name*. EOF is returned if there is no entry.

For programs wishing to read the entire database, getpwent reads the next line (opening the database if necessary). In addition to opening the database, setpwent can be used to make getpwent begin its search from the beginning of the database.

setpwfile changes the default password file to *name* thus allowing alternate password files to be used. Note that it does *not* close the previous file. If this is desired, endpwent should be called prior to it.

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

FILES

/etc/passwd

DIAGNOSTICS

The routines getpwent, getpwuid, getpwnam, and fgetpwent, return a null pointer (0) on EOF or error.

SEE ALSO

getlogin(3C), getgrent(3C), passwd(5), ypserv(8).

BUGS

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

STANDARDS

When using m88kbcs as the Software Development Environment target, the functions mentioned above will be implemented on top of the bcs_cat command. Because of this, some performance degradation may be noticed in comparison to using these routines in /lib/libc.a.

NAME

getrpcent, getrpcbyname, getrpcbynumber, setrpcent, endrpcent – 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.

```
structrpcent {
      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.

getrpcent 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
/var/yp/*domainname*/rpc.bynumber

SEE ALSO

rpc(5), rpcinfo(8), ypservices(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

`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.

SEE ALSO

`getrpcent`(3N), `rpc`(3N).

NAME

gets, fgets – get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char *gets (char *s);

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

DESCRIPTION

gets reads characters from the standard input stream [see intro(3)], stdin, into the array pointed to by *s*, until a newline character is read or an end-of-file condition is encountered. The newline character is discarded and the string is terminated with a null character.

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

When using gets, if the length of an input line exceeds the size of *s*, indeterminate behavior may result. For this reason, it is strongly recommended that gets be avoided in favor of fgets.

SEE ALSO

lseek(2), read(2), ferror(3S), fopen(3S), fread(3S), getc(3S), scanf(3S), stdio(3S), ungetc(3S).

DIAGNOSTICS

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a null pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a null pointer is returned and the error indicator for the stream is set. If end-of-file is encountered, the EOF indicator for the stream is set. 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.

```
structservent {
      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 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.

Setservent opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to getservbyname or getservbyport.

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 EOF 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

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. Expecting port numbers to fit in a 32 bit quantity is probably naive.

NAME

getspent, getspnam, setspent, endspent, fgetspent, lckpwdf, ulckpwdf – manipulate shadow password file entry

SYNOPSIS

```
#include <shadow.h>

struct spwd *getspent (void);

struct spwd *getspnam (const char *name);

int lckpwdf (void);

int ulckpwdf (void);

void setspent (void);

void endspent (void);

struct spwd *fgetspent (FILE *fp);
```

DESCRIPTION

The getspent and getspnam routines each return a pointer to an object with the following structure containing the broken-out fields of a line in the /etc/shadow file. Each line in the file contains a "shadow password" structure, declared in the shadow.h header file:

```
struct spwd{
        char  *sp_namp;
        char  *sp_pwdp;
        long  sp_lstchg;
        long  sp_min;
        long  sp_max;
        long  sp_warn;
        long  sp_inact;
        long  sp_expire;
        unsigned long     sp_flag;
};
```

The getspent routine when first called returns a pointer to the first spwd structure in the file; thereafter, it returns a pointer to the next spwd structure in the file; so successive calls can be used to search the entire file. The getspnam routine searches from the beginning of the file until a login name matching *name* is found, and returns a pointer to the particular structure in which it was found. The getspent and getspnam routines populate the sp_min, sp_max, sp_lstchg, sp_warn, sp_inact, sp_expire, or sp_flag field with −1 if the corresponding field in /etc/shadow is empty. If an end-of-file or an error is encountered on reading, or there is a format error in the file, these functions return a null pointer and set errno to EINVAL.

/etc/.pwd.lock is the lock file. It is used to coordinate modification access to the password files /etc/passwd and /etc/shadow. lckpwdf and ulckpwdf are routines that are used to gain modification access to the password files, through the lock file. A process first uses lckpwdf to lock the lock file, thereby gaining exclusive rights to modify the /etc/passwd or /etc/shadow password file. Upon completing modifications, a process should release the lock on the lock file via ulckpwdf. This mechanism prevents simultaneous modification of the password files.

lckpwdf attempts to lock the file /etc/.pwd.lock within 15 seconds. If unsuccessful, e.g., /etc/.pwd.lock is already locked, it returns −1. If successful, a return code other than −1 is returned.

ulckpwdf attempts to unlock the file /etc/.pwd.lock. If unsuccessful, e.g., /etc/.pwd.lock is already unlocked, it returns −1. If successful, it returns 0.

A call to the setspent routine has the effect of rewinding the shadow password file to allow repeated searches. The endspent routine may be called to close the shadow password file when processing is complete.

The fgetspent routine returns a pointer to the next spwd structure in the stream *fp*, which matches the format of /etc/shadow.

FILES

/etc/shadow
/etc/passwd
/etc/.pwd.lock

SEE ALSO

getpwent(3C), putpwent(3C), putspent(3C).

DIAGNOSTICS

getspent, getspnam, lckpwdf, ulckpwdf, and fgetspent return a null pointer on EOF or error.

NOTES

This routine is for internal use only; compatibility is not guaranteed.

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

NAME

getsubopt – parse suboptions from a string

SYNOPSIS

```
#include <stdlib.h>

int getsubopt (char **optionp, char * const *tokens, char **valuep);
```

DESCRIPTION

getsubopt parses suboptions in a flag argument that was initially parsed by getopt. 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. A command that uses this syntax is mount(1M), which allows the user to specify mount parameters with the -o option as follows:

```
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 null character at the end of the string; otherwise it isolates the suboption by replacing the comma separator with a null character, 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 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 to another program.

EXAMPLE

The following code fragment shows how to process options to the mount command using getsubopt.

```
#include <stdlib.h>

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(3C).

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 null character if there are no more options.

NOTES

During parsing, commas in the option input string are changed to null characters. White space in tokens or token-value pairs must be protected from the shell by quotes.

NAME

gettxt – retrieve a text string

SYNOPSIS

```
#include <nl_types.h>

char *gettxt (const char *msgid, const char *dflt_str);
```

DESCRIPTION

gettxt retrieves a text string from an AT&T-style message file. The arguments to the function are a message identification *msgid* and a default string *dflt_str* to be used if the retrieval fails.

The text strings are in files created by the mkmsgs utility [see mkmsgs(1)] and installed in directories in /usr/lib/locale/*locale*/LC_MESSAGES.

The directory *locale* can be viewed as the language in which the text strings are written. The user can request that messages be displayed in a specific language by setting the environment variable LC_MESSAGES. If LC_MESSAGES is not set, the environment variable LANG will be used. If LANG is not set, the files containing the strings are in /usr/lib/locale/C/LC_MESSAGES/*.

The user can also change the language in which the messages are displayed by invoking the setlocale function with the appropriate arguments.

If gettxt fails to retrieve a message in a specific language it will try to retrieve the same message in U.S. English. On failure, the processing depends on what the second argument *dflt_str* points to. A pointer to the second argument is returned if the second argument is not the null string. If *dflt_str* points to the null string, a pointer to the U.S. English text string "Message not found!!\n" is returned.

The following depicts the acceptable syntax of *msgid* for a call to gettxt.

msgid = msgfilename:msgnumber

The first field is used to indicate the file that contains the text strings and must be limited to 14 characters. These characters must be selected from the set of all character values excluding \0 (null) and the ASCII code for / (slash) and : (colon). The names of message files must be the same as the names of files created by mkmsgs and installed in /usr/lib/locale/*locale*/LC_MESSAGES/*. The numeric field indicates the sequence number of the string in the file. The strings are numbered from *1* to *n* where *n* is the number of strings in the file.

On failure to pass the correct msgid or a valid message number to gettxt a pointer to the text string "Message not found!!\n" is returned.

EXAMPLE

```
gettxt("UX:10", "hello world\n")
gettxt("UX:10", "")
```

UX is the name of the file that contains the messages. 10 is the message number.

FILES

/usr/lib/locale/C/LC_MESSAGES/*	contains default message files created by mkmsgs
/usr/lib/locale/*locale*/LC_MESSAGES/*	contains message files for different languages created by mkmsgs

SEE ALSO

fmtmsg(3C), setlocale(3C), environ(5).

exstr(1), mkmsgs(1), srchtxt(1) in the *User's Reference Manual*.
gencat(1), catopen(3C), catgets(3C) — X/Open-style message facilities.

NAME

getut: getutent, getutid, getutline, pututline, setutent, endutent, utmpname – access utmp file entry

SYNOPSIS

```
#include <utmp.h>

struct utmp *getutent (void);

struct utmp *getutid (const struct utmp *id);

struct utmp *getutline (const struct utmp *line);

struct utmp *pututline (const struct utmp *utmp);

void setutent (void);

void endutent (void);

int utmpname (const char *file);
```

DESCRIPTION

getutent, getutid, getutline, and pututline each return a pointer to a structure with the following members:

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

The structure exit status includes the following members:

```
short    e_termination;  /* termination status */
short    e_exit;         /* exit status */
```

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

getutid searches forward from the current point in the utmp file until it finds an entry with a *ut_type* matching id->ut_type if the type specified is RUN_LVL, BOOT_TIME, OLD_TIME, or NEW_TIME. If the type specified in id is INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS, or DEAD_PROCESS, then getutid will return a pointer to the first entry whose type is one of these four and whose ut_id field matches, character by character, id->ut_id . If the end of file is reached without a match, it fails.

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

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

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

endutent closes the currently open file.

utmpname allows the user to change the name of the file examined, from /etc/utmp to any other file. It is most often expected that this other file will be /etc/wtmp. If the file does not exist, this will not be apparent until the first attempt to reference the file is made. utmpname does not open the file. It just closes the old file if it is currently open and saves the new file name. If the file name given is longer than 79 characters, utmpname returns 0. Otherwise, it will return 1.

FILES

/etc/utmp
/etc/wtmp

SEE ALSO

ttyslot(3C), utmp(4).

DIAGNOSTICS

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

NOTES

The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made. On each call to either getutid or getutline, the routine examines the static structure before performing more I/O. If the contents of the static structure match what it is searching for, it looks no further. For this reason, to use getutline to search for multiple occurrences, it would be necessary to zero out the static area after each success, or getutline would just return the same structure over and over again. There is one exception to the rule about emptying the structure before further reads are done. The implicit read done by pututline (if it finds that it is not already at the correct place in the file) will not hurt the contents of the static structure returned by the getutent, getutid or getutline routines, if the user has just modified those contents and passed the pointer back to pututline.

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

NAME

getwc, getwchar, fgetwc – get wchar_t character from a stream

SYNOPSIS

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

int getwc(FILE *stream);

int getwchar(void);

int fgetwc(FILE *stream);
```

DESCRIPTION (International Functions)

getwc() transforms the next EUC character from the named input stream into a wchar_t character, and returns it It also increments the file pointer, if defined, by one EUC character in the stream. getwchar() is defined as getwc(stdin). getwc() and getwchar() are macros.

fgetwc() behaves like getwc(), however, it is a function.

DIAGNOSTICS

These functions return the constant EOF at the end-of-file or upon an error and set the EOF or error indicator of *stream*, respectively. If the error is an illegal sequence, EILSEQ is set to *errno*.

WARNINGS

If the value returned by getwc(), getwchar(), or fgetwc() is compared with the integer constant EOF after being stored in a wchar_t variable, the comparison may not succeed unless EOF is cast to type wchar_t.

SEE ALSO

getws(3W), putwc(3W), scanf(3W), widec(3W).
fclose(3S), ferror(3S), fopen(3S), scanf(3S), stdio(3S) in the *System V Release 4.0 Programmer's Reference Manual*.

NAME

getwd – get current working directory pathname

SYNOPSIS

```
char *getwd(pathname)
char *pathname;
```

DESCRIPTION

getwd copies the absolute pathname of the current working directory to *pathname* and returns a pointer to the result.

LIMITATIONS

The maximum pathname length is limited by the length of the array pathname.

DIAGNOSTICS

getwd returns zero and sets errno to an appropriate value if an error occurs.

SEE ALSO

getcwd(3C).

NAME

getwidth – get information of supplementary code sets

SYNOPSIS

```
#include <sys/euc.h>
#include <getwidth.h>

void getwidth(eucwidth_t *ptr);
```

DESCRIPTION

`getwidth()` reads the *character class table*, which is generated by `chrtbl` or `wchrtbl`, to get information of supplementary code sets, and sets it into the structure `eucwidth_t`.

The structure `eucwidth_t` is defined in the header file `/usr/include/euc.h` as follows:

```
        typedef struct {
                short int _eucw1,_eucw2,_eucw3;
                short int _scrw1,_scrw2,_scrw3;
                short int _pcw;
                char    _multibyte;
        } eucwidth_t;
```

Code set width values for three supplementary code sets are set in `_eucw1`, `_eucw2` and `_eucw3`, respectively. *Screen width* values for the three supplementary code sets are set in `_scrw1`, `_scrw2` and `_scrw3`, respectively. The width of EUC process code is set in `_pcw`. The maximum width in bytes of EUC is set in `_multibyte`.

If the **cswidth** parameter is not set, the system default is required. The system default is `cswidth 1:1,0:0,0:0`.

SEE ALSO

wchrtbl(1M).
chrtbl(1M) in the *Sytem V System Release 4.0 System Administration Reference Manual*.

NAME

getws, fgetws – get a wchar_t string from a stream

SYNOPSIS

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

wchar_t *getws(wchar_t *s);

wchar_t *fgetws(wchar_t *s, int n, FILE *stream);
```

DESCRIPTION (International Functions)

getws() reads EUC characters from *stdin*, converts them to wchar_t characters, and places them in the wchar_t array pointed to by *s*. getws() reads until a new-line character is read or an end-of-file condition is encountered. The new-line character is discarded and the wchar_t string is terminated with a wchar_t null character.

fgetws() reads EUC characters from the *stream*, converts them to wchar_t characters, and places them in the wchar_t array pointed to by *s*. fgetws() reads until *n-1* wchar_t characters are transferred to *s*, or a new-line character or an end-of-file condition is encountered. The wchar_t string is then terminated with a wchar_t null character.

DIAGNOSTICS

If end-of-file or a read error is encountered and no characters have been transformed, no wchar_t characters are transferred to *s* and a null pointer is returned and the error indicator for the stream is set. If the read error is an illegal byte sequence, EILSEQ is set to *errno*. If end-of-file is encountered, the EOF indicator for the stream is set. Otherwise, *s* is returned.

SEE ALSO

getwc(3W), scanf(3W), widec(3W).
ferror(3S), fopen(3S), fread(3S), scanf(3S), stdio(3S) in the *System V Release 4.0 Programmer's Reference Manual.*

NAME

gmatch – shell global pattern matching

SYNOPSIS

```
cc [flag ...] file ...  -lgen [library ...]

#include <libgen.h>

int gmatch (const char *str, const char *pattern);
```

DESCRIPTION

gmatch checks whether the null-terminated string *str* matches the null-terminated pattern string *pattern*. See the sh(1) section "File Name Generation" for a discussion of pattern matching. gmatch returns non-zero if the pattern matches the string, zero if the pattern doesn't. A backslash ('\') is used as an escape character in pattern strings.

EXAMPLE

```
        char *s;

        gmatch (s, "*[a\-]" )
```

gmatch returns non-zero (true) for all strings with 'a' or '–' as their last character.

SEE ALSO

sh(1) in the *User's Reference Manual*

NAME

`grantpt` – grant access to the slave pseudo-terminal device

SYNOPSIS

```
int grantpt(int fildes);
```

DESCRIPTION

The function `grantpt` changes the mode and ownership of the slave pseudo-terminal device associated with its master pseudo-terminal counter part. *fildes* is the file descriptor returned from a successful open of the master pseudo-terminal device. A `setuid` root program [see `setuid`(2)] is invoked to change the permissions. The user ID of the slave is set to the effective owner of the calling process and the group ID is set to a reserved group. The permission mode of the slave pseudo-terminal is set to readable, writeable by the owner and writeable by the group.

RETURN VALUE

Upon successful completion, the function `grantpt` returns `0`; otherwise it returns `-1`. Failure could occur if *fildes* is not an open file descriptor, if *fildes* is not associated with a master pseudo-terminal device, or if the corresponding slave device could not be accessed.

SEE ALSO

`open`(2), `setuid`(2).

`ptsname`(3C), `unlockpt`(3C)
in the *Programmer's Guide: STREAMS*.

NAME

hsearch, hcreate, hdestroy – manage hash search tables

SYNOPSIS

```
#include <search.h>

ENTRY *hsearch (ENTRY item, ACTION action);

int hcreate (size_t nel);

void hdestroy (void);
```

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. The comparison function used by hsearch is strcmp [see string(3C)]. *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 void should be cast to pointer-to-void.) *action* is a member of an enumeration type ACTION (defined in search.h) 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. Given a duplicate of an existing item, the new item is not entered and hsearch returns a pointer to the existing item. 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.

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>
#include <string.h>
#include <stdlib.h>

struct info {           /* this is the info stored in 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;
     /* 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 = (void *)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)
         }
     }
     return 0;
}
```

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.

SEE ALSO

bsearch(3C), lsearch(3C), malloc(3C), malloc(3X), string(3C), tsearch(3C).

NOTES

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

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

NAME

hypot – Euclidean distance function

SYNOPSIS

cc [*flag* ...] *file* ... -lm [*library* ...]

```
#include <math.h>

double hypot (double x, double y);
```

DESCRIPTION

hypot returns

```
    sqrt(x * x + y * y)
```

taking precautions against unwarranted overflows.

EXAMPLE

```
/* Program test for the hypot() function */

#include <stdio.h>
#include <math.h>

double  atof(), x, y, z, hypot();

main(argc, argv)
int     argc;
char    *argv[];
{
    x = atof(argv[1]);
    y = atpf(argv[2]);
    printf("Hypotenuse = %f.\n", z = hypot(x, y));
}
```

A call to the program test with the numbers 3.3 and 4.4

Hypotenuse = 5.500000.

SEE ALSO

matherr(3M).

DIAGNOSTICS

When the correct value would overflow, hypot returns HUGE and sets errno to ERANGE.

Except when the -Xc compilation option is used, these error-handling procedures may be changed with the function matherr. When the -Xa or -Xc compilation options are used, HUGE_VAL is returned instead of HUGE.

NAME

finite, unordered, copysign – IEEE floating-point routines

SYNOPSIS

```
#include <ieeefp.h>

int finite (x)
float x;
  or
double x;

int unordered (x, y)
float x;
  or
double x;
float y;
  or
double y;

double copysign (x, y)
double x;
double y;
```

DESCRIPTION

Copysign returns the value of *x* with the same sign as *y*.

Finite returns true (1) if its argument is finite: that is, -infinity < *x* < +infinity. Otherwise finite returns false (0).

Unordered returns true (1) if its arguments are unordered; otherwise it returns false (0). The arguments *x* and *y* are unordered if either or both are NaNs (Not-a-Number).

Both finite and unordered are implemented as macros included in <ieeefp.h>. Both accept either single- or double-precision arguments. To access finite as a function, the user may suppress the macro definition:

```
#include <ieeefp.h>
#undef finite
```

The finite function is also accessed when <ieeefp.h> is not included.

DIAGNOSTICS

None of these routines generates any exception.

SEE ALSO

fpgetround(3C), isnan(3C).

NAME

index – search for the first occurrence of a character in a string

SYNOPSIS

```
#include <string.h>
char *search, template, *index();
...
index(search, template);
```

where:

search is the character array to inspect.
template is the character you want to match.

DESCRIPTION

Use the index function to find the first occurrence of a specified character in a string. The include file string.h defines this function. The index function is the same as the strchr function.

EXAMPLE

```
/* Program test for the index() function */

#include <string.h>
#include <stdio.h>
#define MAX     80

char  c, string[MAX], *index();
int   i = 1, loc;

main(argc, argv)

int    argc;
char   *argv[];
{
    printf("Character template?\n");
    scanf("%c", &c);
    while (i < argc) {
        sprintf(string, "%s", argv[i]);
        if ((loc = index( string, c)) == 0)
            printf("Character '%c' does not occur in \n\t'%s'\n",
                   c, string);
        else
            printf("First occurrence of '%c' in\n\t'%s'\nat %o.\n",
                   c, string, loc);
        i++;
    }
}
```

If you call the program test with the strings alphabet, syllabary, and string, and then respond to the query with the character a, you generate the output

First occurrence of 'a' in
'alphabet'
at 34000023356.
First occurrence of 'a' in

```
        'syllabary'
at 34000023362.
Character 'a' does not occur in
        'string'
```

(The locations returned will vary with execution.)

RETURNS

The function returns NULL if the character does not occur in the string; otherwise it returns the pointer to the byte it found.

SEE ALSO

`memchr`(3C), `strchr`(3C), `strrchr`(3C).

NAME

inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof – Internet address manipulation routines

SYNOPSIS

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

struct in_addr inet_addr(cp)
char *cp;

unsigned long inet_network(cp)
char *cp;

char *inet_ntoa(in)
struct in_addr in;

struct in_addr inet_makeaddr(net, lna)
int net, lna;

int inet_lnaof(in)
struct in_addr in;

int inet_netof(in)
struct in_addr in;
```

DESCRIPTION

inet_addr, inet_network
: Interpret character strings representing numbers expressed in the Internet standard dot notation, returning numbers suitable for use as Internet addresses and Internet network numbers, respectively.

inet_ntoa
: Takes an Internet address and returns an ASCII string representing the address in dot notation.

inet_makeaddr
: Takes an Internet network number and a local network address, and constructs an Internet address from it.

inet_netof, inet_lnaof
: Break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet addresses are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine-format integer values.

Internet Addresses

Values specified using the dot notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the rightmost 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 rightmost three bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as net.host.

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as parts in a dot notation may be decimal, octal, or hexadecimal, as specified in the C language (i.e., 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`(4), `networks`(4).

DIAGNOSTICS

`Inet_addr` and `inet_network` return the value −1 for malformed requests.

BUGS

The problem of host byte ordering versus network byte ordering is confusing. There is no simple way to specify Class C network addresses, as there is for Classes A and B. The string returned by `inet_ntoa` resides in a static memory area.

NAME

initgroups – initialize the supplementary group access list

SYNOPSIS

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

int initgroups (const char *name, gid_t basegid)
```

DESCRIPTION

initgroups reads the group file, using getgrent, to get the group membership for the user specified by *name* and then initializes the supplementary group access list of the calling process using setgroups. The *basegid* group id is also included in the supplementary group access list. This is typically the real group id from the password file.

While scanning the group file, if the number of groups, including the *basegid* entry, exceeds {NGROUPS_MAX}, subsequent group entries are ignored.

initgroups will fail and not change the supplementary group access list if:

EPERM The effective user id is not superuser.

initgroups uses the routines based on getgrent(3C). If the invoking program uses any of these routines, the group structure will be overwritten in the call to initgroups

SEE ALSO

setgroups(2), getgrent(3C).

DIAGNOSTICS

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

NAME

insque, remque – insert/remove element from a queue

SYNOPSIS

```
include <search.h>

void insque(struct qelem *elem, struct qelem *pred);

void remque(struct qelem *elem);
```

DESCRIPTION

insque and remque manipulate queues built from doubly linked lists. Each element in the queue must be in the following form:

```
struct qelem {
      structqelem *q_forw;
      structqelem *q_back;
      char  q_data[];
};
```

insque inserts *elem* in a queue immediately after *pred*. remque removes an entry *elem* from a queue.

NAME

isalphanum – determine if a character is alphanumeric

SYNOPSIS

```
#include <ctype.h>

int c, result;
...
result = isalphanum(c);
```

DESCRIPTION

Use the isalphanum macro to determine whether a character is alphabetic or numeric. isalphanum is the same as isalnum. Alphabetics are A-Z and a-z; numerics are 0-9.

Do not try to redeclare this macro or you might get unexpected results.

Return Value

The isalphanum macro returns a nonzero value if the character is alphanumeric; otherwise, it returns 0.

EXAMPLES

```
/* Program test for the isalphanum() macro */

#include <ctype.h>
#include <stdio.h>

int  i = 1, result;

main(argc, argv)
int argc;
char *argv[];
{
    while (i < argc) {
        printf("Is character %c alphanumeric?  ", argv[i][0]);
printf("%s.\n", (result = isalphanum(argv[i][0])) == 0 ? "No" : "Yes");
        i++;
    }
    return 0;
}
```

A call to the program test with the characters &, 7, \, and g generates the output

```
Is character & alphanumeric? No.
Is character 7 alphanumeric? Yes.
Is character \ alphanumeric? No.
Is character g alphanumeric? Yes.
```

SEE ALSO

ishex(3C), isnan(3C).

NAME

`isastream` – test a file descriptor

SYNOPSIS

```
int isastream(int fildes);
```

DESCRIPTION

The function `isastream()` determines if a file descriptor represents a STREAMS file. *fildes* refers to an open file.

RETURN VALUE

If successful, `isastream()` returns 1 if *fildes* represents a STREAMS file, and 0 if not. On failure, `isastream()` returns -1 with `errno` set to indicate an error.

DIAGNOSTICS

Under the following conditions, `isastream()` fails and sets `errno` to:

`EBADF` *fildes* is not a valid open file.

SEE ALSO

`streamio`(7).
in the *Programmer's Guide: STREAMS*

NAME

isencrypt – determine whether a character buffer is encrypted

SYNOPSIS

cc [*flag* ...] *file* ... -lgen [*library* ...]

```
#include <libgen.h>

int isencrypt (const char *fbuf, size_t ninbuf);
```

DESCRIPTION

isencrypt uses heuristics to determine whether a buffer of characters is encrypted. It requires two arguments: a pointer to an array of characters and the number of characters in the buffer.

isencrypt assumes that the file is not encrypted if all the characters in the first block are ASCII characters. If there are non-ASCII characters in the first *ninbuf* characters, isencrypt assumes that the buffer is encrypted if the setlocale LC_CTYPE category is set to C or ascii.

If the LC_CTYPE category is set to a value other than C or ascii, then isencrypt uses a combination of heuristics to determine if the buffer is encrypted. If *ninbuf* has at least 64 characters, a chi-square test is used to determine if the bytes in the buffer have a uniform distribution; and isencrypt assumes the buffer is encrypted if it does. If the buffer has less than 64 characters, a check is made for null characters and a terminating new-line to determine whether the buffer is encrypted.

DIAGNOSTICS

If the buffer is encrypted, 1 is returned; otherwise zero is returned.

SEE ALSO

setlocale(3C).

NAME

ishex – determine if a character is hexadecimal

SYNOPSIS

```
#include <ctype.h>

int c, result;
...
result = ishex(c);
```

DESCRIPTION

Use the ishex macro to determine whether a character is a hexadecimal. Hexadecimals are 0 - 9 and a - f or A - F.

The ishex macro is the same as the isxdigit macro.

Do not try to redeclare this macro or you might get unexpected results.

EXAMPLE

```
/* Program testit for the ishex() macro */

#include <ctype.h>
#include <stdio.h>

int  i = 1, result;

main(argc, argv)
int     argc;
char    *argv[];
{
    while (i < argc) {
        printf("Is %c a hexadecimal digit?  ", argv[i][0]);
        printf("%s.\n",
        (result = ishex(argv[i][0])) == 0 ? "No" : "Yes");
        i++;
    }
}
```

Calling testit with A, f, g, and H generates the output

```
Is A a hexadecimal digit?  Yes.
Is f a hexadecimal digit?  Yes.
Is g a hexadecimal digit?  No.
Is H a hexadecimal digit?  No.
```

RETURNS

The ishex macro returns a nonzero value if the character is hexadecimal. Otherwise, it returns 0.

SEE ALSO

ctype(3C).

NAME

isnan, isnand, isnanf, finite, fpclass, unordered – determine type of floating-point number

SYNOPSIS

```
#include <ieeefp.h>

int isnand (double dsrc);

int isnanf (float fsrc);

int finite (double dsrc);

fpclass_t fpclass (double dsrc);

int unordered (double dsrc1, double dsrc2);

#include <math.h>

int isnan (double dsrc);
```

DESCRIPTION

isnan, isnand, and isnanf return true (1) if the argument *dsrc* or *fsrc* is a NaN; otherwise they return false (0). The functionalty of isnan is identical to that of isnand.

isnanf is implemented as a macro included in the ieeefp.h header file.

fpclass returns the class the *dsrc* belongs to. The 10 possible classes are as follows:

FP_SNAN	signaling NaN
FP_QNAN	quiet NaN
FP_NINF	negative infinity
FP_PINF	positive infinity
FP_NDENORM	negative denormalized non-zero
FP_PDENORM	positive denormalized non-zero
FP_NZERO	negative zero
FP_PZERO	positive zero
FP_NNORM	negative normalized non-zero
FP_PNORM	positive normalized non-zero

finite returns true (1) if the argument *dsrc* is neither infinity nor NaN; otherwise it returns false (0).

unordered returns true (1) if one of its two arguments is unordered with respect to the other argument. This is equivalent to reporting whether either argument is NaN. If neither of the arguments is NaN, false (0) is returned.

None of these routines generate any exception, even for signaling NaNs.

SEE ALSO

fpgetround(3C), intro(3M).

NAME

`itoa` – convert an integer to an ASCII character string

SYNOPSIS

```
int num;
char *itoa(int, char *), string;
...
itoa(num, string);
```

where:

num Value of type int
string A byte pointer to a character array

DESCRIPTION

The `itoa` function converts an integer to an ASCII string. If *num* is negative, the string will contain a minus sign, one to 10 digits, and a null.

The maximum number of characters `itoa` returns is 13.

RETURN VALUE

Address The address of the string returned

EXAMPLE

```
#include <stdio.h>
#define STRINGMIN 13

int i = 1, num;
char string[STRINGMIN], *itoa(int, char *);

main(argc, argv)
int argc;
char *argv[];
{
     while (i < argc) {
          num = atoi(argv[i]);
          num *= i;
          itoa(num, string);
          printf("%s\n", string);
          i++;
     }
     return 0;
}
```

A call to the program test with the numbers 1, 2, 3, and 4 generates the output

```
1
4
9
16
```

SEE ALSO

`atoi`(3C), `sprintf`(3C).

NAME

jobs – summary of DG/UX job control facilities

SYNOPSIS

```
#include <sys/sgtty.h>
#include <signal.h>
#include <sys/resource.h>
#include <sys/time.h>
#include <wait.h>

int fildes, signo;
int pid, pgrp;
union wait status;
int options;
struct rusage ru;

ioctl(fildes, TIOCSPGRP, &pgrp)
ioctl(fildes, TIOCGPGRP, &pgrp)

setpgrp2(pid, pgrp)
getpgrp2(pid)
killpg(pgrp, signo)

sigset(signo, action)
sighold(signo)
sigrelse(signo)
sigpause(signo)
sigsys(signo, action)

wait3(&status, options, &ru)
```

DESCRIPTION

The facilities described here support the job control implemented in csh(1) and may be used in other programs to provide similar facilities.

For descriptions of the individual routines, see SEE ALSO below. This section describes the facilities in general.

Terminal arbitration mechanisms

The job control mechanism works by associating with each process a number called a *process group*; related processes (e.g. in a pipeline) are given the same process group. The system assigns a single process group number to each terminal. Processes running on a terminal are given read access to that terminal only if they are in the same process group as that terminal.

Thus, a command interpreter may start several jobs running in different process groups and arbitrate access to the terminal by controlling which, if any, of these processes is in the same process group as the terminal. When a process outside the process group of the terminal tries to read from the terminal, all members of the process group of the process receive a SIGTTIN signal. This usually stops them until they are continued with a SIGCONT signal. (See sigsys(2) for a description of these signals; see tty(4) for a description of process groups.)

If a process is not in the process group of the terminal, and it tries to change the terminal's mode, the process group of that process is sent a SIGTTOU signal, causing the process group to stop. A similar mechanism is (optionally) available for output, causing processes to block with SIGTTOU when they try to write to the terminal while not in its process group; this is controlled by the LTOSTOP bit in the tty mode word. LTOSTOP is enabled by

stty tostop

and disabled (the default) by stty-tostop. (The LTOSTOP bit is described in tty(4)).

How the shell manipulates process groups

An interactive shell first establishes its own process group and a process group for the terminal; this keeps other processes from being stopped while the terminal is under its control. The shell then assigns a distinct process group to each job it creates. When a job is to be run in the foreground, the shell gives the terminal to the process group of the job using the TIOCSPGRP ioctl (See ioctl(2) and tty(4)). When a job stops or completes, the shell reclaims the terminal by resetting the terminal's process group to that of the shell, using TIOCSPGRP again.

Shells running shell scripts or running non-interactively do not manipulate process groups of jobs they create. Instead, they leave the process group of sub-processes and the terminal unchanged. This assures that if any sub-process they create blocks for terminal I/O, the shell and all its sub-processes will be blocked (since they are a single process group). The first interactive parent of the non-interactive shell can then be used to deal with the stoppage.

Processes whose parents have exited, and descendants of these processes, are protected by the system from stopping, since there can be no interactive parent. Rather than blocking, reads from the control terminal return end-of-file and writes to the control terminal are permitted (i.e., LTOSTOP has no effect for these processes.) Similarly processes that ignore or hold the SIGTTIN or SIGTTOU signal are not sent these signals when accessing their control terminal; if they are not in the process group of the control terminal, reads simply return end-of-file. Output and mode setting are also allowed.

Before a shell suspends itself, it places itself back in the process group in which it was created. It then sends this original group a stopping signal, stopping the shell, and any other intermediate processes, back up to an interactive parent. The shell also restores the process group of the terminal when it finishes; the process that resumes might not have control of the terminal otherwise.

Naive processes

A naive process does not alter the state of the terminal, and does no job control. It can usually invoke subprocesses safely, even if it has shell escapes or invokes other processes. If such a process issues a system(3C) call and this command is then stopped, both of the processes will stop together. Thus simple processes need not worry about job control.

Processes that modify the terminal state

When first setting the terminal into an unusual mode, the process should check, with the stopping signals held, that it is in the foreground. It should then change the state of the terminal, and set the catches for SIGTTIN, SIGTTOU and SIGTSTP. The following is a sample of the code that will be needed, assuming that unit 2 is known to be a terminal.

```
        int   tpgrp;
        ...

retry:
        sigset(SIGTSTP, SIG_HOLD);
```

```
    sigset(SIGTTIN, SIG_HOLD);
    sigset(SIGTTOU, SIG_HOLD);
    if (ioctl(2, TIOCGPGRP, &tpgrp) != 0)
          goto nottty;
    if (tpgrp != getpgrp(0)) { /* not in foreground */
          sigset(SIGTTOU, SIG_DFL);
          kill(0, SIGTTOU);
          /* job stops here waiting for SIGCONT */
          goto retry;
    }
    ...save old terminal modes and set new modes...
    sigset(SIGTTIN, onstop);
    sigset(SIGTTOU, onstop);
    sigset(SIGTSTP, onstop);
```

SIGTSTP is ignored in this code to prevent our process from being moved from the foreground to the background while checking if it is in the foreground. The process holds all the stopping signals in this critical section so that no other process in our process group can block us on one of these signals in the middle of our check. (This code assumes that the command interpreter will not move a process from foreground to background without stopping it; if it did, we could not make the check correctly.)

The signal-handling routine should clear the catch for the stop signal and `kill`(2) the processes in its process group with the same signal. The statement after this `kill` will be executed when the process is continued with SIGCONT.

Thus the code for the catch routine might look like:

```
      . . .
      sigset(SIGTSTP, onstop);
      sigset(SIGTTIN, onstop);
      sigset(SIGTTOU, onstop);
      . . .

onstop(signo)
      int signo;
{
      ... restore old terminal state ...
      sigset(signo, SIG_DFL);
      kill(0, signo);
      /* stop here until continued */
      sigset(signo, onstop);
      ... restore our special terminal state ...
}
```

This routine can also simulate a stop signal.

If a process does not need to save and restore state when it is stopped, but wishes to be notified when it is continued after a stop, it can catch the SIGCONT signal; the SIGCONT handler will be run when the process is continued.

Processes that lock data bases (such as the password file) should ignore SIGTTIN, SIGTTOU, and SIGTSTP signals while the data bases are being manipulated. While a process is ignoring SIGTTIN signals, reads that would normally have hung will return end-of-file; writes that would normally have caused SIGTTOU signals are instead permitted while SIGTTOU is ignored.

Interrupt-level process handling

sigset(3C) lets you handle process state changes as they occur. It provides an interrupt-handling routine for the SIGCHLD signal,. a signal that occurs whenever the status of a child process changes. You establish a signal handler as follows:

```
      sigset(SIGCHLD, onchild);
```

The shell or other process then waits for a change in child status with code like this:

```
recheck:
      sighold(SIGCHLD);        /* start critical section */
      if (no children to process) {
            sigpause(SIGCHLD);/* release SIGCHLD and pause */
            goto recheck;
      }
      sigrelse(SIGCHLD);       /* end critical region */
      /* now have a child to process */
```

Here, sighold temporarily blocks the SIGCHLD signal while the data structures are checked for a child to process. If we didn't block the signal, we would have a race condition; the signal might corrupt our decision by arriving shortly after we had finished checking the condition but before we paused.

If we need to wait, we call sigpause, which automically releases the hold on the SIGCHLD signal and waits for a signal to occur by starting a pause(2). Otherwise, we simply release the SIGCHLD signal and process the child.

Important: a long-standing bug in the signal mechanism has been fixed. The bug lost a SIGCHLD signal if it occurred while the signal was blocked. This is because sighold uses the SIG_HOLD signal set of sigsys(2) to prevent the signal action from being taken without losing the signal. Similarly, a signal action set with sigset has the signal held while the action routine is running, much as the interrupt priority of a processor is raised when a device interrupt is taken.

In this interrupt-driven style of termination processing, wait calls must not block when they retrieve status in the SIGCHLD signal handler. This is because a single invocation of the SIGCHLD handler may indicate an arbitrary number of process status changes: signals are not queued. This is similar to the case in a disk driver where several drives on a single controller may report status at once, while only one interrupt is taken.

It is even possible that no children will be ready to report status when the SIGCHLD handler is invoked, if the signal was posted while the SIGCHLD handler was active, and the child was noticed due to a SIGCHLD initially sent for another process. This causes no problem, since the handler will be called whenever there is work to do; the handler just has to collect all information by calling wait3 until no more information is available. Further status changes are guaranteed to be reflected in another SIGCHLD handler call.

Restarting system calls

In older versions of UNIX, slow system calls were interrupted when signals occurred, returning EINTR. The new signal mechanism sigset(3C) normally restarts such calls rather than interrupting them. To summarize: pause and wait return error EINTR (as before), ioctl and wait3 restart, and read and write restart unless some data was read or written; in that case, they return indicating how much data was read or written. In programs that use the older signal(2) mechanisms, all of these calls return EINTR if a signal occurs during the call.

SEE ALSO

csh(1), ioctl(2), killpg(2), setpgrp(2), sigsys(2), wait3(2), sigset(3C), tty(4).

NAME

l3tol, ltol3 – convert between 3-byte integers and long integers

SYNOPSIS

```
#include <stdlib.h>

void l3tol (long *lp, const char *cp, int n);

void ltol3 (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 file-system maintenance where the block numbers are three bytes long.

SEE ALSO

fs(4).

NOTES

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;
```

DESCRIPTION

If TYPE(*ldptr*) is the archive file magic number, ldahread reads the archive header of the common object file currently associated with *ldptr* into the area of memory beginning at *arhead*.

Ldahread returns SUCCESS or FAILURE. Ldahread 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), ldopen(3X), ar(4), ldfcn(4).

NAME

ldclose, ldaclose – close a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldclose (ldptr)
LDFILE *ldptr;

int ldaclose (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

Ldopen(3X) and ldclose provide uniform access to both simple common object (COFF) files and common object files that are members of archive files. Thus an archive of object files can be processed as if it were a series of simple object 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. Ldaclose is often used in conjunction with ldaopen(3X).

The program must be loaded with the object file access routine library libld.a.

SEE ALSO

fclose(3S), ldopen(3X), ldfcn(4).

NAME

ldfhread – read the file header of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldfhread (ldptr, filehead)
LDFILE *ldptr;
FILHDR *filehead;
```

DESCRIPTION

Ldfhread reads the file header of the common object file currently associated with *ldptr* into the area of memory beginning at *filehead*.

Ldfhread returns SUCCESS or FAILURE. Ldfhread will fail if it cannot read the file header.

In most cases you can avoid using ldfhread by using the macro HEADERETI (*ldptr*) defined in ldfcn.h (see ldfcn(4)). The information in any field *fieldname* of the file header may be accessed using HEADER(*ldptr*).*fieldname* or HEADER(*ldptr*).*sectionname*.*fieldname* (for a field within a section descriptor).

The program must be loaded with the object file access routine library libld.a.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldsyshread(3X), filehdr(4), ldfcn(4).

NAME

ldgetname – retrieve symbol name for object file symbol table entry

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

char *ldgetname (ldptr, symbol)
LDFILE *ldptr;
SYMENT *symbol;
```

DESCRIPTION

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 will return NULL (defined in stdio.h) if the name cannot be retrieved.

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), ldopen(3X), ldtbread(3X), ldtbseek(3X), ldfcn(4).

NAME

ldlread, ldlinit, ldlitem – manipulate line number entries of a common object file function

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <linenum.h>
#include <ldfcn.h>

int ldlread(ldptr, fcnindx, linenum, linent)
LDFILE *ldptr;
long fcnindx;
unsigned short linenum;
LINENO *linent;

int ldlinit(ldptr, fcnindx)
LDFILE *ldptr;
long fcnindx;

int ldlitem(ldptr, linenum, linent)
LDFILE *ldptr;
unsigned short linenum;
LINENO *linent;
```

DESCRIPTION

Ldlread searches the line number entries of the common object 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), ldopen(3X), ldtbindex(3X), ldfcn(4).

NAME

ldlseek, ldnlseek – seek to line number entries of a section of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldlseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnlseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

DESCRIPTION

Ldlseek seeks to the line number entries of the section specified by *sectindx* of the common object 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 1.

The program must be loaded with the object file access routine library libld.a.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

NAME

ldohseek – seek to the optional file header of an object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldohseek (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

Ldohseek positions the file at the optional file header of the object file currently associated with *ldptr*.

Ldohseek returns SUCCESS or FAILURE. Ldohseek will fail if it cannot seek to the system header.

The program must be loaded with the object file access routine library libld.a.

SEE ALSO

ldclose(3X), ldopen(3X), ldfhread(3X), ldfcn(4).

NAME

ldopen, ldaopen – open an object file for reading

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

LDFILE *ldopen (filename, ldptr)
char *filename;
LDFILE *ldptr;

LDFILE *ldaopen (filename, oldptr)
char *filename;
LDFILE *oldptr;
```

DESCRIPTION

Ldopen and ldclose(3X) provide uniform access to both simple common object (COFF) files and object files that are members of archive files. Thus an archive of object files can be processed as if it were a series of simple object 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) work in concert. Ldclose will return FAILURE only when TYPE(*ldptr*) is the archive magic number and when another file in the archive is to be processed. Only then should ldopen be called with the current value of *ldptr*. In all other cases, and particularly whenever a new *filename* is opened, ldopen should be called with a NULL *ldptr* argument.

The following is a model 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 reopen *filename* 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.

Both ldopen and ldaopen open *filename* for reading. Both functions return NULL if

a) *filename* cannot be opened

b) memory for the LDFILE structure cannot be allocated

c) *filename* is too small to be an object file or an archive of object files. The functions try to read the header of object files, so a file smaller than a header will cause the functions to return NULL.

Note that a successful open does not ensure that the given file is an object file or an archive of object files.

The program must be loaded with the object file access routine library `libld.a`.

SEE ALSO

`fopen`(3S), `ldclose`(3X), `ldfcn`(4).

NAME

ldrseek, ldnrseek – seek to relocation entries of a section of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldrseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnrseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

DESCRIPTION

Ldrseek seeks to the relocation entries of the section specified by *sectindx* of the common object 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 that the first section has an index of 1.

The program must be loaded with the object file access routine library libld.a.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

NAME

ldshread, ldnshread – read an indexed/named section header of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <scnhdr.h>
#include <ldfcn.h>

int ldshread (ldptr, sectindx, secthead)
LDFILE *ldptr;
unsigned short sectindx;
SCNHDR *secthead;

int ldnshread (ldptr, sectname, secthead)
LDFILE *ldptr;
char *sectname;
SCNHDR *secthead;
```

DESCRIPTION

Ldshread reads the section header specified by *sectindx* of the common object 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 that the first section header has an index of 1.

The program must be loaded with the object file access routine library libld.a.

SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4).

NAME

ldsseek, ldnsseek – seek to an indexed/named section of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldsseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnsseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

DESCRIPTION

Ldsseek seeks to the section specified by *sectindx* of the common object 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 that the first section has an index of 1.

The program must be loaded with the object file access routine library libld.a.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

NAME

ldtbindex – compute index of symbol table entry of an object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

long ldtbindex (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

Ldtbindex returns the (long) index of the symbol table entry at the current position of the object file associated with *ldptr*.

The index returned by ldtbindex may be used in subsequent calls to ldtbread(3X). 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, returning BADINDEX, 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 0.

The program must be loaded with the object file access routine library libld.a.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldtbseek(3X), ldfcn(4).

NAME

ldtbread – read an indexed symbol table entry of an object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldtbread (ldptr, symindex, symbol)
LDFILE *ldptr;
long symindex;
SYMENT *symbol;
```

DESCRIPTION

Ldtbread reads the symbol table entry specified by *symindex* of the object file currently associated with *ldptr* into the area of memory beginning at *symbol*.

Ldtbread returns SUCCESS or FAILURE. Ldtbread will fail if *symindex* is greater than the number of symbols in the object file or if it cannot read the specified symbol table entry.

Note that the first symbol in the symbol table has an index of 0.

The program must be loaded with the object file access routine library libld.a.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbseek(3X), ldgetname(3X), ldfcn(4).

NAME

ldtbseek – seek to the symbol table of an object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldtbseek (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

Ldtbseek seeks to the symbol table of the object file currently associated with *ldptr*.

Ldtbseek returns SUCCESS or FAILURE. Ldtbseek will fail if the symbol table has been stripped from the object file or if it cannot seek to the symbol table.

The program must be loaded with the object file access routine library libld.a.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldfcn(4).

NAME

localeconv – get numeric formatting information

SYNOPSIS

```
#include <locale.h>

struct lconv *localeconv (void);
```

DESCRIPTION

localeconv sets the components of an object with type struct lconv (defined in locale.h) with the values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale [see setlocale(3C)]. The definition of struct lconv is given below (the values for the fields in the C locale are given in comments):

```
char *decimal_point;       /* "." */
char *thousands_sep;       /* "" (zero length string) */
char *grouping;            /* "" */
char *int_curr_symbol;     /* "" */
char *currency_symbol;     /* "" */
char *mon_decimal_point;   /* "" */
char *mon_thousands_sep;   /* "" */
char *mon_grouping;        /* "" */
char *positive_sign;       /* "" */
char *negative_sign;       /* "" */
char int_frac_digits;      /* CHAR_MAX */
char frac_digits;          /* CHAR_MAX */
char p_cs_precedes;        /* CHAR_MAX */
char p_sep_by_space;       /* CHAR_MAX */
char n_cs_precedes;        /* CHAR_MAX */
char n_sep_by_space;       /* CHAR_MAX */
char p_sign_posn;          /* CHAR_MAX */
char n_sign_posn;          /* CHAR_MAX */
```

The members of the structure with type char * are strings, any of which (except decimal_point) can point to "", to indicate that the value is not available in the current locale or is of zero length. The members with type char are nonnegative numbers, any of which can be CHAR_MAX (defined in the limits.h header file) to indicate that the value is not available in the current locale. The members are the following:

char *decimal_point
: The decimal-point character used to format non-monetary quantities.

char *thousands_sep
: The character used to separate groups of digits to the left of the decimal-point character in formatted non-monetary quantities.

char *grouping
: A string in which each element is taken as an integer that indicates the number of digits that compose the current group in a formatted non-monetary quantity. The elements of grouping are interpreted according to the following:

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 compose 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.

`char *int_curr_symbol`
The international currency symbol applicable to the current locale, left-justified within a four-character space-padded field. The character sequences should match with those specified in: *ISO 4217 Codes for the Representation of Currency and Funds*.

`char *currency_symbol`
The local currency symbol applicable to the current locale.

`char *mon_decimal_point`
The decimal point used to format monetary quantities.

`char *mon_thousands_sep`
The separator for groups of digits to the left of the decimal point in formatted monetary quantities.

`char *mon_grouping`
A string in which each element is taken as an integer that indicates the number of digits that compose the current group in a formatted monetary quantity. The elements of `mon_grouping` are interpreted according to the rules described under `grouping`.

`char *positive_sign`
The string used to indicate a nonnegative-valued formatted monetary quantity.

`char *negative_sign`
The string used to indicate a negative-valued formatted monetary quantity.

`char int_frac_digits`
The number of fractional digits (those to the right of the decimal point) to be displayed in an internationally formatted monetary quantity.

`char frac_digits`
The number of fractional digits (those to the right of the decimal point) to be displayed in a formatted monetary quantity.

`char p_cs_precedes`
Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a nonnegative formatted monetary quantity.

`char p_sep_by_space`
Set to 1 or 0 if the `currency_symbol` respectively is or is not separated by a space from the value for a nonnegative formatted monetary quantity.

`char n_cs_precedes`
Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a negative formatted monetary quantity.

`char n_sep_by_space`
Set to 1 or 0 if the `currency_symbol` respectively is or is not separated by a space from the value for a negative formatted monetary quantity.

`char p_sign_posn`
Set to a value indicating the positioning of the `positive_sign` for a nonnegative formatted monetary quantity. The value of `p_sign_posn` is interpreted according to the following:

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 succeeds the currency_symbol.

char n_sign_posn

Set to a value indicating the positioning of the negative_sign for a negative formatted monetary quantity. The value of n_sign_posn is interpreted according to the rules described under p_sign_posn.

RETURNS

localeconv returns a pointer to the filled-in object. The structure pointed to by the return value may be overwritten by a subsequent call to localeconv.

EXAMPLES

The following table illustrates the rules used by four countries to format monetary quantities.

Country	Positive format	Negative format	International format
Italy	L.1.234	-L.1.234	ITL.1.234
Netherlands	F 1.234,56	F -1.234,56	NLG 1.234,56
Norway	kr1.234,56	kr1.234,56-	NOK 1.234,56
Switzerland	SFrs.1,234.56	SFrs.1,234.56C	CHF 1,234.56

For these four countries, the respective values for the monetary members of the structure returned by localeconv are as follows:

	Italy	Netherlands	Norway	Switzerland
int_curr_symbol	"ITL."	"NLG "	"NOK "	"CHF "
currency_symbol	"L."	"F"	"kr"	"SFrs."
mon_decimal_point	""	","	","	"."
mon_thousands_sep	"."	"."	"."	","
mon_grouping	"\3"	"\3"	"\3"	"\3"
positive_sign	""	""	""	""
negative_sign	"-"	"-"	"-"	"C"
int_frac_digits	0	2	2	2
frac_digits	0	2	2	2
p_cs_precedes	1	1	1	1
p_sep_by_space	0	1	0	0
n_cs_precedes	1	1	1	1
n_sep_by_space	0	1	0	0
p_sign_posn	1	1	1	1
n_sign_posn	1	4	2	2

FILES

/usr/lib/locale/*locale*/LC_MONETARY	LC_MONETARY database for *locale*
/usr/lib/locale/*locale*/LC_NUMERIC	LC_NUMERIC database for *locale*

SEE ALSO

setlocale(3C).
chrtbl(1M), montbl(1M) in the *System Administrator's Reference Manual*.

NAME

lockf – record locking on files

SYNOPSIS

```
#include <unistd.h>

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

DESCRIPTION

lockf allows sections of a file to be locked; advisory or mandatory write locks depending on the mode bits of the file [see chmod(2)]. Locking calls from other processes that attempt to lock the locked file section will either return an error value or be put to sleep until the resource becomes unlocked. All the locks for a process are removed when the process terminates. [See fcntl(2) for more information about record locking.]

fildes is an open file descriptor. The file descriptor must have O_WRONLY or O_RDWR permission in order to establish locks with this function call.

function is a control value that specifies the action to be taken. The permissible values for *function* are defined in unistd.h as follows:

```
#define   F_ULOCK    0   /* unlock previously locked section */
#define   F_LOCK     1   /* lock section for exclusive use */
#define   F_TLOCK    2   /* test & lock section for exclusive use */
#define   F_TEST     3   /* test section for other locks */
```

All other values of *function* are reserved for future extensions and will result in an error return if not implemented.

F_TEST is used to detect if a lock by another process is present on the specified section. F_LOCK and F_TLOCK both lock a section of a file if the section is available. F_ULOCK removes locks from a section of the file.

size is the number of contiguous bytes to be locked or unlocked. The resource to be locked or unlocked starts at the current offset in the file and extends forward for a positive size and backward for a negative size (the preceding bytes up to but not including the current offset). If *size* is zero, the section from the current offset through the largest file offset is locked (i.e., from the current offset through the present or any future end-of-file). An area need not be allocated to the file in order to be locked as such locks may exist past the end-of-file.

The sections locked with F_LOCK or F_TLOCK may, in whole or in part, contain or be contained by a previously locked section for the same process. Locked sections will be unlocked starting at the the point of the offset through *size* bytes or to the end of file if *size* is (off_t) 0. When this situation occurs, or if this situation occurs in adjacent sections, the sections are combined into a single section. If the request requires that a new element be added to the table of active locks and this table is already full, an error is returned, and the new section is not locked.

F_LOCK and F_TLOCK requests differ only by the action taken if the resource is not available. F_LOCK will cause the calling process to sleep until the resource is available. F_TLOCK will cause the function to return a -1 and set errno to EACCES if the section is already locked by another process.

F_ULOCK requests may, in whole or in part, release one or more locked sections controlled by the process. When sections are not fully released, the remaining sections are still locked by the process. Releasing the center section of a locked section requires an additional element in the table of active locks. If this table is full, an errno is set to ENOLK and the requested section is not released.

A potential for deadlock occurs if a process controlling a locked resource is put to sleep by requesting another process's locked resource. Thus calls to lockf or fcntl scan for a deadlock prior to sleeping on a locked resource. An error return is made if sleeping on the locked resource would cause a deadlock.

Sleeping on a resource is interrupted with any signal. The alarm system call may be used to provide a timeout facility in applications that require this facility.

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

EBADF *fildes* is not a valid open descriptor.

EAGAIN *cmd* is F_TLOCK or F_TEST and the section is already locked by another process.

EDEADLK *cmd* is F_LOCK and a deadlock would occur.

ENOLK *cmd* is F_LOCK, F_TLOCK, or F_ULOCK and the number of entries in the lock table would exceed the number allocated on the system.

ECOMM *fildes* is on a remote machine and the link to that machine is no longer active.

SEE ALSO

intro(2), alarm(2), chmod(2), close(2), creat(2), fcntl(2), open(2), read(2), write(2).

DIAGNOSTICS

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

NOTES

Unexpected results may occur in processes that do buffering in the user address space. The process may later read/write data that is/was locked. The standard I/O package is the most common source of unexpected buffering.

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

NAME

logname – return login name of user

SYNOPSIS

```
char *logname( )
```

DESCRIPTION

Logname returns a pointer to the null-terminated login name; it extracts the $LOGNAME variable from the user's environment.

This routine is kept in /lib/libPW.a.

FILES

/etc/profile

SEE ALSO

profile(4), environ(5), cuserid(3S), getlogin(3C), getpwuid(3C).
env(1), login(1) in the *User's Reference for the DG/UX System*

CAVEATS

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

This method of determining a login name is subject to forgery.

This routine will cease to exist in the future. Use cuserid, or a combination of getlogin and getpwuid instead.

NAME

lsearch, lfind – linear search and update

SYNOPSIS

```
#include <search.h>

void *lsearch (const void *key, void * base,
     size_t *nelp, size_t width,
     int (*compar) (const void *, const void *));

void *lfind (const void *key, const void *base,
     size_t *nelp, size_t width,
     int (*compar)(const void *, const void *));
```

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. *width* is the size of an element in bytes.

compar is a pointer to the comparison function that 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 may be pointers to any type.

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 value returned should be cast into type pointer-to-element.

EXAMPLE

This program will read in less than TABSIZE strings of length less than ELSIZE and store them in a table, eliminating duplicates, and then will print each entry.

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

#define TABSIZE 50
#define ELSIZE 120

main()
{
      char line[ELSIZE];/* buffer to hold input string */
      char tab[TABSIZE][ELSIZE];     /* table of strings */
      size_t nel = 0;          /* number of entries in tab */
      int i;
```

```
            while (fgets(line, ELSIZE, stdin) != NULL &&
                  nel < TABSIZE)
                  (void) lsearch(line, tab, &nel, ELSIZE, strcmp);
            for( i = 0; i < nel; i++ )
                  (void)fputs(tab[i], stdout);
            return 0;
      }
```

SEE ALSO

bsearch(3C), hsearch(3C), string(3C), tsearch(3C).

NOTES

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.

Undefined results can occur if there is not enough room in the table to add a new item.

NAME

main – enter a C main program

SYNOPSIS

```
main ([argc, argv, envp])
int argc;
char *argv[];
char *envp[];
{
.
.
.
}
```

where:

argc is optional and is the number of arguments, including the program name, with which you invoke the C program.

argv is optional and is a pointer to an array of pointers to strings; argv[0] is the name you invoked the program with; argv[1] is the first argument on the command line after the program name; argv[argc] is a null pointer.

envp is optional and is a pointer to an array of pointers to strings, each of which is a separate environment variable of the form NAME = VALUE. The getenv function (described earlier in this chapter) accesses this list of environment variables. The DG/UX system routines execle and execve can change the environment of the new process if it is a C program.

DESCRIPTION

Every C program has at least one function: the main function. This function provides a place for the program to begin execution; the main function *must* be present to initiate a C program. The runtime initializer calls main and returns to the system when main returns.

Since argv[0] is the program name, the initial value of argc is always at least 1. If you want to manipulate the argv character arrays as something other than strings, you must make explicit use of such functions as atof or atoi.

If your program does not process command line arguments, begin the function with the following:

```
main()
```

EXAMPLE

```
/* Program test for the main() function */

int  i = 1;

main(argc, argv, envp)
int     argc;
char   *argv[];
char   *envp[];
{
    printf("You called the program %s.\n", argv[0]);
    while (i < argc) {
        printf("Argument %d for this run is %s.\n",
        i, argv[i]);
```

```
            i++;
        }
    }
```

A call to the program test with arguments alpha, beta, and gamma generates the output

```
You called the program test.
Argument 1 for this run is alpha.
Argument 2 for this run is beta.
Argument 3 for this run is gamma.
```

RETURNS

The `main` function's return value is used as an argument to the `exit` function. The value 0 typically means no errors occurred, and a non-zero value indicates an error condition. The wait function can retrieve this value.

SEE ALSO

`exec`(2), `wait`(2), `wait3`(2), `exit`(3C), `getenv`(3C).

NAME

malloc, free, realloc, calloc, memalign, valloc, – memory allocator

SYNOPSIS

```
#include <stdlib.h>

void *malloc (size_t size);

void free (void *ptr);

void *realloc (void *ptr, size_t size);

void *calloc (size_t nelem, size_t elsize);

void *memalign(size_t alignment, size_t size);

void *valloc(size_t size);

#include <alloca.h>

char *alloca(int size);
```

DESCRIPTION

malloc and free provide a simple general-purpose memory allocation package. malloc returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to free is a pointer to a block previously allocated by malloc, calloc or realloc. After free is performed this space is made available for further allocation. If *ptr* is a NULL pointer, no action occurs.

Undefined results will occur if the space assigned by malloc is overrun or if some random number is handed to free.

realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If *ptr* is NULL, realloc behaves like malloc for the specified size. If *size* is zero and *ptr* is not a null pointer, the object pointed to is freed.

calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

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(sysconf(_SC_PAGESIZE),size).

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

malloc, realloc, calloc, memalign, and valloc will fail if there is not enough available memory.

alloca allocates *size* bytes of space in the stack frame of the caller. This temporary space is automatically freed on return. alloca is only supported by the GNU C language compiler.

SEE ALSO

malloc(3X), gcc(1).

DIAGNOSTICS

If there is no available memory, malloc, realloc, memalign, valloc, and calloc return a null pointer. When realloc returns NULL, the block pointed to by *ptr* is left intact. If *size*, *nelem*, or *elsize* is 0, a unique pointer to the arena is returned.

NAME

malloc, free, realloc, calloc, mallopt, mallinfo – memory allocator

SYNOPSIS

cc [*flag* ...] *file* ... -lmalloc [*library* ...]

```
#include <stdlib.h>

void *malloc (size_t size)

void free (void *ptr)

void *realloc (void *ptr, size_t size)

void *calloc (size_t nelem, size_t elsize)

#include <malloc.h>

int mallopt (int cmd, int value)

struct mallinfo mallinfo (void)

#include <alloca.h>

char *alloca(int size);
```

DESCRIPTION

malloc and free provide a simple general-purpose memory allocation package. It is found in the library "malloc", and is loaded if the option "–lmalloc" is used with cc(1) or ld(1).

malloc returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to free is a pointer to a block previously allocated by malloc; after free is performed this space is made available for further allocation, and its contents have been destroyed (but see mallopt below for a way to change this behavior). If *ptr* is a null pointer, no action occurs.

Undefined results occur if the space assigned by malloc is overrun or if some random number is handed to free.

realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents are unchanged up to the lesser of the new and old sizes. If *ptr* is a null pointer, realloc behaves like malloc for the specified size. If *size* is zero and *ptr* is not a null pointer, the object it points to is freed.

calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

mallopt provides for control over the allocation algorithm. The available values for *cmd* are:

M_MXFAST Set *maxfast* to *value*. The algorithm allocates all blocks below the size of *maxfast* in large groups and then doles them out very quickly. The default value for *maxfast* is 96.

M_NLBLKS Set *numlblks* to *value*. The above mentioned "large groups" each contain *numlblks* blocks. *numlblks* must be greater than 0. The default value for *numlblks* is 100.

M_GRAIN Set *grain* to *value*. The sizes of all blocks smaller than *maxfast* are considered to be rounded up to the nearest multiple of *grain*. *grain* must be greater than 0. The default value of *grain* is the smallest number of bytes that will allow alignment of any data type. Value will be rounded up to a multiple of the default when *grain* is set.

M_KEEP Preserve data in a freed block until the next malloc, realloc, or calloc. This option is provided only for compatibility with the old version of malloc and is not recommended.

These values are defined in the malloc.h header file.

mallopt may be called repeatedly, but may not be called after the first small block is allocated.

mallinfo provides instrumentation describing space usage. It returns the structure:

```
struct mallinfo  {
     int arena;      /* total space in arena */
     int ordblks;    /* number of ordinary blocks */
     int smblks;     /* number of small blocks */
     int hblkhd;     /* space in holding block headers */
     int hblks;      /* number of holding blocks */
     int usmblks;    /* space in small blocks in use */
     int fsmblks;    /* space in free small blocks */
     int uordblks;   /* space in ordinary blocks in use */
     int fordblks;   /* space in free ordinary blocks */
     int keepcost;   /* space penalty if keep option */
                     /* is used */
}
```

This structure is defined in the malloc.h header file.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

alloca allocates *size* bytes of space in the stack frame of the caller. This temporary space is automatically freed on return. alloca is only supported by the GNU C language compiler.

SEE ALSO

brk(2), malloc(3C), gcc(1).

DIAGNOSTICS

malloc, realloc, and calloc return a NULL pointer if there is not enough available memory. When realloc returns NULL, the block pointed to by *ptr* is left intact. If mallopt is called after any allocation or if *cmd* or *value* are invalid, nonzero is returned. Otherwise, it returns zero.

NOTES

Note that unlike malloc(3C), this package does not preserve the contents of a block when it is freed, unless the M_KEEP option of mallopt is used.

Undocumented features of malloc(3C) have not been duplicated.

Function prototypes for malloc, realloc, calloc and free are also defined in the <malloc.h> header file for compatibility with old applications. New applications should include <stdlib.h> to access the prototypes for these functions.

NAME

matherr – error-handling function

SYNOPSIS

```
cc [flag ...] file ...  -lm [library ...]

#include <math.h>

int matherr (struct exception *x);
```

DESCRIPTION

matherr is invoked by functions in the math libraries when errors are detected. Note that matherr is not invoked when the -Xc compilation option is used. Users may define their own procedures for handling errors, by including a function named matherr in their programs. matherr must be of the form described above. When an error occurs, a pointer to the exception structure *x* will be passed to the user-supplied matherr function. This structure, which is defined in the math.h header file, is as follows:

```
struct exception {
      int type;
      char *name;
      double arg1, arg2, retval;
};
```

The element type is an integer describing the type of error that has occurred, from the following list of constants (defined in the header file):

DOMAIN	argument domain error
SING	argument singularity
OVERFLOW	overflow range error
UNDERFLOW	underflow range error
TLOSS	total loss of significance
PLOSS	partial loss of significance

The element name points to a string containing the name of the function that incurred the error. The variables arg1 and arg2 are the arguments with which the function was invoked. retval is set to the default value that will be returned by the function unless the user's matherr sets it to a different value.

If the user's matherr function returns non-zero, no error message will be printed, and errno will not be set.

If matherr is not supplied by the user, the default error-handling procedures, described with the math functions involved, will be invoked upon error. These procedures are also summarized in the table below. In every case, errno is set to EDOM or ERANGE and the program continues.

Default Error Handling Procedures						
	Types of Errors					
type	DOMAIN	SING	OVERFLOW	UNDERFLOW	TLOSS	PLOSS
errno	EDOM	EDOM	ERANGE	ERANGE	ERANGE	ERANGE
BESSEL:	-	-	-	-	M, 0	-
y0, y1, yn (arg ≤ 0)	M, -H	-	-	-	-	-
EXP, EXPF:	-	-	H	0	-	-
LOG, LOG10:						
LOGF, LOG10F:						
(arg < 0)	M, -H	-	-	-	-	-
(arg = 0)	M, -H	-	-	-	-	-
POW, POWF:	-	-	±H	0	-	-
neg ** non-int	M, 0	-	-	-	-	-
0 ** non-pos	M, 0	-	-	-	-	-
SQRT, SQRTF:	M, 0	-	-	-	-	-
FMOD, FMODF:						
(arg2 = 0)	M, X	-	-	-	-	-
REMAINDER:						
(arg2 = 0)	M, N	-	-	-	-	-
GAMMA, LGAMMA:	-	M, H	H	-	-	-
HYPOT:	-	-	H	-	-	-
SINH, SINHF:	-	-	±H	-	-	-
COSH, COSHF:	-	-	H	-	-	-
ASIN, ACOS, ATAN2:						
ASINF, ACOSF, ATAN2F:	M, 0	-	-	-	-	-
ACOSH:	M, N	-	-	-	-	-
ATANH:						
(|arg| > 1)	M, N	-	-	-	-	-
(|arg| = 1)	-	M, N	-	-	-	-

Abbreviations	
M	Message is printed (not with the -Xa or -Xc options).
H	HUGE is returned (HUGE_VAL with the -Xa or -Xc options).
-H	-HUGE is returned (-HUGE_VAL with the -Xa or -Xc options).
±H	HUGE or -HUGE is returned. (HUGE_VAL or -HUGE_VAL with the -Xa or -Xc options).
0	0 is returned.
X	*arg1* is returned.
N	NaN is returned.

EXAMPLE

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

```
int
matherr(register struct exception *x);
{
      switch (x->type) {
      case DOMAIN:
            /* change sqrt to return sqrt(-arg1), not 0 */
            if (!strcmp(x->name, "sqrt")) {
                  x->retval = sqrt(-x->arg1);
                  return (0); /* print message and set errno */
            }
      case SING:
            /* all other domain or sing errors, print message */
            /* and abort */
            fprintf(stderr, "domain error in %s\n", x->name);
            abort( );
      case PLOSS:
            /* print detailed error message */
            fprintf(stderr, "loss of significance in %s(%g)=%g\n",
                  x->name, x->arg1, x->retval);
            return (1); /* take no other action */
      }
      return (0); /* all other errors, execute default procedure */
}
```

SEE ALSO

erf(3M).

NOTES

Error handling in -Xa and -Xt modes [see cc(1)] is described more completely on individual math library pages.

NAME

mbchar: mbtowc, mblen, wctomb – multibyte character handling

SYNOPSIS

```
#include <stdlib.h>

int mbtowc (wchar_t *pwc, const char *s, size_t n);

int mblen (const char *s, size_t n);

int wctomb (char *s, wchar_t wchar);
```

DESCRIPTION

Multibyte characters are used to represent characters in an extended character set. This is needed for locales where 8 bits are not enough to represent all the characters in the character set.

The multibyte character handling functions provide the means of translating multibyte characters into wide characters and back again. Wide characters have type `wchar_t` (defined in `stdlib.h`), which is an integral type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales.

A maximum of 3 extended character sets are supported for each locale. The number of bytes in an extended character set is defined by the `LC_CTYPE` category of the locale [see `setlocale`(3C)]. However, the maximum number of bytes in any multibyte character will never be greater than `MB_LEN_MAX`. which is defined in `stdlib.h`. The maximum number of bytes in a character in an extended character set in the `current` locale is given by the macro, `MB_CUR_MAX`, also defined in `stdlib.h`.

`mbtowc` determines the number of bytes that comprise the multibyte character pointed to by *s*. Also, if *pwc* is not a null pointer, `mbtowc` converts the multibyte character to a wide character and places the result in the object pointed to by *pwc*. (The value of the wide character corresponding to the null character is zero.) At most *n* characters will be examined, starting at the character pointed to by *s*.

If *s* is a null pointer, `mbtowc` simply returns 0. If *s* is not a null pointer, then, if *s* points to the null character, `mbtowc` returns 0; if the next *n* or fewer bytes form a valid multibyte character, `mbtowc` returns the number of bytes that comprise the converted multibyte character; otherwise, *s* does not point to a valid multibyte character and `mbtowc` returns −1.

`mblen` determines the number of bytes comprising the multibyte character pointed to by *s*. It is equivalent to

```
mbtowc ((wchar_t *)0, s, n);
```

`wctomb` determines the number of bytes needed to represent the multibyte character corresponding to the code whose value is *wchar*, and, if *s* is not a null pointer, stores the multibyte character representation in the array pointed to by *s*. At most `MB_CUR_MAX` characters are stored.

If *s* is a null pointer, `wctomb` simply returns 0. If *s* is not a null pointer, `wctomb` returns −1 if the value of *wchar* does not correspond to a valid multibyte character; otherwise it returns the number of bytes that comprise the multibyte character corresponding to the value of *wchar*.

SEE ALSO

`mbstring`(3C), `setlocale`(3C), `environ`(5).

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

NAME

mbchar: mbtowc, wctomb, mblen – multibyte character conversion

SYNOPSIS

```
#include <stdlib.h>

int mbtowc(wchar_t *pwc, char *s, size_t n);

int wctomb(char *s, wchar_t wchar);

int mblen(char *s, size_t n);
```

DESCRIPTION

These three functions can support both

```
typedef unsigned short wchar_t;
```

and

```
typedef long wchar_t;
```

conditionally.

SEE ALSO

mbstring(3W), wchrtbl(1M).
mbchar(3C) in the *System V Release 4.0 Programmer's Reference Manual*.

NAME

mbstring: mbstowcs, wcstombs – multibyte string functions

SYNOPSIS

```
#include <stdlib.h>

size_t mbstowcs (wchar_t *pwcs, const char *s, size_t n);

size_t wcstombs (char *s, const wchar_t *pwcs, size_t n);
```

DESCRIPTION

mbstowcs converts a sequence of multibyte characters from the array pointed to by *s* into a sequence of corresponding wide character codes and stores these codes into the array pointed to by *pwcs*, stopping after *n* codes are stored or a code with value zero (a converted null character) is stored. If an invalid multibyte character is encountered, mbstowcs returns (size_t)–1. Otherwise, mbstowcs returns the number of array elements modified, not including the terminating zero code, if any.

wcstombs converts a sequence of wide character codes from the array pointed to by *pwcs* into a sequence of multibyte characters 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. If a wide character code is encountered that does not correspond to a valid multibyte character, wcstombs returns (size_t)–1. Otherwise, wcstombs returns the number of bytes modified, not including a terminating null character, if any.

SEE ALSO

mbchar(3C), setlocale(3C), environ(5).
chrtbl(1M) in the *System Administrator's Reference Manual.*

NAME

mbstring: mbstowcs, wctombs – multibyte string conversion

SYNOPSIS

```
#include <stdlib.h>

size_t mbstowcs(wchar_t *pwcs, char *s, size_t n);

size_t wcstombs(char *s, wchar_t *pwcs, size_t n);
```

DESCRIPTION

These two functions can support both

```
      typedef unsigned short wchar_t;
```

and

```
      typedef long wchar_t;
```

conditionally.

SEE ALSO

wchrtbl(1M), mbchar(3W), mbstring(3C).

NAME

memory: memccpy, memchr, memcmp, memcpy, memmove, memset – memory operations

SYNOPSIS

```
#include <string.h>

void *memccpy (void *s1, const void *s2, int c, size_t n);

void *memchr (const void *s, int c, size_t n);

int memcmp (const void *s1, const void *s2, size_t n);

void *memcpy (void *s1, const void *s2, size_t n);

void *memmove (void *s1, const void *s2, size_t n);

void *memset (void *s, int c, size_t n);
```

DESCRIPTION

These functions operate as efficiently as possible on memory areas (arrays of bytes bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

memccpy copies bytes from memory area *s2* into *s1*, stopping after the first occurrence of *c* (converted to an unsigned char) has been copied, or after *n* bytes have been copied, whichever comes first. It returns a pointer to the byte after the copy of *c* in *s1*, or a null pointer if *c* was not found in the first *n* bytes of *s2*.

memchr returns a pointer to the first occurrence of *c* (converted to an unsigned char) in the first *n* bytes (each interpreted as an unsigned char) of memory area *s*, or a null pointer if *c* does not occur.

memcmp compares its arguments, looking at the first *n* bytes (each interpreted as an unsigned char), 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* when taken to be unsigned characters.

memcpy copies *n* bytes from memory area *s2* to *s1*. It returns *s1*.

memmove copies *n* bytes from memory areas *s2* to *s1*. Copying between objects that overlap will take place correctly. It returns *s1*.

memset sets the first *n* bytes in memory area *s* to the value of *c* (converted to an unsigned char). It returns *s*.

SEE ALSO

string(3C).

NAME

menu_attributes: set_menu_fore, menu_fore, set_menu_back, menu_back, set_menu_grey, menu_grey, set_menu_pad, menu_pad – control menus display attributes

SYNOPSIS

```
#include <menu.h>

int set_menu_fore(MENU *menu, chtype attr);
chtype menu_fore(MENU *menu);

int set_menu_back(MENU *menu, chtype attr);
chtype menu_back(MENU *menu);

int set_menu_grey(MENU *menu, chtype attr);
chtype menu_grey(MENU *menu);

int set_menu_pad(MENU *menu, int pad);
int menu_pad(MENU *menu);
```

DESCRIPTION

set_menu_fore sets the foreground attribute of *menu* — the display attribute for the current item (if selectable) on single-valued menus and for selected items on multi-valued menus. This display attribute is a curses library visual attribute. menu_fore returns the foreground attribute of *menu*.

set_menu_back sets the background attribute of menu — the display attribute for unselected, yet selectable, items. This display attribute is a curses library visual attribute.

set_menu_grey sets the grey attribute of *menu* — the display attribute for nonselectable items in multi-valued menus. This display attribute is a curses library visual attribute. menu_grey returns the grey attribute of *menu*.

The pad character is the character that fills the space between the name and description of an item. set_menu_pad sets the pad character for *menu* to *pad*. menu_pad returns the pad character of *menu*.

RETURN VALUE

These routines return one of the following:

E_OK	– The routine returned successfully.
E_SYSTEM_ERROR	– System error.
E_BAD_ARGUMENT	– An incorrect argument was passed to the routine.

NOTES

The header file <menu.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), menus(3X).

NAME

menu_cursor: pos_menu_cursor – correctly position a menus cursor

SYNOPSIS

```
#include <menu.h>

int pos_menu_cursor(MENU *menu);
```

DESCRIPTION

pos_menu_cursor moves the cursor in the window of *menu* to the correct position to resume menu processing. This is needed after the application calls a curses library I/O routine.

RETURN VALUE

This routine returns one of the following:

E_OK	– The routine returned successfully.
E_SYSTEM_ERROR	– System error.
E_BAD_ARGUMENT	– An incorrect argument was passed to the routine.
E_NOT_POSTED	– The menu has not been posted.

NOTES

The header file <menu.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), menus(3X), panels(3X), panel_update(3X).

NAME

menu_driver - command processor for the menus subsystem

SYNOPSIS

```
#include <menu.h>

int menu_driver(MENU *menu, int c);
```

DESCRIPTION

menu_driver is the workhorse of the menus subsystem. It checks to determine whether the character *c* is a menu request or data. If *c* is a request, the menu driver executes the request and reports the result. If *c* is data (a printable ASCII character), it enters the data into the pattern buffer and tries to find a matching item. If no match is found, the menu driver deletes the character from the pattern buffer and returns E_NO_MATCH. If the character is not recognized, the menu driver assumes it is an application-defined command and returns E_UNKNOWN_COMMAND.

Menu driver requests:

REQ_LEFT_ITEM	Move left to an item.
REQ_RIGHT_ITEM	Move right to an item.
REQ_UP_ITEM	Move up to an item.
REQ_DOWN_ITEM	Move down to an item.
REQ_SCR_ULINE	Scroll up a line.
REQ_SCR_DLINE	Scroll down a line.
REQ_SCR_DPAGE	Scroll up a page.
REQ_SCR_UPAGE	Scroll down a page.
REQ_FIRST_ITEM	Move to the first item.
REQ_LAST_ITEM	Move to the last item.
REQ_NEXT_ITEM	Move to the next item.
REQ_PREV_ITEM	Move to the previous item.
REQ_TOGGLE_ITEM	Select/de-select an item.
REQ_CLEAR_PATTERN	Clear the menu pattern buffer.
REQ_BACK_PATTERN	Delete the previous character from pattern buffer.
REQ_NEXT_MATCH	Move the next matching item.
REQ_PREV_MATCH	Move to the previous matching item.

RETURN VALUE

menu_driver returns one of the following:

E_OK	- The routine returned successfully.
E_SYSTEM_ERROR	- System error.
E_BAD_ARGUMENT	- An incorrect argument was passed to the routine.
E_BAD_STATE	- The routine was called from an initialization or termination function.
E_NOT_POSTED	- The menu has not been posted.

E_UNKNOWN_COMMAND – An unknown request was passed to the menu driver.
E_NO_MATCH – The character failed to match.
E_NOT_SELECTABLE – The item cannot be selected.
E_REQUEST_DENIED – The menu driver could not process the request.

NOTES

Application defined commands should be defined relative to (greater than) MAX_COMMAND, the maximum value of a request listed above.

The header file <menu.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), menus(3X).

NAME

menu_format: set_menu_format, menu_format – set and get maximum numbers of rows and columns in menus

SYNOPSIS

```
#include <menu.h>

int set_menu_format(MENU *menu, int rows, int cols);

void menu_format(MENU *menu, int *rows, int *cols);
```

DESCRIPTION

set_menu_format sets the maximum number of rows and columns of items that may be displayed at one time on a menu. If the menu contains more items than can be displayed at once, the menu will be scrollable.

menu_format returns the maximum number of rows and columns that may be displayed at one time on *menu*. *rows* and *cols* are pointers to the variables used to return these values.

RETURN VALUE

set_menu_format returns one of the following:

E_OK	– The routine returned successfully.
E_SYSTEM_ERROR	– System error.
E_BAD_ARGUMENT	– An incorrect argument was passed to the routine.
E_POSTED	– The menu is already posted.

NOTES

The header file <menu.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), menus(3X).

NAME

menu_hook: set_item_init, item_init, set_item_term, item_term, set_menu_init, menu_init, set_menu_term, menu_term – assign application-specific routines for automatic invocation by menus

SYNOPSIS

```
#include <menu.h>

int set_item_init(MENU *menu, void (*func)(MENU *));
void (*)(MENU *) item_init(MENU *menu);

int set_item_term(MENU *menu, void (*func)(MENU *));
void (*)(MENU *) item_term(MENU *menu);

int set_menu_init(MENU *menu, void (*func)(MENU *));
void (*)(MENU *) menu_init(MENU *menu);

int set_menu_term(MENU *menu, void (*func)(MENU *));
void (*)(MENU *) menu_term(MENU *menu);
```

DESCRIPTION

set_item_init assigns the application-defined function to be called when the *menu* is posted and just after the current item changes. item_init returns a pointer to the item initialization routine, if any, called when the *menu* is posted and just after the current item changes.

set_item_term assigns an application-defined function to be called when the *menu* is unposted and just before the current item changes. item_term returns a pointer to the termination function, if any, called when the *menu* is unposted and just before the current item changes.

set_menu_init assigns an application-defined function to be called when the *menu* is posted and just after the top row changes on a posted menu. menu_init returns a pointer to the menu initialization routine, if any, called when the *menu* is posted and just after the top row changes on a posted menu.

set_menu_term assigns an application-defined function to be called when the *menu* is unposted and just before the top row changes on a posted menu. menu_term returns a pointer to the menu termination routine, if any, called when the *menu* is unposted and just before the top row changes on a posted menu.

RETURN VALUE

Routines that return pointers always return NULL on error. Routines that return an integer return one of the following:

E_OK	– The routine returned successfully.
E_SYSTEM_ERROR	– System error.

NOTES

The header file <menu.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), menus(3X), menu_control(3X), menu_hook(3X).

NAME

menu_item_current: set_current_item, current_item, set_top_row, top_row, item_index – set and get current menus items

SYNOPSIS

```
#include <menu.h>

int set_current_item(MENU *menu, ITEM *item);
ITEM *current_item(MENU *menu);

int set_top_row(MENU *menu, int row);
int top_row(MENU *menu);

int item_index(ITEM *item);
```

DESCRIPTION

The current item of a menu is the item where the cursor is currently positioned. set_current_item sets the current item of *menu* to *item*. current_item returns a pointer to the the current item in *menu*.

set_top_row sets the top row of *menu* to *row*. The left-most item on the new top row becomes the current item. top_row returns the number of the menu row currently displayed at the top of *menu*.

item_index returns the index to the *item* in the item pointer array. The value of this index ranges from 0 through *N*-1, where *N* is the total number of items connected to the menu.

RETURN VALUE

current_item returns NULL on error.

top_row and index_item return -1 on error.

set_current_item and set_top_row return one of the following:

E_OK	– The routine returned successfully.
E_SYSTEM_ERROR	– System error.
E_BAD_ARGUMENT	– An incorrect argument was passed to the routine.
E_BAD_STATE	– The routine was called from an initialization or termination function.
E_NOT_CONNECTED	– No items are connected to the menu.

NOTES

The header file <menu.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), menus(3X).

NAME

menu_item_name: item_name, item_description – get menus item name and description

SYNOPSIS

```
#include <menu.h>

char *item_name(ITEM *item);

char *item_description(ITEM *item);
```

DESCRIPTION

item_name returns a pointer to the name of *item*.

item_description returns a pointer to the description of *item*.

RETURN VALUE

These routines return NULL on error.

NOTES

The header file <menu.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), menus(3X), menu_new(3X).

NAME

menu_item_new: new_item, free_item – create and destroy menus items

SYNOPSIS

```
#include <menu.h>

ITEM *new_item(char *name, char *desc);

int free_item(ITEM *item);
```

DESCRIPTION

new_item creates a new item from *name* and *description*, and returns a pointer to the new item.

free_item frees the storage allocated for *item*. Once an item is freed, the user can no longer connect it to a menu.

RETURN VALUE

new_item returns NULL on error.

free_item returns one of the following:

E_OK	– The routine returned successfully.
E_SYSTEM_ERROR	– System error.
E_BAD_ARGUMENT	– An incorrect argument was passed to the routine.
E_CONNECTED	– One or more items are already connected to another menu.

NOTES

The header file <menu.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), menus(3X).

NAME

menu_item_opts: set_item_opts, item_opts_on, item_opts_off, item_opts − menus item option routines

SYNOPSIS

```
#include <menu.h>

int set_item_opts(ITEM *item, OPTIONS opts);
int item_opts_on(ITEM *item, OPTIONS opts);
int item_opts_off(ITEM *item, OPTIONS opts);
OPTIONS item_opts(ITEM *item);
```

DESCRIPTION

set_item_opts turns on the named options for *item* and turns off all other options. Options are boolean values that can be OR-ed together.

item_opts_on turns on the named options for *item*; no other option is changed.

item_opts_off turns off the named options for *item*; no other option is changed.

item_opts returns the current options of *item*.

Item Options:

O_SELECTABLE
The item can be selected during menu processing.

RETURN VALUE

Except for item_opts, these routines return one of the following:

E_OK	− The routine returned successfully.
E_SYSTEM_ERROR	− System error.

NOTES

The header file <menu.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), menus(3X).

NAME

menu_item_userptr: set_item_userptr, item_userptr – associate application data with menus items

SYNOPSIS

```
#include <menu.h>

int set_item_userptr(ITEM *item, char *userptr);

char *item_userptr(ITEM *item);
```

DESCRIPTION

Every item has an associated user pointer that can be used to store relevant information. set_item_userptr sets the user pointer of *item*. item_userptr returns the user pointer of *item*.

RETURN VALUE

item_userptr returns NULL on error. set_item_userptr returns one of the following:

E_OK	– The routine returned successfully.
E_SYSTEM_ERROR	– System error.

NOTES

The header file <menu.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), menus(3X).

NAME

menu_item_value: set_item_value, item_value – set and get menus item values

SYNOPSIS

```
#include <menu.h>

int set_item_value(ITEM *item, int bool);

int item_value(ITEM *item);
```

DESCRIPTION

Unlike single-valued menus, multi-valued menus enable the end-user to select one or more items from a menu. set_item_value sets the selected value of the *item* — TRUE (selected) or FALSE (not selected). set_item_value may be used only with multi-valued menus. To make a menu multi-valued, use set_menu_opts or menu_opts_off to turn off the option O_ONEVALUE. [see menu_opts(3X)].

item_value returns the select value of *item*, either TRUE (selected) or FALSE (unselected).

RETURN VALUE

set_item_value returns one of the following:

E_OK	– The routine returned successfully.
E_SYSTEM_ERROR	– System error.
E_REQUEST_DENIED	– The menu driver could not process the request.

NOTES

The header file <menu.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), menus(3X), menu_opts(3X).

NAME

menu_item_visible: item_visible – tell if menus item is visible

SYNOPSIS

```
#include <menu.h>

int item_visible(ITEM *item);
```

DESCRIPTION

A menu item is visible if it currently appears in the subwindow of a posted menu. item_visible returns TRUE if *item* is visible, otherwise it returns FALSE.

NOTES

The header file <menu.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), menus(3X), menu_new(3X).

NAME

menu_items: set_menu_items, menu_items, item_count – connect and disconnect items to and from menus

SYNOPSIS

```
#include <menu.h>

int set_menu_items(MENU *menu, ITEM **items);

ITEM **menu_items(MENU *menu);

int item_count(MENU *menu);
```

DESCRIPTION

set_menu_items changes the item pointer array connected to *menu* to the item pointer array *items*.

menu_items returns a pointer to the item pointer array connected to *menu*.

item_count returns the number of items in *menu*.

RETURN VALUE

menu_items returns NULL on error.

item_count returns -1 on error.

set_menu_items returns one of the following:

E_OK	– The routine returned successfully.
E_SYSTEM_ERROR	– System error.
E_BAD_ARGUMENT	– An incorrect argument was passed to the routine.
E_POSTED	– The menu is already posted.
E_CONNECTED	– One or more items are already connected to another menu.

NOTES

The header file <menu.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), menus(3X).

NAME

menu_mark: set_menu_mark, menu_mark – menus mark string routines

SYNOPSIS

```
#include <menu.h>

int set_menu_mark(MENU *menu, char *mark);

char *menu_mark(MENU *menu);
```

DESCRIPTION

menus displays mark strings to distinguish selected items in a menu (or the current item in a single-valued menu). set_menu_mark sets the mark string of *menu* to *mark*. menu_mark returns a pointer to the mark string of *menu*.

RETURN VALUE

menu_mark returns NULL on error. set_menu_mark returns one of the following:

E_OK	– The routine returned successfully.
E_SYSTEM_ERROR	– System error.
E_BAD_ARGUMENT	– An incorrect argument was passed to the routine.

NOTES

The header file <menu.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), menus(3X).

NAME

menu_new: new_menu, free_menu – create and destroy menus

SYNOPSIS

```
#include <menu.h>

MENU *new_menu(ITEM **items);

int free_menu(MENU *menu);
```

DESCRIPTION

new_menu creates a new menu connected to the item pointer array *items* and returns a pointer to the new menu.

free_menu disconnects *menu* from its associated item pointer array and frees the storage allocated for the menu.

RETURN VALUE

new_menu returns NULL on error.

free_menu returns one of the following:

E_OK	– The routine returned successfully.
E_SYSTEM_ERROR	– System error.
E_BAD_ARGUMENT	– An incorrect argument was passed to the routine.
E_POSTED	– The menu is already posted.

NOTES

The header file <menu.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), menus(3X).

NAME

menu_opts: set_menu_opts, menu_opts_on, menu_opts_off, menu_opts – menus option routines

SYNOPSIS

```
#include <menu.h>

int set_menu_opts(MENU *menu, OPTIONS opts);
int menu_opts_on(MENU *menu, OPTIONS opts);
int menu_opts_off(MENU *menu, OPTIONS opts);
OPTIONS menu_opts(MENU *menu);
```

DESCRIPTION

Menu Options

set_menu_opts turns on the named options for *menu* and turns off all other options. Options are boolean values that can be OR-ed together.

menu_opts_on turns on the named options for *menu*; no other option is changed.

menu_opts_off turns off the named options for *menu*; no other option is changed.

menu_opts returns the current options of *menu*.

Menu Options:

O_ONEVALUE	Only one item can be selected from the menu.
O_SHOWDESC	Display the description of the items.
O_ROWMAJOR	Display the menu in row major order.
O_IGNORECASE	Ignore the case when pattern matching.
O_SHOWMATCH	Place the cursor within the item name when pattern matching.
O_NONCYCLIC	Make certain menu driver requests non-cyclic.

RETURN VALUE

Except for menu_opts, these routines return one of the following:

E_OK	– The routine returned successfully.
E_SYSTEM_ERROR	– System error.
E_POSTED	– The menu is already posted.

NOTES

The header file <menu.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), menus(3X).

NAME

menu_pattern: set_menu_pattern, menu_pattern – set and get menus pattern match buffer

SYNOPSIS

```
#include <menu.h>

int set_menu_pattern(MENU *menu, char *pat);

char *menu_pattern(MENU *menu);
```

DESCRIPTION

Every menu has a pattern buffer to match entered data with menu items. set_menu_pattern sets the pattern buffer to *pat* and tries to find the first item that matches the pattern. If it does, the matching item becomes the current item. If not, the current item does not change. menu_pattern returns the string in the pattern buffer of *menu*.

RETURN VALUE

menu_pattern returns NULL on error. set_menu_pattern returns one of the following:

E_OK	– The routine returned successfully.
E_SYSTEM_ERROR	– System error.
E_BAD_ARGUMENT	– An incorrect argument was passed to the routine.
E_NO_MATCH	– The character failed to match.

NOTES

The header file <menu.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), menus(3X).

NAME

menu_post: post_menu, unpost_menu – write or erase menus from associated subwindows

SYNOPSIS

```
#include <menu.h>

int post_menu(MENU *menu);

int unpost_menu(MENU *menu);
```

DESCRIPTION

post_menu writes *menu* to the subwindow. The application programmer must use curses library routines to display the menu on the physical screen or call update_panels if the panels library is being used.

unpost_menu erases *menu* from its associated subwindow.

RETURN VALUE

These routines return one of the following:

E_OK	– The routine returned successfully.
E_SYSTEM_ERROR	– System error.
E_BAD_ARGUMENT	– An incorrect argument was passed to the routine.
E_POSTED	– The menu is already posted.
E_BAD_STATE	– The routine was called from an initialization or termination function.
E_NO_ROOM	– The menu does not fit within its subwindow.
E_NOT_POSTED	– The menu has not been posted.
E_NOT_CONNECTED	– No items are connected to the menu.

NOTES

The header file <menu.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), menus(3X), panels(3X).

NAME

menu_userptr: set_menu_userptr, menu_userptr – associate application data with menus

SYNOPSIS

```
#include <menu.h>

int set_menu_userptr(MENU *menu, char *userptr);
char *menu_userptr(MENU *menu);
```

DESCRIPTION

Every menu has an associated user pointer that can be used to store relevant information. set_menu_userptr sets the user pointer of *menu*. menu_userptr returns the user pointer of *menu*.

RETURN VALUE

menu_userptr returns NULL on error.

set_menu_userptr returns one of the following:

E_OK	– The routine returned successfully.
E_SYSTEM_ERROR	– System error.

NOTES

The header file <menu.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), menus(3X).

NAME

menu_win: set_menu_win, menu_win, set_menu_sub, menu_sub, scale_menu – menus window and subwindow association routines

SYNOPSIS

```
#include <menu.h>

int set_menu_win(MENU *menu, WINDOW *win);
WINDOW *menu_win(MENU *menu);

int set_menu_sub(MENU *menu, WINDOW *sub);
WINDOW *menu_sub(MENU *menu);

int scale_window(MENU *menu, int *rows, int *cols);
```

DESCRIPTION

set_menu_win sets the window of *menu* to *win*. menu_win returns a pointer to the window of *menu*.

set_menu_sub sets the subwindow of *menu* to *sub*. menu_sub returns a pointer to the subwindow of *menu*.

scale_window returns the minimum window size necessary for the subwindow of *menu*. *rows* and *cols* are pointers to the locations used to return the values.

RETURN VALUE

Routines that return pointers always return NULL on error. Routines that return an integer return one of the following:

E_OK	– The routine returned successfully.
E_SYSTEM_ERROR	– System error.
E_BAD_ARGUMENT	– An incorrect argument was passed to the routine.
E_POSTED	– The menu is already posted.
E_NOT_CONNECTED	– No items are connected to the menu.

NOTES

The header file <menu.h> automatically includes the header files <eti.h> and <curses.h>.

SEE ALSO

curses(3X), menus(3X).

NAME

menus – character based menus package

SYNOPSIS

```
#include <menu.h>
```

DESCRIPTION

The menu library is built using the curses library, and any program using menus routines must call one of the curses initialization routines, such as initscr. A program using these routines must be compiled with -lmenu and -lcurses on the cc command line.

The menus package gives the applications programmer a terminal-independent method of creating and customizing menus for user interaction. The menus package includes: item routines, which are used to create and customize menu items; and menu routines, which are used to create and customize menus, assign pre- and post-processing routines, and display and interact with menus.

Current Default Values for Item Attributes

The menus package establishes initial current default values for item attributes. During item initialization, each item attribute is assigned the current default value for that attribute. An application can change or retrieve a current default attribute value by calling the appropriate set or retrieve routine with a NULL item pointer. If an application changes a current default item attribute value, subsequent items created using new_item will have the new default attribute value. (The attributes of previously created items are not changed if a current default attribute value is changed.)

Routine Name Index

The following table lists each menus routine and the name of the manual page on which it is described.

menus Routine Name	Manual Page Name
current_item	menu_item_current(3X)
free_item	menu_item_new(3X)
free_menu	menu_new(3X)
item_count	menu_items(3X)
item_description	menu_item_name(3X)
item_index	menu_item_current(3X)
item_init	menu_hook(3X)
item_name	menu_item_name(3X)
item_opts	menu_item_opts(3X)
item_opts_off	menu_item_opts(3X)
item_opts_on	menu_item_opts(3X)
item_term	menu_hook(3X)
item_userptr	menu_item_userptr(3X)
item_value	menu_item_value(3X)
item_visible	menu_item_visible(3X)
menu_back	menu_attributes(3X)
menu_driver	menu_driver(3X)
menu_fore	menu_attributes(3X)
menu_format	menu_format(3X)

menu_grey	menu_attributes(3X)
menu_init	menu_hook(3X)
menu_items	menu_items(3X)
menu_mark	menu_mark(3X)
menu_opts	menu_opts(3X)
menu_opts_off	menu_opts(3X)
menu_opts_on	menu_opts(3X)
menu_pad	menu_attributes(3X)
menu_pattern	menu_pattern(3X)
menu_sub	menu_win(3X)
menu_term	menu_hook(3X)
menu_userptr	menu_userptr(3X)
menu_win	menu_win(3X)
new_item	menu_item_new(3X)
new_menu	menu_new(3X)
pos_menu_cursor	menu_cursor(3X)
post_menu	menu_post(3X)
scale_menu	menu_win(3X)
set_current_item	menu_item_current(3X)
set_item_init	menu_hook(3X)
set_item_opts	menu_item_opts(3X)
set_item_term	menu_hook(3X)
set_item_userptr	menu_item_userptr(3X)
set_item_value	menu_item_value(3X)
set_menu_back	menu_attributes(3X)
set_menu_fore	menu_attributes(3X)
set_menu_format	menu_format(3X)
set_menu_grey	menu_attributes(3X)
set_menu_init	menu_hook(3X)
set_menu_items	menu_items(3X)
set_menu_mark	menu_mark(3X)
set_menu_opts	menu_opts(3X)
set_menu_pad	menu_attributes(3X)
set_menu_pattern	menu_pattern(3X)
set_menu_sub	menu_win(3X)
set_menu_term	menu_hook(3X)
set_menu_userptr	menu_userptr(3X)
set_menu_win	menu_win(3X)
set_top_row	menu_item_current(3X)
top_row	menu_item_current(3X)
unpost_menu	menu_post(3X)

RETURN VALUE

Routines that return pointers always return NULL on error. Routines that return an integer return one of the following:

E_OK – The routine returned successfully.

E_SYSTEM_ERROR – System error.

E_BAD_ARGUMENT – An incorrect argument was passed to the routine.

`E_POSTED` – The menu is already posted.
`E_CONNECTED` – One or more items are already connected to another menu.
`E_BAD_STATE` – The routine was called from an initialization or termination function.
`E_NO_ROOM` – The menu does not fit within its subwindow.
`E_NOT_POSTED` – The menu has not been posted.
`E_UNKNOWN_COMMAND` – An unknown request was passed to the menu driver.
`E_NO_MATCH` – The character failed to match.
`E_NOT_SELECTABLE` – The item cannot be selected.
`E_NOT_CONNECTED` – No items are connected to the menu.
`E_REQUEST_DENIED` – The menu driver could not process the request.

NOTES

The header file `<menu.h>` automatically includes the header files `<eti.h>` and `<curses.h>`.

SEE ALSO

`curses`(3X), and 3X pages whose names begin "menu_" for detailed routine descriptions.

NAME

mkdirp, rmdirp – create, remove directories in a path

SYNOPSIS

cc [*flag* ...] *file* ... -lgen [*library* ...]

```
#include <libgen.h>
```

```
int mkdirp (const char *path, mode_t mode);
```

```
int rmdirp (char *d, char *dl);
```

DESCRIPTION

mkdirp creates all the missing directories in the given *path* with the given *mode*. [See chmod(2) for the values of *mode.*]

rmdirp removes directories in path *d*. This removal starts at the end of the path and moves back toward the root as far as possible. If an error occurs, the remaining path is stored in *dl*. rmdirp returns a 0 only if it is able to remove every directory in the path.

EXAMPLES

```
/* create scratch directories */
if(mkdirp("/tmp/sub1/sub2/sub3", 0755) == -1) {
      fprintf(stderr, "cannot create directory");
      exit(1);
}
chdir("/tmp/sub1/sub2/sub3");
.
.
.
/* cleanup */
chdir("/tmp");
rmdirp("sub1/sub2/sub3");
```

SEE ALSO

mkdir(2), rmdir(2).

DIAGNOSTICS

If a needed directory cannot be created, mkdirp returns -1 and sets errno to one of the mkdir error numbers. If all the directories are created, or existed to begin with, it returns zero.

NOTES

mkdirp uses malloc to allocate temporary space for the string.

rmdirp returns -2 if a "." or ".." is in the path and -3 if an attempt is made to remove the current directory. If an error occurs other than one of the above, -1 is returned.

NAME

mkfifo – create a new FIFO

SYNOPSIS

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

int mkfifo (const char *path, mode_t mode);
```

DESCRIPTION

The mkfifo routine creates a new FIFO special file named by the pathname pointed to by *path*. The mode of the new FIFO is initialized from *mode*. The file permission bits of the *mode* argument are modified by the process's file creation mask [see umask(2)].

The FIFO's owner id is set to the process's effective user id. The FIFO's group id is set to the process's effective group id, or if the S_ISGID bit is set in the parent directory then the group id of the FIFO is inherited from the parent.

mkfifo calls the system call mknod to make the file.

SEE ALSO

chmod(2), exec(2), mknod(2), umask(2), fs(4), stat(5).
mkdir(1) in the *User's Reference Manual*.

DIAGNOSTICS

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

NOTES

Bits other than the file permission bits in *mode* are ignored.

NAME

`mkstemp` – make a unique file name

SYNOPSIS

```
mkstemp(template)
char *template;
```

DESCRIPTION

`Mkstemp` creates a unique file name, typically in a temporary filesystem, by replacing *template* with a unique file name and returning a file descriptor for the template file open for reading and writing. The template should contain a file name with six trailing X's, which are replaced with the current process id and a unique letter. `Mkstemp` avoids the race between testing whether the file exists and opening it for use.

DIAGNOSTICS

`Mkstemp` returns an open file descriptor upon success. It returns –1 if no suitable file could be created.

SEE ALSO

`getpid`(2), `open`(2).

NAME

mktemp – make a unique file name

SYNOPSIS

```
#include <stdlib.h>

char *mktemp(char *template);
```

DESCRIPTION

mktemp replaces the contents of the string pointed to by *template* with a unique file name, and returns *template*. The string in *template* should look like a file name with six trailing Xs; mktemp will replace the Xs with a character string that can be used to create a unique file name.

SEE ALSO

tmpfile(3S), tmpnam(3S).

DIAGNOSTIC

mktemp will assign to *template* the empty string if it cannot create a unique name.

NOTES

mktemp can create only 26 unique file names per process for each unique *template*.

NAME

`mlock`, `munlock` – lock (or unlock) pages in memory

SYNOPSIS

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

int mlock(caddr_t addr, size_t len);

int munlock(caddr_t addr, size_t len);
```

DESCRIPTION

The function `mlock` uses the mappings established for the address range [*addr*, *addr* + *len*) to identify pages to be locked in memory. The effect of `mlock(`*addr*`,` *len*`)` is equivalent to `memcntl(`*addr*`,` *len*`, MC_LOCK, 0, 0, 0)`.

`munlock` removes locks established with `mlock`. The effect of `munlock(`*addr*`,` *len*`)` is equivalent to `memcntl(`*addr*`,` *len*`, MC_UNLOCK, 0, 0, 0)`.

Locks established with `mlock` are not inherited by a child process after a `fork` and are not nested.

RETURN VALUE

Upon successful completion, the functions `mlock` and `munlock` return a value of `0`; otherwise, they return a value of `-1` and set `errno` to indicate an error.

DIAGNOSTICS

See `memcntl`(2).

NOTES

Use of `mlock` and `munlock` requires that the user have appropriate privileges.

SEE ALSO

`fork`(2), `memcntl`(2), `mlockall`(3C), `plock`(2), `sysconf`(3C).

NAME

mlockall, munlockall – lock or unlock address space

SYNOPSIS

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

int mlockall(int flags);

int munlockall(void);
```

DESCRIPTION

The function mlockall causes all pages mapped by an address space to be locked in memory. The effect of mlockall(*flags*) is equivalent to:

```
memcntl(0, 0, MC_LOCKAS, flags, 0, 0)
```

The value of *flags* determines whether the pages to be locked are those currently mapped by the address space, those that will be mapped in the future, or both:

MCL_CURRENT	Lock current mappings
MCL_FUTURE	Lock future mappings

The function munlockall removes address space locks and locks on mappings in the address space. The effect of munlockall is equivalent to:

```
memcntl(0, 0, MC_UNLOCKAS, 0, 0, 0)
```

Locks established with mlockall are not inherited by a child process after a fork and are not nested.

RETURN VALUE

Upon successful completion, the functions mlockall and munlockall return a value of 0; otherwise, they return a value of -1 and set errno to indicate an error.

DIAGNOSTICS

See memcntl(2).

NOTES

Use of mlockall and munlockall requires that the user have appropriate privileges.

SEE ALSO

fork(2), memcntl(2), mlock(3C), plock(2), sysconf(3C).

NAME

monitor – prepare execution profile

SYNOPSIS

```
#include <mon.h>

void monitor (int (*lowpc)( ), int (*highpc)( ), WORD *buffer,
    size_t bufsize, size_t nfunc);
```

DESCRIPTION

monitor is an interface to profil, and is called automatically with default parameters by any program created by cc -p. Except to establish further control over profiling activity, it is not necessary to explicitly call monitor.

When used, monitor is called at least at the beginning and the end of a program. The first call to monitor initiates the recording of two different kinds of execution-profile information: execution-time distribution and function call count. Execution-time distribution data is generated by profil and the function call counts are generated by code supplied to the object file (or files) by cc -p. Both types of information are collected as a program executes. The last call to monitor writes this collected data to the output file mon.out.

lowpc and *highpc* are the beginning and ending addresses of the region to be profiled.

buffer is the address of a user-supplied array of WORD (WORD is defined in the header file mon.h). *buffer* is used by monitor to store the histogram generated by profil and the call counts.

bufsize identifies the number of array elements in *buffer*.

nfunc is the number of call count cells that have been reserved in *buffer*. Additional call count cells will be allocated automatically as they are needed.

bufsize should be computed using the following formula:

```
size_of_buffer =
        sizeof(struct hdr) +
        nfunc * sizeof(struct cnt) +
        ((highpc-lowpc)/BARSIZE) * sizeof(WORD) +
        sizeof(WORD) - 1 ;

bufsize = (size_of_buffer / sizeof(WORD)) ;
```

where:

lowpc, *highpc*, *nfunc* are the same as the arguments to monitor;

BARSIZE is the number of program bytes that correspond to each histogram bar, or cell, of the profil buffer;

the hdr and cnt structures and the type WORD are defined in the header file mon.h.

The default call to monitor is shown below:

```
monitor (&eprol, &etext, wbuf, wbufsz, 600);
```

where:

eprol is the beginning of the user's program when linked with cc -p [see end(3C)];

etext is the end of the user's program [see end(3C)];

wbuf is an array of WORD with *wbufsz* elements;

wbufsz is computed using the *bufsize* formula shown above with *BARSIZE* of 8;

600 is the number of call count cells that have been reserved in *buffer*.

These parameter settings establish the computation of an execution-time distribution histogram that uses profil for the entire program, initially reserves room for 600 call count cells in *buffer*, and provides for enough histogram cells to generate significant distribution-measurement results. [For more information on the effects of *bufsize* on execution-distribution measurements, see profil(2).]

To stop execution monitoring and write the results to a file, use the following:

```
monitor((int (*)())0, (int (*)())0, (WORD *)0, 0, 0);
```

Use prof to examine the results.

FILES

mon.out

SEE ALSO

cc(1), prof(1), profil(2), end(3C).

NOTE

Additional calls to monitor after main has been called and before exit has been called will add to the function-call count capacity, but such calls will also replace and restart the profil histogram computation.

The name of the file written by monitor is controlled by the environment variable PROFDIR. If PROFDIR does not exist, the file mon.out is created in the current directory. If PROFDIR exists but has no value, monitor does no profiling and creates no output file. If PROFDIR is *dirname*, and monitor is called automatically by compilation with cc -p, the file created is *dirname/pid.progname* where *progname* is the name of the program.

NAME

mp: madd, msub, mult, mdiv, pow, gcd, invert, rpow, msqrt, mcmp, move, min, omin, fmin, m_in, mout, omout, fmout, m_out, sdiv, itom – multiple precision integer arithmetic

SYNOPSIS

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

typedef struct mint { int len; short *val; } MINT;

madd(a, b, c)
msub(a, b, c)
mult(a, b, c)
mdiv(a, b, q, r)
pow(a, b, m, c)
gcd(a, b, c)
invert(a, b, c)
rpow(a, n, c)
msqrt(a, b, r)
mcmp(a, b)
move(a, b)
min(a)
omin(a)
fmin(a, f)
m_in(a, n, f)
mout(a)
omout(a)
fmout(a, f)
m_out(a, n, f)
MINT *a, *b, *c, *m, *q, *r;
FILE *f;
int n;

sdiv(a, n, q, r)
MINT *a, *q;
short n;
short *r;

MINT *itom(n)
```

DESCRIPTION

These routines perform arithmetic on integers of arbitrary length. The integers are stored using the defined type MINT. Pointers to a MINT can be initialized using the function itom which sets the initial value to *n*. After that, space is managed automatically by the routines.

madd, msub and mult assign to *c* the sum, difference and product, respectively, of *a* and *b*. mdiv assigns to *q* and *r* the quotient and remainder obtained from dividing *a* by *b*. sdiv is like mdiv except that the divisor is a short integer *n* and the remainder is placed in a short whose address is given as *r*. msqrt produces the integer square root of *a* in *b* and places the remainder in *r*. rpow calculates in *c* the value of *a* raised to the ("regular" integral) power *n*, while pow calculates this with a full multiple precision exponent *b* and the result is reduced modulo *m*. gcd returns the greatest common denominator of *a* and *b* in *c*, and invert computes *c* such that *a***c* mod *b* = 1, for *a* and *b* relatively prime. mcmp returns a negative, zero or positive integer value when *a* is less than, equal to or greater than *b*, respectively. *move* copies BIR a " to " b . `min` and `mout` do decimal input and output while `omin` and `omout` do octal input and output. More generally, `fmin` and `fmout` do decimal input and output using file *f*, and *m_in* and *m_out* do I/O with arbitrary radix *n*. On input, records should have the form of strings of digits terminated by a newline; output records have a similar form.

Programs which use the multiple-precision arithmetic library must be loaded using the loader flag `-lmp`.

FILES

/usr/include/mp.h	include file
/usr/lib/libmp.a	object code library

SEE ALSO

`bc(1)`, `dc(1)`.

DIAGNOSTICS

Illegal operations and running out of memory produce messages and core images.

BUGS

Bases for input and output should be <= 10.

`dc(1)` and `bc(1)` don't use this library.

pow is also the name of a standard math library routine.

NAME

msync – synchronize memory with physical storage

SYNOPSIS

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

int msync(caddr_t addr, size_t len, int flags);
```

DESCRIPTION

The function msync writes all modified copies of pages over the range [*addr*, *addr* + *len*) to their backing storage locations. msync optionally invalidates any copies so that further references to the pages will be obtained by the system from their backing storage locations. The backing storage for a modified MAP_SHARED mapping is the file the page is mapped to; the backing storage for a modified MAP_PRIVATE mapping is its swap area.

flags is a bit pattern built from the following values:

MS_ASYNC	perform asynchronous writes
MS_SYNC	perform synchronous writes
MS_INVALIDATE	invalidate mappings

If MS_ASYNC is set, msync returns immediately once all write operations are scheduled; if MS_SYNC is set, msync does not return until all write operations are completed.

MS_INVALIDATE invalidates all cached copies of data in memory, so that further references to the pages will be obtained by the system from their backing storage locations.

The effect of msync(*addr*, *len*, *flags*) is equivalent to:

memcntl(*addr*, *len*, MC_SYNC, *flags*, 0, 0)

SEE ALSO

memcntl(2), mmap(2), sysconf(3C).

DIAGNOSTICS

Upon successful completion, the function msync returns 0; otherwise, it returns −1 and sets errno to indicate the error.

NOTES

msync should be used by programs that require a memory object to be in a known state, for example, in building transaction facilities.

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(2)).

Once open, the data stored under a key is accessed by *dbm_fetch* and data is placed under a key by *dbm_store*. The *flags* field can be either DBM_INSERT or DBM_REPLACE. DBM_INSERT 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.

DIAGNOSTICS

All functions that return an *int* indicate errors with negative values. A zero return indicates ok. Routines that return a *datum* indicate errors with a null (0) *dptr*. 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 UNIX systems may create real file blocks for these holes when touched. These files cannot be copied by normal means (cp, cat, tp, tar, ar) without filling in the holes.

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

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 1024 bytes - 6 bytes of overhead). 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.

SEE ALSO

dbm(3X)

NAME

netdir: netdir_getbyname, netdir_getbyaddr, netdir_free, netdir_mergeaddr, taddr2uaddr, uaddr2taddr, netdir_perror, netdir_sperror – generic transport name-to-address translation

SYNOPSIS

```
#include <netdir.h>

int
netdir_getbyname(config, service, addrs)
      struct netconfig  *config;
      struct nd_hostserv  *service;
      struct nd_addrlist  **addrs;

int
netdir_getbyaddr(config, service, netaddr)
      struct netconfig  *config;
      struct nd_hostservlist  **service;
      struct netbuf  *netaddr;

void
netdir_free(ptr, ident)
      void *ptr;
      int ident;

int
netdir_mergeaddr(config, mrg_uaddr, s_uaddr, c_uaddr)
      struct netconfig  *config;
      char  **mrg_uaddr, *s_uaddr, *c_uaddr;

char *
taddr2uaddr(config, addr)
      struct netconfig  *config;
      struct netbuf  *addr;

struct netbuf *
uaddr2taddr(config, uaddr)
      struct netconfig  *config;
      char  *uaddr;

int
netdir_options(netconfig, option, fd, pointer_to_args)
      struct netconfig  *netconfig;
      int  option;
      int fd;
      char  *point_to_args;

void
netdir_perror(s)
char  *s;

char *
netdir_sperror()
```

DESCRIPTION

These routines provide a generic interface for name-to-address mapping that will work with a all transport protocols. This interface provides a generic way for programs to convert transport specific addresses into common structures and back again.

The netdir_getbyname( ) routine maps the machine name and service name in the nd_hostserv structure to a collection of addresses of the type understood by the transport identified in the netconfig structure. This routine returns all addresses that are valid for that transport in the nd_addrlist structure. The nd_hostserv and nd_addrlist structures have the following elements. The netconfig structure is described on the netconfig(4) manual page.

```
struct nd_addrlist
      int n_cnt
      struct netbuf  *n_addrs;

struct nd_hostserv
      char  *h_host;
      char  *h_serv;
```

netdir_getbyname( ) accepts some special-case host names. These host names are hints to the underlying mapping routines that define the intent of the request. This information is required for some transport provider developers to provide the correct information back to the caller. The host names are defined in /usr/include/netdir.h. The currently defined host names are:

HOST_SELF Represents the address to which local programs will bind their endpoints. HOST_SELF differs from the host name provided by gethostname(3), which represents the address to which *remote* programs will bind their endpoints.

HOST_ANY Represents any host accessible by this transport provider. HOST_ANY allows applications to specify a required service without specifying a particular host name.

HOST_BROADCAST
Represents the address for all hosts accessible by this transport provider. Network requests to this address will be received by all machines.

All fields of the nd_hostserv structure must be initialized.

To find all available transports, call the netdir_getbyname( ) routine with each struct netconfig structure returned by the getnetpath(3N) call.

The netdir_getbyaddr( ) routine maps addresses to service names. This routine returns a list of host and service pairs that would yield this address. If more than one tuple of host and service name is returned then the first tuple contains the preferred host and service names.

```
struct nd_hostservlist
      int   *h_cnt;
      struct hostserv   *h_hostservs;
```

The netdir_free( ) structure is used to free the structures allocated by the name to address translation routines.

The netdir_mergeaddr( ) routine is used by a network service to return an optimized network addresses to a client. This routine takes the universal address of the endpoint that the service has bound to, which is pointed to by the *s_uaddr* parameter, and the address of the endpoint that a request came in on, which is pointed to by the *c_uaddr* paramter, to create an optimized address for communication with the service. The service address should be an address returned by the netdir_getbyname( ) call, specified with the special host name HOST_SELF.

The taddr2uaddr( ) and uaddr2taddr( ) routines support translation between universal addresses and TLI type netbufs. The take and return character string pointers. The taddr2uaddr( ) routine returns a pointer to a string that contains the universal address and returns NULL if the conversion is not possible. This is not a fatal condition as some transports may not suppose a universal address form.

option, *fd*, and *pointer_to_args* are passed to the netdir_options routine for the transport specified in netconfigp. There are four values for *option*:

ND_SET_BROADCAST
ND_SET_RESERVEDPORT
ND_CHECK_RESERVEDPORT
ND_MERGEADDR

If a transport provider does not support an option, netdir_options returns -1 and sets _nderror to ND_NOCTRL.

The specific actions of each option follow.

ND_SET_BROADCAST Sets the transport provider up to allow broadcast, if the transport supports broadcast. *fd* is a file descriptor into the transport (i.e., the result of a t_open of /dev/udp). *pointer_to_args* is not used. If this completes, broadcast operations may be performed on file descriptor *fd*.

ND_SET_RESERVEDPORT
Allows the application to bind to a reserved port, if that concept exists for the transport provider. *fd* is a file descriptor into the transport (it must not be bound to an address). If *pointer_to_args* is NULL, *fd* will be bound to a reserved port. If *pointer_to_args* is a pointer to a netbuf structure, an attempt will be made to bind to a reserved port on the specified address.

ND_CHECK_RESERVEDPORT
Used to verify that an address corresponds to a reserved port, if that concept exists for the transport provider. *fd* is not used. *pointer_to_args* is a pointer to a netbuf structure that contains an address. This option returns 0 only if the address specified in *pointer_to_args* is reserved.

ND_MERGEADDR Used to take a "local address" (like the 0.0.0.0 address that TCP uses) and return a "real address" that client machines can connect to. *fd* is not used. *pointer_to_args* is a pointer to a struct nd_mergearg, which has the following form:

```
struct nd_mergearg {
      char *s_uaddr;  /* server's universal address */
      char *c_uaddr;  /* client's universal address */
      char *m_uaddr;  /* the result */
}
```

s_uaddr is something like 0.0.0.0.1.12, and, if the call is successful, m_uaddr will be set to something like 192.11.109.89.1.12. For most transports, m_uaddr is exactly what s_uaddr is.

The netdir_perror() routine prints an error message on the standard output stating why one of the name-to-address mapping routines failed. The error message is preceded by the string given as an argument.

The netdir_sperror() routine returns a string containing an error message stating why one of the name-to-address mapping routines failed.

SEE ALSO

getnetpath(3N).

NAME

nl_langinfo – language information

SYNOPSIS

```
#include <nl_types.h>
#include <langinfo.h>

char *nl_langinfo (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 programs locale. The manifest constant names and values of *item* are defined by langinfo.h.

For example:

```
nl_langinfo (ABDAY_1);
```

would return a pointer to the string "Dim" if the identified language was French and a French locale was correctly installed; or "Sun" if the identified language was English.

SEE ALSO

gettxt(3C), localeconv(3C), setlocale(3C), strftime(3C), langinfo(5), nl_types(5).

DIAGNOSTICS

If setlocale has not been called successfully, or if langinfo data for a supported language is either not available or *item* is not defined therein, then 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.

WARNING

The array pointed to by the return value should not be modified by the program. Subsequent calls to nl_langinfo may overwrite the array.

The nl_langinfo function is built upon the functions localeconv, strftime, and gettxt [see langinfo(5)]. Where possible users are advised to use these interfaces to the required data instead of using calls to nl_langinfo.

NAME

nlist – get entries from name list

SYNOPSIS

```
#include <nlist.h>

int nlist (file-name, nl)
char *file-name;
struct nlist *nl;
```

DESCRIPTION

nlist examines the name list in the COFF executable file whose name is pointed to by *filename*, and selectively extracts a list of values and puts them in the array of nlist structures pointed to by *nl*. The name list *nl* consists of an array of structures containing names of variables, types, and values. The list is terminated with a null name, that is, a null string is in the name position of the structure.

Each variable name is looked up in the name list of the file. If the name is found, the type, value, storage class, and section number of the name are inserted in the other fields. The type field may be set to 0 if the file was not compiled with the -g option. If the name is not found, all fields in the structure except n_name are set to 0. See a.out(4) for a discussion of the symbol table structure.

This function is useful for examining the system name list kept in the file /dgux. In this way programs can obtain system addresses that are up to date.

NOTES

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

DIAGNOSTICS

All value entries are set to 0 if the file cannot be read or if it does not contain a valid name list.

nlist returns 0 on success, -1 on error.

SEE ALSO

a.out(4).

NAME

nlsgetcall – get client's data passed via the listener

SYNOPSIS

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

struct t_call *nlsgetcall (int fd);
```

DESCRIPTION

nlsgetcall allows server processes started by the listener process to access the client's t_call structure, that is, the *sndcall* argument of t_connect(3N).

The t_call structure returned by nlsgetcall can be released using t_free(3N).

nlsgetcall returns the address of an allocated t_call structure or NULL if a t_call structure cannot be allocated. If the t_alloc succeeds, undefined environment variables are indicated by a negative *len* field in the appropriate netbuf structure. A *len* field of zero in the netbuf structure is valid and means that the original buffer in the listener's t_call structure was NULL.

WARNING

The *len* field in the netbuf structure is defined as being unsigned. In order to check for error returns, it should first be cast to an int.

The listener process limits the amount of user data (*udata*) and options data (*opt*) to 128 bytes each. Address data *addr* is limited to 64 bytes. If the original data was longer, no indication of overflow is given.

DIAGNOSTICS

A NULL pointer is returned if a t_call structure cannot be allocated by t_alloc. t_errno can be inspected for further error information. Undefined environment variables are indicated by a negative length field (*len*) in the appropriate netbuf structure.

FILES

/usr/lib/libnsl_s.a
/usr/lib/libslan.a
/usr/lib/libnls.a

SEE ALSO

nlsadmin(1), getenv(3), t_connect(3N), t_alloc(3N), t_free(3N), t_error(3N).

NOTES

Server processes must call t_sync(3N) before calling this routine.

NAME

nlsprovider – get name of transport provider

SYNOPSIS

```
char *nlsprovider();
```

DESCRIPTION

nlsprovider returns a pointer to a null terminated character string which contains the name of the transport provider as placed in the environment by the listener process. If the variable is not defined in the environment, a NULL pointer is returned.

The environment variable is only available to server processes started by the listener process.

SEE ALSO

nlsadmin(1M).

DIAGNOSTICS

If the variable is not defined in the environment, a NULL pointer is returned.

FILES

```
/usr/lib/libslan.a (7300)
/usr/lib/libnls.a (3B2 Computer)
/usr/lib/libnsl_s.a
```

NAME

nlsrequest – format and send listener service request message

SYNOPSIS

```
#include <listen.h>

int nlsrequest (int fd, char *service_code);

extern int _nlslog, t_errno;
extern char *_nlsrmsg;
```

DESCRIPTION

Given a virtual circuit to a listener process (*fd*) and a service code of a server process, nlsrequest formats and sends a *service request message* to the remote listener process requesting that it start the given service. nlsrequest waits for the remote listener process to return a *service request response message*, which is made available to the caller in the static, null terminated data buffer pointed to by _nlsrmsg. The *service request response message* includes a success or failure code and a text message. The entire message is printable.

SEE ALSO

nlsadmin(1), t_error(3).

FILES

/usr/lib/libnls.a
/usr/lib/libslan.a
/usr/lib/libnsl_s.a

DIAGNOSTICS

The success or failure code is the integer return code from nlsrequest. Zero indicates success, other negative values indicate nlsrequest failures as follows:

-1: Error encountered by nlsrequest, see t_errno.

Postive values are error return codes from the *listener* process. Mnemonics for these codes are defined in <listen.h>.

2: Request message not interpretable.
3: Request service code unknown.
4: Service code known, but currently disabled.

If non-null, _nlsrmsg contains a pointer to a static, null terminated character buffer containing the *service request response message*. Note that both _nlsrmsg and the data buffer are overwritten by each call to nlsrequest.

If _nlslog is non-zero, nlsrequest prints error messages on stderr. Initially, _nlslog is zero.

WARNING

nlsrequest cannot always be certain that the remote server process has been successfully started. In this case, nlsrequest returns with no indication of an error and the caller will receive notification of a disconnect event via a T_LOOK error before or during the first t_snd or t_rcv call.

NAME

p2open, p2close – open, close pipes to and from a command

SYNOPSIS

cc [*flag* ...] *file* ... -lgen [*library* ...]

```
#include <libgen.h>

int p2open (const char *cmd, FILE *fp[2]);

int p2close (FILE *fp[2]);
```

DESCRIPTION

p2open forks and execs a shell running the command line pointed to by *cmd*. On return, fp[0] points to a FILE pointer to write the command's standard input and fp[1] points to a FILE pointer to read from the command's standard output. In this way the program has control over the input and output of the command.

The function returns 0 if successful; otherwise it returns –1.

p2close is used to close the file pointers that p2open opened. It waits for the process to terminate and returns the process status. It returns 0 if successful; otherwise it returns –1.

EXAMPLES

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

main(argc,argv)
int argc;
char **argv;
{
        FILE *fp[2];
        pid_t pid;
        char buf[16];

        pid=p2open("/usr/bin/cat", fp);
        if ( pid == 0 ) {
              fprintf(stderr, "p2open failed\n");
              exit(1);
        }
        write(fileno(fp[0]),"This is a test\n", 16);
        if(read(fileno(fp[1]), buf, 16) <=0)
              fprintf(stderr, "p2open failed\n");
        else
              write(1, buf, 16);
        (void)p2close(fp);
}
```

SEE ALSO

fclose(3S), popen(3S), setbuf(3S).

DIAGNOSTICS

A common problem is having too few file descriptors. p2close returns –1 if the two file pointers are not from the same p2open.

NOTES

Buffered writes on fp[0] can make it appear that the command is not listening. Judiciously placed fflush calls or unbuffering fp[0] can be a big help; see fclose(3S).

Many commands use buffered output when connected to a pipe. That, too, can make it appear as if things are not working.

Usage is not the same as for popen, although it is closely related.

NAME

panel_above: panel_above, panel_below – panels deck traversal primitives

SYNOPSIS

```
#include <panel.h>

PANEL *panel_above(PANEL *panel);

PANEL *panel_below(PANEL *panel);
```

DESCRIPTION

panel_above returns a pointer to the panel just above *panel*, or NULL if *panel* is the top panel. panel_below returns a pointer to the panel just below *panel*, or NULL if *panel* is the bottom panel.

If NULL is passed for *panel*, panel_above returns a pointer to the bottom panel in the deck, and panel_below returns a pointer to the top panel in the deck.

RETURN VALUE

NULL is returned if an error occurs.

NOTES

These routines allow traversal of the deck of currently visible panels.

The header file <panel.h> automatically includes the header file <curses.h>.

SEE ALSO

curses(3X), panels(3X).

NAME

panel_move: move_panel – move a panels window on the virtual screen

SYNOPSIS

```
#include <panel.h>

int move_panel(PANEL *panel, int starty, int startx);
```

DESCRIPTION

move_panel moves the curses window associated with *panel* so that its upper left-hand corner is at *starty*, *startx*. See usage note, below.

RETURN VALUE

OK is returned if the routine completes successfully, otherwise ERR is returned.

NOTES

For panels windows, use move_panel instead of the mvwin curses routine. Otherwise, update_panels will not properly update the virtual screen.

The header file <panel.h> automatically includes the header file <curses.h>.

SEE ALSO

curses(3X), panels(3X), panel_update(3X).

NAME

panel_new: new_panel, del_panel – create and destroy panels

SYNOPSIS

```
#include <panel.h>

PANEL *new_panel(WINDOW *win);

int del_panel(PANEL *panel);
```

DESCRIPTION

new_panel creates a new panel associated with *win* and returns the panel pointer. The new panel is placed on top of the panel deck.

del_panel destroys *panel*, but not its associated window.

RETURN VALUE

new_panel returns NULL if an error occurs.

del_win returns OK if successful, ERR otherwise.

NOTES

The header file <panel.h> automatically includes the header file <curses.h>.

SEE ALSO

curses(3X), panels(3X), panel_update(3X).

NAME

panel_show: show_panel, hide_panel, panel_hidden – panels deck manipulation routines

SYNOPSIS

```
#include <panel.h>

int show_panel(PANEL *panel);

int hide_panel(PANEL *panel);

int panel_hidden(PANEL *panel);
```

DESCRIPTION

show_panel makes *panel*, previously hidden, visible and places it on top of the deck of panels.

hide_panel removes *panel* from the panel deck and, thus, hides it from view. The internal data structure of the panel is retained.

panel_hidden returns TRUE (1) or FALSE (0) indicating whether or not *panel* is in the deck of panels.

RETURN VALUE

show_panel and hide_panel return the integer OK upon successful completion or ERR upon error.

NOTES

The header file <panel.h> automatically includes the header file <curses.h>.

SEE ALSO

curses(3X), panels(3X), panel_update(3X).

NAME

panel_top: top_panel, bottom_panel - panels deck manipulation routines

SYNOPSIS

```
#include <panel.h>

int top_panel(PANEL *panel);

int bottom_panel(PANEL *panel);
```

DESCRIPTION

top_panel pulls *panel* to the top of the desk of panels. It leaves the size, location, and contents of its associated window unchanged.

bottom_panel puts *panel* at the bottom of the deck of panels. It leaves the size, location, and contents of its associated window unchanged.

RETURN VALUE

All of these routines return the integer OK upon successful completion or ERR upon error.

NOTES

The header file <panel.h> automatically includes the header file <curses.h>.

SEE ALSO

curses(3X), panels(3X), panel_update(3X).

NAME

panel_update: update_panels – panels virtual screen refresh routine

SYNOPSIS

```
#include <panel.h>

void update_panels(void);
```

DESCRIPTION

update_panels refreshes the virtual screen to reflect the depth relationships between the panels in the deck. The user must use the curses library call doupdate [see curs_refresh(3X)] to refresh the physical screen.

NOTES

The header file <panel.h> automatically includes the header file <curses.h>.

SEE ALSO

curses(3X), panels(3X), curs_refresh(3X).

NAME

panel_userptr: set_panel_userptr, panel_userptr – associate application data with a panels panel

SYNOPSIS

```
#include <panel.h>

int set_panel_userptr(PANEL *panel, char *ptr);

char * panel_userptr(PANEL *panel);
```

DESCRIPTION

Each panel has a user pointer available for maintaining relevant information.

set_panel_userptr sets the user pointer of *panel* to *ptr*.

panel_userptr returns the user pointer of *panel*.

RETURN VALUE

set_panel_userptr returns OK if successful, ERR otherwise.

panel_userptr returns NULL if there is no user pointer assigned to *panel*.

NOTES

The header file <panel.h> automatically includes the header file <curses.h>.

SEE ALSO

curses(3X), panels(3X).

NAME

panel_window: panel_window, replace_panel – get or set the current window of a panels panel

SYNOPSIS

```
#include <panel.h>

WINDOW *panel_window(PANEL *panel);

int replace_panel(PANEL *panel, WINDOW *win);
```

DESCRIPTION

panel_window returns a pointer to the window of *panel*.

replace_panel replaces the current window of *panel* with *win*.

RETURN VALUE

panel_window returns NULL on failure.

replace_panel returns OK on successful completion, ERR otherwise.

NOTES

The header file <panel.h> automatically includes the header file <curses.h>.

SEE ALSO

curses(3X), panels(3X).

NAME

panels – character based panels package

SYNOPSIS

#include <panel.h>

DESCRIPTION

The panel library is built using the curses library, and any program using panels routines must call one of the curses initialization routines such as initscr. A program using these routines must be compiled with -lpanel and -lcurses on the cc command line.

The panels package gives the applications programmer a way to have depth relationships between curses windows; a curses window is associated with every panel. The panels routines allow curses windows to overlap without making visible the overlapped portions of underlying windows. The initial curses window, stdscr, lies beneath all panels. The set of currently visible panels is the *deck* of panels.

The panels package allows the applications programmer to create panels, fetch and set their associated windows, shuffle panels in the deck, and manipulate panels in other ways.

Routine Name Index

The following table lists each panels routine and the name of the manual page on which it is described.

panels **Routine Name**	**Manual Page Name**
bottom_panel	panel_top(3X)
del_panel	panel_new(3X)
hide_panel	panel_show(3X)
move_panel	panel_move(3X)
new_panel	panel_new(3X)
panel_above	panel_above(3X)
panel_below	panel_above(3X)
panel_hidden	panel_show(3X)
panel_userptr	panel_userptr(3X)
panel_window	panel_window(3X)
replace_panel	panel_window(3X)
set_panel_userptr	panel_userptr(3X)
show_panel	panel_show(3X)
top_panel	panel_top(3X)
update_panels	panel_update(3X)

RETURN VALUE

Each panels routine that returns a pointer to an object returns NULL if an error occurs. Each panel routine that returns an integer, returns OK if it executes successfully and ERR if it does not.

NOTES

The header file <panel.h> automatically includes the header file <curses.h>.

SEE ALSO

curses(3X), and 3X pages whose names begin "panel_," for detailed routine descriptions.

NAME

pathfind – search for named file in named directories

SYNOPSIS

```
cc [flag ...] file ...  -lgen [library ...]

#include <libgen.h>

char *pathfind (const char *path, const char *name, const char
    *mode);
```

DESCRIPTION

pathfind searches the directories named in *path* for the file *name*. The directories named in *path* are separated by semicolons. *mode* is a string of option letters chosen from the set rwxfbcdpugks:

Letter	Meaning
r	readable
w	writable
x	executable
f	normal file
b	block special
c	character special
d	directory
p	FIFO (pipe)
u	set user ID bit
g	set group ID bit
k	sticky bit
s	size nonzero

Options read, write, and execute are checked relative to the real (not the effective) user ID and group ID of the current process.

If the file *name*, with all the characteristics specified by *mode*, is found in any of the directories specified by *path*, then pathfind returns a pointer to a string containing the member of *path*, followed by a slash character (/), followed by *name*.

If *name* begins with a slash, it is treated as an absolute path name, and *path* is ignored.

An empty *path* member is treated as the current directory. rather, the unadorned *name* is returned.

EXAMPLES

To find the ls command using the PATH environment variable:

```
pathfind (getenv ("PATH"), "ls", "rx")
```

SEE ALSO

access(2), mknod(2), stat(2), getenv(3C).
sh(1), test(1) in the *User's Reference Manual*.

DIAGNOSTICS

If no match is found, pathname returns a null pointer, ((char *) 0).

NOTES

The string pointed to by the returned pointer is stored in a static area that is reused on subsequent calls to pathfind.

NAME

perror – print system error messages

SYNOPSIS

```
#include <stdio.h>

void perror (const char *s);
```

DESCRIPTION

perror produces a message on the standard error output (file descriptor 2), describing the last error encountered during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, then the message and a newline. (However, if *s* is a null pointer or points to a null string, the colon is not printed.) To be of most use, the argument string should include the name of the program that incurred the error. The error number is taken from the external variable errno, which is set when errors occur but not cleared when non-erroneous calls are made.

FILES

/usr/lib/locale/*locale*/LC_MESSAGES/uxsyserr — message catalog.

SEE ALSO

intro(2), fmtmsg(3C), strerror(3C).

NAME

popen, pclose – initiate pipe to/from a process

SYNOPSIS

```
#include <stdio.h>

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

int pclose (FILE *stream);
```

DESCRIPTION

popen creates a pipe between the calling program and the command to be executed. The arguments to popen are pointers to null-terminated strings. *command* consists of a shell command line. *type* is an I/O mode, either r for reading or w for writing. The value returned is a stream pointer such that one can write to the standard input of the command, if the I/O mode is w, by writing to the file *stream* [see intro(3)]; and one can read from the standard output of the command, if the I/O mode is r, by reading from the file *stream*.

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

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

EXAMPLE

Here is an example of a typical call:

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

main()
{
        char *cmd = "/usr/bin/ls *.c";
        char buf[BUFSIZ];
        FILE *ptr;

        if ((ptr = popen(cmd, "r")) != NULL)
                while (fgets(buf, BUFSIZ, ptr) != NULL)
                        (void) printf("%s", buf);
        return 0;
}
```

This program will print on the standard output [see stdio(3S)] all the file names in the current directory that have a .c suffix.

SEE ALSO

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

DIAGNOSTICS

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

pclose returns –1 if *stream* is not associated with a popened command or waitpid (used in the implementation of pclose) returns a –1 for some reason (see NOTES below).

NOTES

If the original and popened processes concurrently read or write a common file, neither should use buffered I/O. Problems with an output filter may be forestalled by careful buffer flushing, e.g., with fflush [see fclose(3S)].

If SIGCHLD is set to SIG_IGN, pclose will return a −1 with errno set to ECHILD (the results of calling waitpid).

A security hole exists through the IFS and PATH environment variables. Full pathnames should be used (or PATH reset) and IFS should be set to space and tab (" \t").

NAME

printf, fprintf, sprintf – print formatted output

SYNOPSIS

```
#include <stdio.h>

int printf(const char *format, .../* args */);

int fprintf(FILE *strm, const char *format, .../* args */);

int sprintf(char *s, const char *format, .../* args */);
```

DESCRIPTION

printf places output on the standard output stream stdout.

fprintf places output on *strm*.

sprintf places output, followed by the null character (\0), in consecutive bytes starting at *s*. It is the user's responsibility to ensure that enough storage is available. Each function returns the number of characters transmitted (not including the \0 in the case of sprintf) or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its *args* under control of the *format*. The *format* is a character string that contains three types of objects defined below:

1. plain characters that are simply copied to the output stream;
2. escape sequences that represent non-graphic characters;
3. conversion specifications.

The following escape sequences produce the associated action on display devices capable of the action:

\a Alert. Ring the bell.

\b Backspace. Move the printing position to one character before the current position, unless the current position is the start of a line.

\f Form feed. Move the printing position to the initial printing position of the next logical page.

\n Newline. Move the printing position to the start of the next line.

\r Carriage return. Move the printing position to the start of the current line.

\t Horizontal tab. Move the printing position to the next implementation-defined horizontal tab position on the current line.

\v Vertical tab. Move the printing position to the start of the next implementation-defined vertical tab position.

All forms of the printf functions allow for the insertion of a language-dependent decimal-point character. The decimal-point character is defined by the program's locale (category LC_NUMERIC). In the C locale, or in a locale where the decimal-point character is not defined, the decimal-point character defaults to a period (.).

Each conversion specification is introduced by the character %. After the character %, the following appear in sequence:

> An optional field, consisting of a decimal digit string followed by a $, specifying the next *args* to be converted. If this field is not provided, the *args* following the last *args* converted will be used.

Zero or more *flags*, which modify the meaning of the conversion specification.

An optional string of decimal digits to specify 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.

An optional precision that gives the minimum number of digits to appear for the d, i, o, u, x, or X conversions (the field is padded with leading zeros), the number of digits to appear after the decimal-point character for the e, E, and f conversions, the maximum number of significant digits for the g and G conversions, 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 h specifies that a following d, i, o, u, x, or X conversion specifier applies to a short int or unsigned short int argument (the argument will be promoted according to the integral promotions and its value converted to short int or unsigned short int before printing); an optional h specifies that a following n conversion specifier applies to a pointer to a short int argument. An optional l (ell) specifies that a following d, i, o, u, x, or X conversion specifier applies to a long int or unsigned long int argument; an optional l (ell) specifies that a following n conversion specifier applies to a pointer to long int argument. An optional L specifies that a following e, E, f, g, or G conversion specifier applies to a long double argument. If an h, l, or L appears before any other conversion specifier, the behavior is undefined.

A conversion character (see below) that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk (*) instead of a digit string. In this case, an integer *args* supplies the field width or precision. The *args* that is actually converted is not fetched until the conversion letter is seen, so the *args* specifying field width or precision must appear before the *args* (if any) to be converted. If the *precision* argument is negative, it will be changed to zero. A negative field width argument is taken as a – flag, followed by a positive field width.

In format strings containing the **digits*$ form of a conversion specification, a field width or precision may also be indicated by the sequence **digits*$, giving the position in the argument list of an integer *args* containing the field width or precision.

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.

The *flag* characters and their meanings are:

– The result of the conversion will be left-justified within the field. (It will be right-justified if this flag is not specified.)

+ The result of a signed conversion will always begin with a sign (+ or –). (It will begin with a sign only when a negative value is converted if this flag is not specified.)

space If the first character of a signed conversion is not a sign, a space will be placed before the result. This means that if the space and + flags both appear, the space flag will be ignored.

# The value is to be converted to an alternate form. For c, d, i, s, and u conversions, the flag has no effect. For an 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) prepended to it. For e, E, f, g, and G conversions, the result will always contain a decimal-point character, 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 zeros will not be removed from the result as they normally are.

0 For d, i, o, u, x, X, e, E, f, g, and G conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the 0 and flags both appear, the 0 flag will be ignored. For d, i, o, u, x, and X conversions, if a precision is specified, the 0 flag will be ignored. For other conversions, the behavior is undefined.

Each conversion character results in fetching 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 ignored.

The conversion characters and their meanings are:

d,i,o,u,x,X The integer *arg* is converted to signed decimal (d or i), (unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal notation (x and X). The x conversion uses the letters abcdef and the X conversion uses the letters ABCDEF. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits than the specified minimum, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

f The double *args* is converted to decimal notation in the style [-]*ddd*.*ddd*, where the number of digits after the decimal-point character [see setlocale(3C)] is equal to the precision specification. If the precision is omitted from *arg*, six digits are output; if the precision is explicitly zero and the # flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least 1 digit appears before it. The value is rounded to the appropriate number of digits.

e,E The double *args* is converted to the style [-]*d*.*ddd*e±*dd*, where there is one digit before the decimal-point character (which is non-zero if the argument is non-zero) and the number of digits after it is equal to the precision. When the precision is missing, six digits are produced; if the precision is zero and the # flag is not specified, no decimal-point character appears. The E conversion character will produce a number with E instead of e introducing the exponent. The exponent always contains at least two digits. The value is rounded to the appropriate number of digits.

g,G
: The double *args* is printed in style f or e (or in style E in the case of a G conversion character), with the precision specifying the number of significant digits. If the precision is zero, it is taken as one. 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 or equal to the precision. Trailing zeros are removed from the fractional part of the result. A decimal-point character appears only if it is followed by a digit.

c
: The int *args* is converted to an unsigned char, and the resulting character is printed.

s
: The *args* is taken to be a string (character pointer) and characters from the string are written up to (but not including) a terminating null character; if the precision is specified, no more than that many characters are written. If the precision is not specified, it is taken to be infinite, so all characters up to the first null character are printed. A NULL value for *args* will yield undefined results.

p
: The *args* should be a pointer to void. The value of the pointer is converted to an implementation-defined set of sequences of printable characters, which should be the same as the set of sequences that are matched by the %p conversion of the scanf function.

n
: The argument should be a pointer to an integer into which is written the number of characters written to the output standard I/O stream so far by this call to printf, fprintf, or sprintf. No argument is converted.

%
: Print a %; no argument is converted.

If the character after the % or %*digits*$ sequence is not a valid conversion character, the results of the conversion are undefined.

If a floating-point value is the internal representation for infinity, the output is [±]*inf*, where *inf* is either inf or INF, depending on the conversion character. Printing of the sign follows the rules described above.

If a floating-point value is the internal representation for "not-a-number," the output is [±]*nan*0*xm*. Depending on the conversion character, *nan* is either nan or NAN. Additionally, 0*xm* represents the most significant part of the mantissa. Again depending on the conversion character, *x* will be x or X, and *m* will use the letters abcdef or ABCDEF. Printing of the sign follows the rules described above.

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

EXAMPLE

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

```
printf("%s, %s %i, %d:%.2d",
        weekday, month, day, hour, min);
```

To print π to 5 decimal places:

```
        printf("pi = %.5f", 4 * atan(1.0));
```

SEE ALSO

exit(2), lseek(2), write(2), abort(3C), ecvt(3C), putc(3S), scanf(3S), setlocale(3C), stdio(3S).

DIAGNOSTICS

printf, fprintf, and sprintf return the number of characters transmitted, or return a negative value if an error was encountered.

NAME

printf, fprintf, sprintf – print formatted output

SYNOPSIS

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

int printf (const char *format [, arg] ... );

int fprintf (FILE *stream, const char *format [, arg] ... );

int sprintf (char *s, const char *format [, arg] ... );
```

DESCRIPTION (International Functions)

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 in a character array pointed to by *s*. Each function returns the number of bytes transmitted (not including the NULL character in the case of *sprintf*), or a negative value if an output error was encountered.

Each of these functions converts, formats and prints its *args* under control of the *format*. The *format* is a character string that contains two types of object: plain characters, including ASCII characters and characters in supplementary code sets which are simply copied to the output stream, and conversion specifications which can contain only ASCII characters, each of which results in the fetching of zero or more *args*.

wc and ws are the new conversion specifications for wchar_t character control. Both *wc* and *ws* may be used in all three functions.

wc The wchar_t character *arg* is transformed into EUC, and then printed. If a field width is specified and the transformed EUC has fewer bytes than the field width, it will by padded to the given width. A precision specification is ignored, if specified.

ws The *arg* is taken to be a wchar_t string and the wchar_t characters from the string are transformed into EUC, and printed until a wchar_t null character is encountered or the number of bytes indicated by the precision specification is printed. If the precision specification is missing, it is taken to be infinite, and all wchar_t characters up to the first wchar_t null character are transformed into EUC and printed. If a field width is specified and the transformed EUC have fewer bytes than the field width, they are padded to the given width.

The ASCII space character (0x20) is used as a padding characters.

DIAGNOSTICS

printf, fprintf, and sprintf returns the number of bytes transmitted, or return a negative value if an error was encountered.

SEE ALSO

printf(3S), scanf(3W), stdio(3S), vprintf(3W), widec(3W).

NAME

psignal, psiginfo – system signal messages

SYNOPSIS

```
#include <siginfo.h>

void psignal (int sig, const char *s);

void psiginfo (siginfo_t *pinfo, char *s);
```

DESCRIPTION

psignal and psiginfo produce messages on the standard error output describing a signal. *sig* is a signal that may have been passed as the first argument to a signal handler. *pinfo* is a pointer to a siginfo structure that may have been passed as the second argument to an enhanced signal handler [see sigaction(2)]. The argument string *s* is printed first, then a colon and a blank, then the message and a newline.

SEE ALSO

sigaction(2), perror(3C), siginfo(5), signal(5).

NAME

`ptsname` – get name of the slave pseudo-terminal device

SYNOPSIS

```
#include <stdio.h>

char *ptsname(int fildes);
```

DESCRIPTION

The function `ptsname()` returns the name of the slave pseudo-terminal device associated with a master pseudo-terminal device. *fildes* is a file descriptor returned from a successful open of the master device. `ptsname()` returns a pointer to a string containing the null-terminated path name of the slave device of the form `/dev/pts/N`, where `N` is an integer between 0 and 255.

RETURN VALUE

Upon successful completion, the function `ptsname()` returns a pointer to a string which is the name of the pseudo-terminal slave device. This value points to a static data area that is overwritten by each call to `ptsname()`. Upon failure, `ptsname()` returns `NULL`. This could occur if *fildes* is an invalid file descriptor or if the slave device name does not exist in the file system.

SEE ALSO

`open`(2), `grantpt`(3C), `ttyname`(3C), `unlockpt`(3C).
Programmer's Guide: STREAMS.

NAME

putc, putchar, fputc, putw – put character or word on a stream

SYNOPSIS

```
#include <stdio.h>

int putc (int c, FILE *stream);

int putchar (int c);

int fputc (int c, FILE *stream);

int putw (int w, FILEETI *stream);
```

DESCRIPTION

putc writes *c* (converted to an unsigned char) onto the output *stream* [see intro(3)] at the position where the file pointer (if defined) is pointing, and advances the file pointer appropriately. If the file cannot support positioning requests, or *stream* was opened with append mode, the character is appended to the output *stream*. putchar(c) is defined as putc(c, stdout). putc and putchar are macros.

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

putw writes the word (i.e., integer) *w* to the output *stream* (where the file pointer, if defined, is pointing). The size of a word is the size of an integer and varies from machine to machine. putw neither assumes nor causes special alignment in the file.

SEE ALSO

exit(2), lseek(2), write(2), abort(3C), fclose(3S), ferror(3S), fopen(3S), fread(3S), printf(3S), puts(3S), setbuf(3S), stdio(3S).

DIAGNOSTICS

On success, these functions (with the exception of putw) each return the value they have written. putw returns ferror (*stream*). On failure, they return the constant EOF. This result will occur, for example, if the file *stream* is not open for writing or if the output file cannot grow.

NOTES

Because it is implemented as a macro, putc evaluates a *stream* argument more than once. In particular, putc(c, *f++); doesn't work sensibly. fputc should be used instead.

Because of possible differences in word length and byte ordering, files written using putw are machine-dependent, and may not be read using getw on a different processor.

Functions exist for all the above defined macros. To get the function form, the macro name must be undefined (e.g., #undef putc).

NAME

putenv – change or add value to environment

SYNOPSIS

```
#include <stdlib.h>

int putenv (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. Because of this limitation, *string* should be declared static if it is declared within a function.

SEE ALSO

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

DIAGNOSTICS

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

NOTES

putenv manipulates the environment pointed to by *environ*, and can be used in conjunction with getenv. However, *envp* (the third argument to *main*) is not changed.

This routine uses malloc(3C) to enlarge the environment.

After putenv is called, environmental variables are not in alphabetical order. A potential error is to call the function putenv with a pointer to an automatic variable as the argument and to 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 (const struct passwd *p, FILE *f);
```

DESCRIPTION

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

SEE ALSO

getpwent(3C).

DIAGNOSTICS

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

WARNING

The above routine uses <stdio.h>, which causes it to increase the size of programs that otherwise don't use standard I/O. The size increases more than might be expected.

NAME

puts, fputs – put a string on a stream

SYNOPSIS

```
#include <stdio.h>

int puts (const char *s);

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

DESCRIPTION

puts writes the string pointed to by *s*, followed by a new-line character, to the standard output stream stdout [see intro(3)].

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

Neither function writes the terminating null character.

SEE ALSO

exit(2), lseek(2), write(2), abort(3C), fclose(3S), ferror(3S), fopen(3S), fread(3S), printf(3S), putc(3S), stdio(3S).

DIAGNOSTICS

On success both routines return the number of characters written; otherwise they return EOF. In addition, if the routines try to write to a file that has not been opened for writing, **errno** will be set to EBADF.

NOTES

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

NAME

putspent – write shadow password file entry

SYNOPSIS

```
#include <shadow.h>

int putspent (const struct spwd *p, FILE *fp);
```

DESCRIPTION

The putspent routine is the inverse of getspent. Given a pointer to a spwd structure created by the getspent routine (or the getspnam routine), the putspent routine writes a line on the stream *fp*, which matches the format of /etc/shadow.

If the sp_min, sp_max, sp_lstchg, sp_warn, sp_inact, or sp_expire field of the spwd structure is −1, or if sp_flag is 0, the corresponding /etc/shadow field is cleared.

SEE ALSO

getspent(3C), getpwent(3C), putpwent(3C).

DIAGNOSTICS

The putspent routine returns non-zero if an error was detected during its operation, otherwise zero.

NOTES

This routine is for internal use only, compatibility is not guaranteed.

NAME

putwc, putwchar, fputwc – put wchar_t character on a stream

SYNOPSIS

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

int putwc(wchar_t c, FILE *stream);

int putwchar(wchar_t c);

int fputwc(wchar_t c, FILE *stream);
```

DESCRIPTION (International Functions)

putwc() transforms the wchar_t character *c* into EUC, and writes it onto the output stream (at the position where the file pointer, if defined, is pointing). The putwchar(c) is defined as putwc(c, stdout). putwc() and putwchar() are macros.

fputwc() behaves like putwc(), but is a function rather than a macro.

DIAGNOSTICS

On success, each of these functions return the value they have written. On failure, they return the constant EOF.

SEE ALSO

printf(3W), putws(3W), widec(3W).
fclose(3S), ferror(3S), fopen(3S), fread(3S), printf(3S), setbuf(3S), stdio(3S).

NAME

putws, fputws – put a wchar_t string on a stream

SYNOPSIS

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

int putws(const wchar_t *s);

int fputws(const wchar_t *s, FILE *stream);
```

DESCRIPTION (International Functions)

putws() transforms the wchar_t null-terminated wchar_t string pointed to by *s* into a byte string in EUC, and writes the string followed by a new-line character to *stdout*.

fputws() transforms the wchar_t null-terminated wchar_t string pointed to by *s* into a byte string in EUC, and writes the string to the named output stream.

Neither function writes the terminating wchar_t null character.

DIAGNOSTICS

On success both functions return the number of wchar_t characters transformed and written (not including the new-line character in the case of putws()); Otherwise they return EOF.

NOTES

putws() appends a new-line character while fputws() does not.

SEE ALSO

printf(3W), putwc(3W), widec(3W).
ferror(3S), fopen(3S), fread(3S), printf(3S), stdio(3S).

NAME

qsort − quicker sort

SYNOPSIS

```
#include <stdlib.h>

void qsort (void* base, size_t nel, size_t width), int (*compar)
    (const void *, const void *));
```

DESCRIPTION

qsort is an implementation of the quicker-sort algorithm. It sorts a table of data in place. The contents of the table are sorted in ascending order according to the user-supplied comparison function.

base points to the element at the base of the table. *nel* is the number of elements in the table. *width* specifies the size of each element in bytes. *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 to indicate if the first argument is to be considered less than, equal to, or greater than the second.

The contents of the table are sorted in ascending order according to the user supplied comparison function.

SEE ALSO

bsearch(3C), lsearch(3C), string(3C).
sort(1) in the *User's Reference Manual.*

NOTES

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 relative order in the output of two items that compare as equal is unpredictable.

NAME

`raise` – send signal to program

SYNOPSIS

```
#include <signal.h>

int raise (int sig);
```

DESCRIPTION

`raise` sends the signal *sig* to the executing program.

`raise` returns zero if the operation succeeds. Otherwise, `raise` returns –1 and *errno* is set to indicate the error. `raise` uses `kill` to send the signal to the executing program:

```
kill(getpid(), sig);
```

See `kill`(2) for a detailed list of failure conditions. See `signal`(2) for a list of signals.

SEE ALSO

`getpid`(2), `kill`(2), `signal`(2).

NAME

rand, srand – simple random-number generator

SYNOPSIS

```
#include <stdlib.h>

int rand (void);

void srand (unsigned int seed);
```

DESCRIPTION

rand uses a multiplicative congruential random-number generator with period 2^{32} that returns successive pseudo-random numbers in the range from 0 to RAND_MAX (defined in stdlib.h).

The function srand uses the argument *seed* as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to the function rand. If the function srand is then called with the same *seed* value, the sequence of pseudo-random numbers will be repeated. If the function rand is called before any calls to srand have been made, the same sequence will be generated as when srand is first called with a *seed* value of 1.

NOTES

The spectral properties of rand are limited. drand48(3C) provides a much better, though more elaborate, random-number generator.

SEE ALSO

drand48(3C).

NAME

random, srandom, initstate, setstate – generate random numbers better, or change the generator

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(3C) 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(3C), 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.

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.

SEE ALSO

`rand`(3C).

CAVEAT

About 2/3 the speed of `rand`(3C).

NAME

rcmd, rresvport, ruserok – routines for returning a stream to a remote command

SYNOPSIS

```
rem = rcmd(ahost, inport, locuser, remuser, cmd, fd2p);
char **ahost;
u_short inport;
char *locuser, *remuser, *cmd;
int *fd2p;

s = rresvport(port);
int *port;

ruserok(rhost, superuser, ruser, luser);
char *rhost;
int superuser;
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.

rcmd looks up the host **ahost* using gethostbyname(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 call succeeds, a socket of type SOCK_STREAM is returned to the caller, and given to the remote command as stdin and stdout. If *fd2p* is non-zero, then an auxiliary channel to a control process will be set up, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being DG/UX system signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the stderr (unit 2 of the remote command) will be made the same as the stdout and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

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 sevral other routines. Privileged addresses consist of a port 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 gethostent(3N) routine, two user names and a flag indicating if the local user's name is the super-user. It then checks the files /etc/hosts.equiv and, possibly, .rhosts in the 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 "hosts.equiv" file, or the host and remote user name are found in the ".rhosts" file; otherwise ruserok returns –1. If the *superuser* flag is 1, the checking of the "host.equiv" file is bypassed.

SEE ALSO

rlogin(1C), rsh(1C), rexec(3X), rexecd(8C), rlogind(8C),

NOTE

There is no way to specify options to the socket call that rcmd makes.

NAME

realpath – returns the real file name

SYNOPSIS

```
#include <stdlib.h>
#include <sys/param.h>

char *realpath (char * file_name, char * resolved_name);
```

DESCRIPTION

realpath resolves all links and references to "." and ".." in *file_name* and stores it in *resolved_name.*

It can handle both relative and absolute path names. For absolute path names and the relative names whose resolved name cannot be expressed relatively (e.g., ../../reldir), it returns the *resolved absolute* name. For the other relative path names, it returns the *resolved relative* name.

resolved_name must be big enough (MAXPATHLEN) to contain the fully resolved path name.

SEE ALSO

getcwd(3C).

DIAGNOSTICS

If there is no error, realpath returns a pointer to the *resolved_name.* Otherwise it returns a null pointer and places the name of the offending file in *resolved_name.* The global variable errno is set to indicate the error.

NOTES

realpath operates on null-terminated strings.

One should have execute permission on all the directories in the given and the resolved path.

realpath may fail to return to the current directory if an error occurs.

NAME

regcmp, regex – compile and execute regular expression

SYNOPSIS

```
#include <libgen.h>
```

cc [*flag* ...] *file* ... -lgen [*library* ...]

```
char *regcmp (const char *string1 [, char *string2, ...],
    (char *)0);

char *regex (const char *re, const char *subject
    [, char *ret0, ...]);

extern char *__loc1;
```

DESCRIPTION

Regcmp and Regex implement extended regular expressions, without support for internationalization features. See regexpr(3C) as well.

regcmp compiles a regular expression (consisting of the concatenated arguments) and returns a pointer to the compiled form. malloc(3C) is used to create space for the compiled form. It is the user's responsibility to free unneeded space so allocated. A NULL return from regcmp indicates an incorrect argument.

regcmp(1) has been written to generally preclude the need for this routine at execution time. If regcmp(1) is used, the running of regcmp(1) and regex must occur in the same locale (see setlocale(3C)).

regex executes a compiled pattern against the subject string. Additional arguments are passed to receive values back. regex returns NULL on failure or a pointer to the next unmatched character on success. A global character pointer __loc1 points to where the match began. regcmp and regex were mostly borrowed from the editor, ed(1); however, the syntax and semantics have been changed slightly. The following are the valid symbols and associated meanings.

[ ] * . ^ These symbols retain their meaning in ed(1).

$ Matches the end of the string; \n matches a newline.

– Within brackets the minus means *through*. For example, [a-z] is equivalent to [abcd...xyz]. The – can appear as itself only if used as the first or last character. For example, the character class expression []-] matches the characters] and –.

+ A regular expression followed by + means *one or more times*. For example, [0-9]+ is equivalent to [0-9][0-9]*.

{*m*} {*m*,} {*m*,*u*}

Integer values enclosed in { } indicate the number of times the preceding regular expression is to be applied. The value *m* is the minimum number and *u* is a number, less than 256, which is the maximum. If only *m* is present (i.e., {*m*}), it indicates the exact number of times the regular expression is to be applied. The value {*m*,} is analogous to {*m*,*infinity*}. The plus (+) and star (*) operations are equivalent to {1,} and {0,} respectively.

(...)$*n*

The value of the enclosed regular expression is to be returned. The value will be stored in the (*n*+1)th argument following the subject argument. At most, ten enclosed regular expressions are allowed. regex makes its assignments unconditionally.

(. . .) Parentheses are used for grouping. An operator, e.g., *, +, { }, can work on a single character or a regular expression enclosed in parentheses. For example, (a*(cb+)*)$0.

By necessity, all the above defined symbols are special. They must, therefore, be escaped with a \ (backslash) to be used as themselves.

regcmp and regex do *not* support the following international features in regular expressions that are described in ed(1):

[.*ch*.]	multi-character collation symbol
[=*c*=]	collation-order equivalence class
[:*alpha*:]	character class

Moreover, character ranges such as [*a*–*j*] are interpreted by simply comparing the numeric values of the character bytes, not by using collation ordering information.

EXAMPLES

The following example matches a leading newline in the subject string pointed at by cursor.

```
char *cursor, *newcursor, *ptr;
         . . .
newcursor = regex((ptr = regcmp("^\n", (char *)0)), cursor);
free(ptr);
```

The following example matches through the string Testing3 and returns the address of the character after the last matched character (the "4"). The string Testing3 is copied to the character array ret0.

```
char ret0[9];
char *newcursor, *name;
         . . .
name = regcmp("([A-Za-z][A-za-z0-9]{0,7})$0", (char *)0);
newcursor = regex(name, "012Testing345", ret0);
```

The following example applies a precompiled regular expression in file.i [see regcmp(1)] against *string*.

```
#include "file.i"
char *string, *newcursor;
         . . .
newcursor = regex(name, string);
```

SEE ALSO

regcmp(1), malloc(3C).
ed(1) in the *User's Reference Manual*.

NOTES

The user program may run out of memory if regcmp is called iteratively without freeing the vectors no longer required.

NAME

regcmp, regex – compile and execute regular expression

SYNOPSIS

```
char *regcmp (string1 [, string2, ..., stringn], (char *)0)
char *string1, *string2, ..., *stringn;

char *regex (re, subject[, ret0, ...])
char *re, *subject, *ret0, ...;

extern char *__loc1;
```

DESCRIPTION

regcmp and regex implement extended regular expressions, without support for internationalization features. See regexpr(3C) as well.

regcmp compiles a regular expression and returns a pointer to the compiled form. Malloc(3C) is used to create space for the vector. You must free unneeded space so allocated. A NULL return from regcmp indicates an incorrect argument.

Regcmp(1) has been written to generally preclude the need for this routine at execution time. If regcmp(1) is used, the running of regcmp(1) and regex must occur in the same locale (see setlocale(3C)).

regex executes a compiled pattern against the subject string. Additional arguments are passed to receive values back. regex returns NULL on failure or a pointer to the next unmatched character on success. A global character pointer *__loc1* points to where the match began. regcmp and regex were mostly borrowed from the editor, ed(1); however, the syntax and semantics have been changed slightly. The following are the valid symbols and their associated meanings.

[ ] * . ^ These symbols retain their current meaning.

$ Matches the end of the string; \n matches a new-line. The $ symbol *must* be the *last* character of the *last* string*n* argument given to regcmp, or the $ symbol is taken as a literal '$' character (ie., it is given no special meaning at all).

– Within brackets the minus means *through*. For example, [a-z] is equivalent to [abcd . . .xyz]. The – can appear as itself only if used as the first or last character. For example, the character class expression []-] matches the characters]\f4and\f1-.

+ A regular expression followed by + means *one or more times*. For example, [0-9]+ is equivalent to [0-9][0-9]*.

{m} {m,} {m,u}

Integer values enclosed in { } indicate the number of times the preceding regular expression is to be applied. The value *m* is the minimum number and *u* is a number, less than 256, which is the maximum. If only *m* is present (e.g., {m}), it indicates the exact number of times the regular expression is to be applied. The value {m,} is analogous to {m,infinity}. The plus (+) and star (*) operations are equivalent to {1,} and {0,} respectively.

(. . .)$*n*

The value of the enclosed regular expression is returned. The value will be stored in the *(n+1)*th argument following the subject argument. At most ten enclosed regular expressions are allowed. regex makes its assignments unconditionally.

(. . .) Parentheses are used for grouping. An operator, e.g., *, +, { }, can work on a single character or a regular expression enclosed in parentheses. For example, (a*(cb+)*)$0.

All of these symbols are special. They must, therefore, be escaped to be used as themselves (except in the case of the $ symbol which is explained above).

`regcmp` and `regex` do *not* support the following international features in regular expressions that are described in ed(1):

[.*ch*.]	multi-character collation symbol
[=*c*=]	collation-order equivalence class
[:*alpha*:]	character class

Moreover, character ranges such as [*a*–*j*] are interpreted by simply comparing the numeric values of the character bytes, not by using collation ordering information.

EXAMPLES

Example 1:

```
        char *cursor, *newcursor, *ptr;
                    . . .
        newcursor = regex((ptr = regcmp("^\n", 0)), cursor);
        free(ptr);
```

This example will match a leading new-line in the subject string that the cursor points to.

Example 2:

```
        char ret0[9];
        char *newcursor, *name;
                    . . .
        name = regcmp("([A-Za-z][A-za-z0-9_]{0,7})$0", 0);
        newcursor = regex(name, "123Testing321", ret0);
```

This example matches through the string Testing3 and returns the address of the character after the last matched character (cursor+11). The string Testing3 is copied to the character array *ret0*.

Example 3:

```
        char ret0[9];
        char *newcursor, *name;
                    . . .
        name = regcmp("(a+)$0", "$", (char *) 0);
        newcursor = regex(name, "aabcaaa", ret0);
```

This is an example of how the $ symbol should be used to anchor regular expressions. This example matches through the string 'aaa' and returns the address of the character after the last matched character. The string 'aaa' is copied to the character array *ret0*.

Example 4:

```
        #include "file.i"
        char *string, *newcursor;
                    . . .
        newcursor = regex(name, string);
```

This example applies a precompiled regular expression in `file.i` (see `regcmp`(1)) against *string*.

This routine is kept in `/lib/libPW.a`.

SEE ALSO

`regcmp`(3G), `malloc`(3C).
ed(1), `regcmp`(1) in the *User's Reference for the DG/UX System*

CAUTION

The user program may run out of memory if *regcmp* is called iteratively without freeing the vectors no longer required.

This *regcmp* in `/lib/libPW.a` has been replaced by the one in `/lib/libgen.a`. See `regcmp`(3G).

NAME

regexpr: compile, step, advance – regular expression compile and match routines

SYNOPSIS

```
cc [flag ...] file ...  -lgen [library ...]

#include <regexpr.h>

char *compile (const char *instring, char *expbuf, char *endbuf);

int step (const char *string, char *expbuf);

int advance (const char *string, char *expbuf);

char *regerr (int regerrno);

extern char *loc1, *loc2, *locs;

extern int nbra, regerrno, reglength;

extern char *braslist[], *braelist[];
```

DESCRIPTION

These routines are used to compile regular expressions and match the compiled expressions against lines. The regular expressions supported are "simple" internationalized regular expressions, such as those used in ed. For "extended" regular expressions, see *regcmp*(3G).

The syntax of the compile routine is as follows:

```
compile (instring, expbuf, endbuf)
```

The parameter *instring* is a null-terminated string representing the regular expression.

The parameter *expbuf* points to the place where the compiled regular expression is to be placed. If *expbuf* is NULL, compile uses malloc to allocate the space for the compiled regular expression. If an error occurs, this space is freed. It is the user's responsibility to free unneeded space after the compiled regular expression is no longer needed.

The parameter *endbuf* is one more than the highest address where the compiled regular expression may be placed. This argument is ignored if *expbuf* is NULL. If the compiled expression cannot fit in (*endbuf–expbuf*) bytes, compile returns NULL and regerrno (see below) is set to 50.

If compile succeeds, it returns a non-NULL pointer whose value depends on *expbuf*. If *expbuf* is non-NULL, compile returns a pointer to the byte after the last byte in the compiled regular expression. The length of the compiled regular expression is stored in reglength. Otherwise, compile returns a pointer to the space allocated by malloc.

If an error is detected when compiling the regular expression, a NULL pointer is returned from compile and regerrno is set to one of the non-zero error numbers indicated below:

ERROR	MEANING
11	Range endpoint too large.

16	Bad number.
25	"\digit" out of range.
36	Illegal or missing delimiter.
41	No remembered search string.
42	\(~\) imbalance.
43	Too many \(.
44	More than 2 numbers given in \{ ~\}.
45	} expected after \.
46	First number exceeds second in \{ ~\}.
49	[ ] imbalance.
50	Regular expression overflow.
200	Inside [ ], a [.*cc*.] construct was used to describe a two-character collation symbol which does not exist in the current locale.
202	Unterminated [= =] or [. .] construct within [ ].
203	Illegal use of multibyte character in [ ].
204	Unrecognized [:*xxx*:] class in [ ].
205	Both a multibyte character and a multicharacter collation symbol included in a [ ] construct (the collation symbol may not be explicit).

regerror accepts as input a regerrno value, and returns a pointer to a statically-allocated copy of a description of the error. This pointer is good only until the next call to regerror.

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 parameter *expbuf* is the compiled regular expression 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. loc1 is a pointer to the first character that matched the regular expression. The variable loc2 points to the character after the last character that matches the regular expression. Thus if the regular expression matches the entire line, loc1 points to the first character of *string* and loc2 points to the null at the end of *string*.

The purpose of step is to step through the *string* argument until a match is found or until the end of *string* is reached. If the regular expression begins with ^, step tries to match the regular expression at the beginning of the string only.

The function advance has the same arguments and side effects as step, but it always restricts matches to the beginning of the string.

If one is looking for successive matches in the same string of characters, locs should be set equal to loc2, and step should be called with *string* equal to loc2. locs is used by commands like ed and sed so that global substitutions like s/y*//g do not loop forever, and is NULL by default.

The external variable nbra is used to determine the number of subexpressions in the compiled regular expression. braslist and braelist are arrays of character pointers that point to the start and end of the nbra subexpressions in the matched string. For example, after calling step or advance with string sabcdefg and regular expression \(abcdef\), braslist[0] will point at a and braelist[0] will point at g. These arrays are used by commands like ed and sed for substitute replacement patterns that contain the *n* notation for subexpressions.

Note that it isn't necessary to use the external variables regerrno, nbra, loc1, loc2 locs, braelist, and braslist if one is only checking whether or not a string matches a regular expression.

EXAMPLES

The following is similar to the regular expression code from grep:

```
#include <regexpr.h>

. . .

if(compile(*argv, (char *)0, (char *)0) == (char *)0)
    regerr(regerrno);

. . .

if (step(linebuf, expbuf))
    succeed();
```

SEE ALSO

regexpr(3G).
regexp(5).
ed(1), grep(1), sed(1) in the *User's Reference Manual*.

NAME

remove – remove file

SYNOPSIS

```
#include <stdio.h>

int remove(const char *path);
```

DESCRIPTION

remove causes the file or empty directory whose name is the string pointed to by *path* to be no longer accessible by that name. A subsequent attempt to open that file using that name will fail, unless the file is created anew. If a file is removed while one or more processes have the file open, the removal is postponed until all references to the file are closed.

For files, remove is identical to unlink. For directories, remove is identical to rmdir.

See rmdir(2) and unlink(2) for a detailed list of failure conditions.

SEE ALSO

rmdir(2), unlink(2).

RETURN VALUE

Upon successful completion, remove returns a value of 0; otherwise, it returns a value of -1 and sets errno to indicate an error.

NAME

remque – remove an element from a circular queue

SYNOPSIS

```
struct qelem {
struct qelem *q_forw;
struct qelem *q_back;
/* User data follows */
} *elem;

int remque();
. . .
remque(elem);
```

elem A pointer to the structure to remove from a linked list

DESCRIPTION

The remque function removes an element from a circular linked list; this function comes from the University of California Berkeley UNIX (BSD) system.

The structures in the linked list must reserve the first two double words for use as the forward and backward pointers. Since the linked list is a circular list, the initial element must have forward and backward links to itself.

RETURNS

The remque function does not return a value.

SEE ALSO

insque(3C).

EXAMPLE

This program reads lines from standard input, creates a linked list, removes elements from the list with remque, and prints the elements.

```
/* Program test for the remque() function */

#include <stdio.h>

#define LSIZE 256       /* line size */

extern char  *malloc();
extern int   insque(), remque();
extern void  free();

struct qelem {
    struct qelem *q_forw;       /* forward link */
    struct qelem *q_back;       /* backward link */
    char buffer[LSIZE];         /* line buffer */
};

struct qelem head = {&head, &head}; /* head of queue */
struct qelem *p_last = &head;   /* last item */
struct qelem *p_line;           /* ptr to walk list */
struct qelem *p_next;           /* next item in list */

main() {
    for (;;) {
        p_line = (struct qelem *)
                malloc(sizeof(struct qelem));
        if (p_line == (struct qelem *)0) {
            printf("Out of memory.\n");
            exit(1);
        }
        if (!fgets(p_line -> buffer, LSIZE, stdin))
            break;      /* End of file found */
        (void) insque(p_line, p_last);
        p_last = p_line;
    }
    free((char *)p_line);

    /* Now walk list and print elements */

    for (p_line = head.q_forw; p_line != &head;
         p_line = p_next) {
        p_next = p_line -> q_forw;
        (void) remque(p_line);
        fputs(p_line -> buffer, stdout);
        free((char *)p_line);
    }
    return 0;
}
```

NAME

resolver: res_mkquery, res_send, res_init, dn_comp, dn_expand – make, send, and interpret packets to Internet domain name servers

SYNOPSIS

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

res_mkquery(op, dname, class, type, data, datalen, newrr, buf,
buflen)
int op;
char *dname;
int class, type;
char *data;
int datalen;
struct rrec *newrr;
char *buf;
int buflen;

res_send(msg, msglen, answer, anslen)
char *msg;
int msglen;
char *answer;
int anslen;

res_init()

dn_comp(exp_dn, comp_dn, length, dnptrs, lastdnptr)
char *exp_dn, *comp_dn;
int length;
char **dnptrs, **lastdnptr;

dn_expand(msg, eomorig, comp_dn, exp_dn, length)
char *msg, *eomorig, *comp_dn, exp_dn;
int  length;
```

DESCRIPTION

These routines are used for making, sending and interpreting packets to Internet domain name servers. Global information that is used by the resolver routines is kept in the variable *_res*. Most of the values have reasonable defaults and can be ignored. Options stored in *_res.options* are defined in `resolv.h` and are as follows. Options are a simple bit mask and are or'ed in to enable.

RES_INIT
: True if the initial name server address and default domain name are initialized (i.e., `res_init` has been called).

RES_DEBUG
: Print debugging messages.

RES_AAONLY
: Accept authoritative answers only. `Res_send` will continue until it finds an authoritative answer or finds an error. Currently this is not implemented.

RES_USEVC
: Use TCP connections for queries instead of UDP.

RES_STAYOPEN
Used with RES_USEVC to keep the TCP connection open between queries. This is useful only in programs that regularly do many queries. UDP should be the normal mode used.

RES_IGNTC
Unused currently (ignore truncation errors, i.e., don't retry with TCP).

RES_RECURSE
Set the recursion desired bit in queries. This is the default. (`res_send` does not do iterative queries and expects the name server to handle recursion.)

RES_DEFNAMES
Append the default domain name to single label queries. This is the default.

`Res_init`

reads the initialization file to get the default domain name and the Internet address of the initial hosts running the name server. If this line does not exist, the host running the resolver is tried. `Res_mkquery` makes a standard query message and places it in *buf*. `Res_mkquery` will return the size of the query or −1 if the query is larger than *buflen*. *Op* is usually QUERY but can be any of the query types defined in `nameser.h`. *Dname* is the domain name. If *dname* consists of a single label and the RES_DEFNAMES flag is enabled (the default), *dname* will be appended with the current domain name. 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 will call `res_init` if RES_INIT is not set, send the query to the local name server, and handle timeouts and retries. The length of the message is returned or −1 if there were errors.

`Dn_expand` expands the compressed domain name *comp_dn* to a full domain name. Expanded names are converted to upper case. *Msg* is a pointer to the beginning of the message, *exp_dn* is a pointer to a buffer of size *length* for the result. The size of compressed name is returned or -1 if there was an error.

`Dn_comp` compresses the domain name *exp_dn* and stores it in `comp_dn`. The size of the compressed name is returned or -1 if there were errors. *length* is the size of the 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 to the beginning of the message and the list ends with NULL. *lastdnptr* is a pointer to the end of the array pointed to *dnptrs*. A side effect is to update the list of pointers for labels inserted into the message by `dn_comp` as the name is compressed. If *dnptr* is NULL, we don't try to compress names. If *lastdnptr* is NULL, we don't update the list.

FILES

`/etc/resolv.conf`

SEE ALSO

`named(1M), resolve.conf(5).`

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(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(1M).

If the call succeeds, a socket of type SOCK_STREAM is returned to the caller, and given to the remote command as stdin and stdout. If *fd2p* is non-zero, then 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 being DG/UX system signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the stderr (unit 2 of the remote command) will be made the same as the stdout and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

SEE ALSO

rexecd(1M), rcmd(3X).

BUGS

There is no way to specify options to the socket call that rexec makes.

NAME

`rindex` – search for the last occurrence of a character in a string

SYNOPSIS

```
#include <string.h>

char *search, c, *rindex(), *result;
...
result = rindex(search, c);
```

where:

search The string to inspect
c The character you want to match

DESCRIPTION

Use the `rindex` function to find the last occurrence of a specified character in a string. This function is the same as the `strrchr` function.

The include file `string.h` defines this function.

EXAMPLE

```
/* Program test for the rindex() function */

#include <string.h>
#include <stdio.h>
#define MAX     80

char  c, string[MAX], *rindex();
int   result, i = 1;

main(argc, argv)

int    argc;
char  *argv[];
{
    printf("Character template?\n");
    scanf("%c", &c);
    while (i < argc) {
        sprintf(string, "%s", argv[i]);
        if ((result = strrchr(string, c)) == NULL)
            printf("Character '%c' does not occur in \n\t'%s'\n",
            c, string);
        else
            printf("Last occurrence of '%c' in\n\t'%s'\nat %o.\n",
            c, string, result);
        i++;
    }
}
```

A call to the program test with the strings element, digital, and execute generates the output

```
Character template?
e
Last occurrence of 'e' in
```

```
        'element'
at 34000023362.
Character 'e' does not occur in
        'digital'
Last occurrence of 'e' in
        'execute'
at 34000023364.
```

(The locations will vary with execution.)

RETURNS

The `rindex` function returns NULL if the character does not occur in the string. Otherwise it returns a pointer to the last occurrence of the character.

SEE ALSO

`index`(3C), `memchr`(3C), `strchr`(3C), `strrchr`(3C).

NAME

auth_destroy, authnone_create, authdes_create, authdes_getucred, authunix_create, authunix_create_default, callrpc, clnt_broadcast, clnt_call, clnt_destroy, clnt_create, clnt_control, clnt_freeres, clnt_geterr, clnt_pcreateerror, clnt_perrno, clnt_perror, clnt_spcreateerror, clnt_sperrno, clnt_sperror, clntraw_create, clnttcp_create, clntudp_create, host2netname, key_decryptsession, key_encryptsession, key_gendes, key_setsecret, get_myaddress, getnetname, netname2host, netname2user, pmap_getmaps, pmap_getport, pmap_rmtcall, pmap_set, pmap_unset, registerrpc, svc_destroy, svc_freeargs, svc_getargs, svc_getcaller, svc_getreqset, svc_getreq, svc_register, svc_run, svc_sendreply, svc_unregister, svcerr_auth, svcerr_decode, svcerr_noproc, svcerr_noprog, svcerr_progvers, svcerr_systemerr, svcerr_weakauth, svcraw_create, svctcp_create, svcfd_create, svcudp_create, user2netname, xdr_accepted_reply, xdr_authunix_parms, xdr_callhdr, xdr_callmsg, xdr_opaque_auth, xdr_pmap, xdr_pmaplist, xdr_rejected_reply, xdr_replymsg, xprt_register, xprt_unregister – library routines for remote procedure calls

SYNOPSIS AND DESCRIPTION

These routines let C programs make procedure calls on other machines across the network. First, the client calls a procedure to send a data packet to the server. Upon receipt of the packet, 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.

```
#include <rpc/rpc.h>
```

void
auth_destroy(*auth*)
AUTH **auth*;

> A macro that destroys 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 returns an RPC authentication handle that passes nonusable authentication information with each remote procedure call. This is the

default authentication used by RPC.

```
AUTH *
authdes_create(name, window, syncaddr, ckey)
char *name;
unsigned window;
struct sockaddr *syncaddr;
des_block *ckey;
```

NOTE: Secure RPC using DES Authentication is an additional feature that must be purchased separately from the DG/UX™ ONC™/NFS® product.

authdes_create( ) is the first of two routines which interface to the RPC secure authentication system, known as DES authentication. The second is authdes_getucred( ), below. Note: the keyserver daemon keyserv(8) must be running for the DES authentication system to work.

authdes_create( ), used on the client side, returns an authentication handle that will enable the use of the secure authentication system. The first parameter *name* is the network name, or *netname*, of the owner of the server process. This field usually represents a *hostname* 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 resynchronizations. If an address is supplied, however, then the system will use the address for consulting the remote time service whenever resynchronization is required. This parameter is usually the address of the RPC server itself. The final parameter *ckey* 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 it is supplied, however, then it will be used instead.

```
authdes_getucred(adc, uid, gid, grouplen, groups)
struct authdes_cred *adc;
short *uid;
short *gid;
short *grouplen;
int *groups;
```

authdes_getucred( ), the second of the two DES authentication routines, is used on the server side for converting a DES credential, which is operating system independent, into a UNIX credential. This routine differs from utility routine netname2user in that authdes_getucred( ) pulls its information from a cache, and does not have to do a Network Information Services (NIS) lookup everytime it is called to get its information.

```
AUTH *
authunix_create(host, uid, gid, len, aup_gids)
char *host;
int uid, gid, len, *aup.gids;
```

Create and return an RPC authentication handle that contains UNIX 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 ; *len* and *aup_gids* refer to a counted array of groups to which the user belongs. It is easy to impersonate a user.

```
AUTH *
authunix_create_default( )
```

Calls `authunix_create( )` with the appropriate parameters.

```
callrpc(host, prognum, versnum, procnum, inproc, in, outproc, out)
char *host;
u_long prognum, versnum, procnum;
char *in, *out;
xdrproc_t inproc, 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(s), and *out* is the address of where to place the result(s); *inproc* is used to encode the procedure's parameters, and *outproc* is used to decode the procedure's results. This routine returns zero if it succeeds, or the value of `enum clnt_stat` cast to an integer if it fails. The routine `clnt_perrno( )` is handy for translating failure statuses into messages.

Warning: calling remote procedures with this routine uses UDP/IP as a transport; see `clntudp_create( )` 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, *out;
xdrproc_t inproc, outproc;
resultproc_t eachresult;
```

Like `callrpc( )`, except the call message is broadcast to all locally connected broadcast nets. Each time it receives a response, this routine calls `eachresult( )`, whose form is:

```
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 zero, `clnt_broadcast( )` waits for more replies; otherwise it returns with appropriate status.

Warning: broadcast sockets are limited in size to the maximum transfer unit of the data link. For ethernet, this value is 1500 bytes.

```
enum clnt_stat
clnt_call(clnt, procnum, inproc, in, outproc, out, tout)
CLIENT *clnt;
u_long
procnum;
xdrproc_t inproc, outproc;
char *in, *out;
struct timeval tout;
```

A macro that calls the remote procedure *procnum* associated with the client handle, *clnt*, which is obtained with an RPC client creation routine such as `clnt_create( )`. The parameter *in* is the address of the procedure's argument(s), and *out* is the address of where to place the result(s); *inproc* is used to encode the procedure's parameters, and *outproc* is used to decode the procedure's results; *tout* is the time allowed for results to come back.

```
clnt_destroy(clnt)
CLIENT *clnt;
```

A macro that destroys 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, it will close it also. Otherwise, the socket remains open.

```
CLIENT *
clnt_create(host, prog, vers, proto)
char *host;
u_long prog, vers;
char *proto;
```

Generic client creation routine. *host* identifies the name of the remote host where the server is located. *proto* indicates which kind of transport protocol to use. The currently supported values for this field are udp and tcp. Default timeouts are set, but can be modified using `clnt_control( )`.

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 large arguments or return huge results.

```
bool_t
clnt_control(cl, req, info)
CLIENT *cl;
char *info;
```

A macro used to change or retrieve various information about a client object. *req* indicates the type of operation, and *info* is a pointer to the information. For both UDP and TCP, the supported values of *req* and their argument types and what they do are:

CLSET_TIMEOUT	struct
timeval	set
total	
timeout	
CLGET_TIMEOUT	struct

timeval	get
total	
timeout	

Note: if you set the timeout using `clnt_control( )`, the timeout parameter passed to `clnt_call( )` will be ignored in all future calls.

CLGET_SERVER_ADDR	struct
sockaddr	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.

```
clnt_freeres(clnt, outproc, out)
CLIENT *clnt;
xdrproc_t outproc;
char *out;
```

A macro that frees 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 one if the results were successfully freed, and zero otherwise.

```
void
clnt_geterr(clnt, errp)
CLIENT *clnt;
struct rpc_err *errp;
```

A macro that copies the error structure out of the client handle to the structure at address *errp*.

```
void
clnt_pcreateerror(s)
char *s;
```

Print a message to standard error indicating why a client RPC handle could not be created. The message is prepended with string *s* and a colon. Used when a `clnt_create( )`, `clntraw_create( )`, `clnttcp_create( )`, or `clntudp_create( )` call fails.

```
void
clnt_perrno(stat)
enum clnt_stat stat;
```

Print a message to standard error corresponding to the condition indicated by *stat*. Used after `callrpc( )`.

```
clnt_perror(clnt, s)
CLIENT *clnt;
char *s;
```

Print a message to 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. Used after `clnt_call( )`.

```
char *
clnt_spcreateerror
char *s;
```

Like `clnt_pcreateerror( )`, except that it returns a string instead of printing to the standard error.

Bugs: returns pointer to static data that is overwritten on each 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 call failed, return a pointer to a string which contains the message. The string ends with a NEWLINE.

`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`, or if a message format different than that supported by `clnt_perrno( )` is to be used. Note: unlike `clnt_sperror( )` and `clnt_spcreaterror( )`, `clnt_sperrno( )` does not return pointer to static data so the result will not get overwritten on each call.

```
char *
clnt_sperror(rpch, s)
CLIENT *rpch;
char *s;
```

Like `clnt_perror( )`, except that (like `clnt_sperrno( )`) it returns a string instead of printing to standard error.

Bugs: returns pointer to static data that is overwritten on each call.

```
CLIENT *
clntraw_create(prognum, versnum)
u_long prognum, versnum;
```

This routine creates a toy 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; see `svcraw_create( )`. This allows simulation of RPC and acquisition of RPC overheads, such as round trip times, without any kernel interference. This routine returns NULL if it fails.

```
CLIENT *
clnttcp_create(addr, prognum, versnum, sockp, sendsz, recvsz)
struct sockaddr_in *addr;
u_long prognum, versnum;
int *sockp;
u_int sendsz, recvsz;
```

This routine creates an RPC client 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, then it is set to the actual port that the remote program is listening on (the remote `portmap` service is consulted for this information). The parameter *sockp* is a socket; if it is RPC_ANYSOCK, then this routine opens a new one and sets *sockp*. 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 suitable defaults. This routine returns NULL if it fails.

```
CLIENT *
clntudp_create(addr, pronum, versnum, wait, sockp)
struct sockaddr_in *addr;
u_long prognum, versnum;
struct timeval wait;
int *sockp;
```

This routine creates an RPC client for the remote program *prognum*, version *versnum*; the client uses use UDP/IP as a transport. The remote program is located at 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 socket; if it is RPC_ANYSOCK, then this routine opens a new one and sets *sockp*. 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( )`.

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 large arguments or return huge results.

```
host2netname(name, host, domain)
char *name;
char *host;
char *domain;
```

Convert from a domain-specific hostname to an operating-system independent netname. Return TRUE if it succeeds and FALSE if it fails. Inverse of netname2host( ).

```
key_decryptsession(remotename, deskey)
char *remotename;
des_block *deskey;
```

key_decryptsession( ) is an interface 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 a server netname and a des key, and decrypts the key by using the the public key of the the server and the secret key associated with the effective uid of the calling process. It is the inverse of key_encryptsession( ).

```
key_encryptsession(remotename, deskey)
char *remotename;
des_block *deskey;
```

key_encryptsession( ) is a keyserver interface routine. It takes a server netname and a des key, and encrypts it using the public key of the the server and the secret key associated with the effective uid of the calling process. It is the inverse of key_decryptsession( ).

```
key_gendes(deskey)
des_block *deskey;
```

key_gendes( ) is 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.

```
key_setsecret(key)
char *key;
```

key_setsecret( ) is a keyserver interface routine. It is used to set the key for the effective *uid* of the calling process.

```
void
get_myaddress(addr)
struct sockaddr_in *addr;
```

Stuff the machine's IP address into **addr*, without consulting the library routines that deal with /etc/hosts. The port number is always set to htons(PMAPPORT).

```
getnetname(name)
char name[MAXNETNAMELEN];
```

getnetname( ) installs the unique, operating-system independent netname of the caller in the fixed-length array *name*. Returns TRUE if it succeeds and FALSE if it fails.

```
netname2host(name, host, hostlen)
char *name;
char *host;
int hostlen;
```

Convert from an operating-system independent netname to a domain-specific hostname. Returns TRUE if it succeeds and FALSE if it fails. Inverse of host2netname( ).

```
netname2user(name, uidp, gidp, gidlenp, gidlist)
char *name;
int *uidp;
int *gidp;
int *gidlenp;
int *gidlist;
```

Convert from an operating-system independent netname to a domain-specific user ID. Returns TRUE if it succeeds and FALSE if it fails. Inverse of user2netname( ).

```
struct pmaplist *
pmap_getmaps(addr)
struct sockaddr_in *addr;
```

A user interface to the portmap service, which returns a list of the current RPC program-to-port mappings on the host located at IP address **addr*. This routine can return NULL . The command '**rpcinfo −p**' uses this routine.

```
u_short
pmap_getport(addr, prognum, versnum, protocol)
struct sockaddr_in *addr;
u_long prognum, versnum, protocol;
```

A user interface to the portmap service, which returns the port number on which waits a service that supports program number *prognum*, version *versnum*, and speaks the transport protocol associated with *protocol*. The value of *protocol* is most likely IPPROTO_UDP or IPPROTO_TCP. A return value of zero means that the mapping does not exist or that the RPC system failured to contact the remote portmap service. In the latter case, the global variable rpc_createerr( ) contains the RPC status.

```
enum clnt_stat
pmap_rmtcall(addr, prognum, versnum, procnum, inproc, in, outproc,
     out, tout, portp)
struct sockaddr_in *addr;
u_long prognum, versnum, procnum;
char *in, *out;
xdrproc_t inproc, outproc;
struct timeval tout;
u_long *portp;
```

A user interface to the portmap service, which instructs portmap on the host at IP address **addr* to make an RPC call on your behalf to a procedure on that host. The parameter **portp* will be modified to the program's port number if the procedure succeeds. The definitions of other parameters are discussed in `callrpc( )` and `clnt_call( )`. This procedure should be used for a ping and nothing else. See also `clnt_broadcast( )`.

```
pmap_set(prognum, versnum, protocol, port)
u_long prognum, versnum, protocol;
u_short port;
```

A user interface to the portmap service, which establishes a mapping between the triple [*prognum*,*versnum*,*protocol*] and *port* on the machine's portmap service. The value of *protocol* is most likely `IPPROTO_UDP` or `IPPROTO_TCP`. This routine returns one if it succeeds, zero otherwise. Automatically done by `svc_register( )`.

```
pmap_unset(prognum, versnum)
u_long prognum, versnum;
```

A user interface to the portmap service, which destroys all mapping between the triple [*prognum*,*versnum*,*] and ports on the machine's portmap service. This routine returns one if it succeeds, zero otherwise.

```
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(s); *progname* should return a pointer to its static result(s); *inproc* is used to decode the parameters while *outproc* is used to encode the results. This routine returns zero if the registration succeeded, −1 otherwise.

Warning: remote procedures registered in this form are accessed using the UDP/IP transport; see `svcudp_create( )` for restrictions.

```
struct rpc_createerr     rpc_createerr;
```

A global variable whose value is set by any RPC client creation routine that does not succeed. Use the routine `clnt_pcreateerror( )` to print the reason why.

```
svc_destroy(xprt)
SVCXPRT *xprt;
```

A macro that destroys 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.

```
fd_set svc_fdset;
```

A global variable reflecting the RPC service side'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 his 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.

```
int svc_fds;
```

Similar to `svc_fedset( )`, but limited to 32 descriptors. This interface is obsoleted by `svc_fdset( )`.

```
svc_freeargs(xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;
```

A macro that frees any data allocated by the RPC/XDR system when it decoded the arguments to a service procedure using `svc_getargs( )`. This routine returns 1 if the results were successfully freed, and zero otherwise.

```
svc_getargs(xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;
```

A macro that decodes 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 one if decoding succeeds, and zero 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*.

```
svc_getreqset(rdfds)
fd_set *rdfds;
```

This routine is only of interest if a service implementor does not call `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)** ; *rdfds* is the resultant read file descriptor bit mask. The routine returns when all sockets associated with the value of *rdfds* have been serviced.

```
svc_getreq(rdfds)
int rdfds;
```

Similar to `svc_getreqset( )`, but limited to 32 descriptors. This interface is obsoleted by `svc_getreqset( )`.

```
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, then 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 one if it succeeds, and zero otherwise.

```
svc_run( )
```

This routine never returns. 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.

```
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 one if it succeeds, zero otherwise.

```
void
svc_unregister(prognum, versnum)
u_long prognum, versnum;
```

Remove all mapping of the double [*prognum*,*versnum*] to dispatch routines, and of the triple [*prognum*,*versnum*,*] to port number.

```
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 its parameters. See also `svc_getargs( )`.

```
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 (but correct) authentication parameters. The routine calls `svcerr_auth(xprt, AUTH_TOOWEAK)`.

```
SVCXPRT *
svcraw_create( )
```

This routine creates a toy RPC service transport, to which it returns a pointer. The transport is really a buffer within the process's address space, so the corresponding RPC client should live in the same address space; see `clntraw_create( )`. This routine allows simulation of RPC and acquisition of RPC overheads (such as round trip times), without any kernel interference. This routine returns NULL if it fails.

```
SVCXPRT *
svctcp_create(sock, send_buf_size, recv_buf_size)
int sock;
u_int send_buf_size, recv_buf_size;
```

This routine creates a TCP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket *sock*, which may be RPC_ANYSOCK, in which case 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, nd `xprt->xp_port` is the transport's port number. This routine returns NULL if it fails. Since TCP-based RPC uses buffered I/O , users may specify the size of buffers; values of zero choose suitable defaults.

```
void
svcfd_create(fd, sendsize, recvsize)
int fd;
u_int sendsize;
u_int recvsize;
```

Create a service on top of any open desciptor. Typically, this descriptor is a connected socket for a stream protocol such as TCP. *sendsize* and *recvsize* indicate sizes for the send and receive buffers. If they are zero, a reasonable default is chosen.

```
SVCXPRT *
svcudp_create(sock)
int sock;
```

This routine creates a UDP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket *sock*, which may be **RPC_ANYSOCK** , in which case 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 transport's socket descriptor, and `xprt->xp_port` is the transport's port number. This routine returns NULL if it fails.

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 large arguments or return huge results.

```
user2netname(name, uid, domain)
char *name;
int uid;
char *domain;
```

Convert from a domain-specific username to an operating-system independent netname. Returns TRUE if it succeeds and FALSE if it fails. Inverse of `netname2user( )`.

```
xdr_accepted_reply(xdrs, ar)
XDR *xdrs;
struct accepted_reply *ar;
```

Used for encoding RPC reply messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

```
xdr_authunix_parms(xdrs, aupp)
XDR *xdrs;
struct authunix_parms *aupp;
```

Used for describing UNIX credentials. This routine is useful for users who wish to generate these credentials without using the RPC authentication package.

```
void
xdr_callhdr(xdrs, chdr)
XDR *xdrs;
struct rpc_msg *chdr;
```

Used for describing RPC call header messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

```
xdr_callmsg(xdrs, cmsg)
XDR *xdrs;
struct rpc_msg *cmsg;
```

Used for describing RPC call messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

```
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.

```
xdr_pmap(xdrs, regs)
XDR *xdrs;
struct pmap *regs;
```

Used for describing parameters to various portmap procedures, externally. This routine is useful for users who wish to generate these parameters without using the pmap interface.

```
xdr_pmaplist(xdrs, rp)
XDR *xdrs;
struct pmaplist **rp;
```

Used for describing a list of port mappings, externally. This routine is useful for users who wish to generate these parameters without using the pmap interface.

```
xdr_rejected_reply(xdrs, rr)
XDR *xdrs;
struct rejected_reply *rr;
```

Used for describing RPC reply messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

```
xdr_replymsg(xdrs, rmsg)
XDR *xdrs;
struct rpc_msg *rmsg;
```

Used for describing RPC reply messages. This routine is useful for users who wish to generate RPC style messages without using the RPC package.

```
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.

SEE ALSO

xdr(3N), keyserv(8).

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

The `rtime` function 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.

SEE ALSO

`gettimeofday`(2), `ftime`(3C).

NAME

`scandir`, `alphasort` – scan a directory

SYNOPSIS

```
#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`(3C). It returns the number of entries in the array and a pointer to the array through *namelist*.

The *select* parameter is a pointer to a user supplied subroutine which is called by `scandir` to select which entries are to be included in the array. The select routine is passed a pointer to a directory entry and should return a non-zero value if the directory entry is to be included in the array. If *select* is null, then all the directory entries will be included.

The *compar* parameter is a pointer to a user supplied subroutine which is passed to `qsort`(3C) to sort the completed array. If this pointer is null, the array is not sorted. `Alphasort` is a routine which can be used for the *compar* parameter to sort the array alphabetically.

The memory allocated for the array can be deallocated with `free` (see `malloc`(3C)) by freeing each pointer in the array and the array itself.

DIAGNOSTICS

Returns -1 if the directory cannot be opened for reading or if `malloc`(3C) cannot allocate enough memory to hold all the data structures.

SEE ALSO

`directory`(3C), `malloc`(3C), `qsort`(3C), `dir`(5)

NAME

scanf, fscanf, sscanf – convert formatted input

SYNOPSIS

```
#include <stdio.h>

int scanf(const char *format, ...);

int fscanf(FILE *strm, const char *format, ...);

int sscanf(const char *s, const char *format, ...);
```

DESCRIPTION

scanf reads from the standard input stream, stdin.

fscanf reads from the stream *strm*.

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. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are simply ignored.

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

1. White-space characters (blanks, tabs, new-lines, or form-feeds) that, except in two cases described below, cause input to be read up to the next non-white-space character.
2. An ordinary character (not %) that must match the next character of the input stream.
3. Conversion specifications consisting of the character % or the character sequence %*digits*$, an optional assignment suppression character *, a decimal digit string that specifies an optional numerical maximum field width, an optional letter l (ell), L, or h indicating the size of the receiving object, and a conversion code. The conversion specifiers d, i, and n should be preceded by h if the corresponding argument is a pointer to short int rather than a pointer to int, or by l if it is a pointer to long int. Similarly, the conversion specifiers o, u, and x should be preceded by h if the corresponding argument is a pointer to unsigned short int rather than a pointer to unsigned int, or by l if it is a pointer to unsigned long int. Finally, the conversion specifiers e, f, and g should be preceded by l if the corresponding argument is a pointer to double rather than a pointer to float, or by L if it is a pointer to long double. The h, l, or L modifier is ignored with any other conversion specifier.

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 character *. The suppression of assignment provides a way of describing an input field that is to be skipped. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the maximum field width, if one is specified, is exhausted. For all descriptors except the character [and the character c, white space leading an input field is ignored.

Conversions can be applied to the *nth* argument in the argument list, rather than to the next unused argument. In this case, the conversion character % (see above) is replaced by the sequence %*digits*$ where *digits* is a decimal integer *n*, giving the position of the argument in the argument list. The first such argument, %1$, immediately follows *format*. The control string can contain either form of a conversion specification, i.e., % or %*digits*$, although the two forms cannot be mixed within a single control string.

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

% A single % is expected in the input at this point; no assignment is done.

d Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the `strtol` function with the value 10 for the *base* argument. The corresponding argument should be a pointer to integer.

u Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the `strtoul` function with the value 10 for the *base* argument. The corresponding argument should be a pointer to unsigned integer.

o Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the `strtoul` function with the value 8 for the *base* argument. The corresponding argument should be a pointer to unsigned integer.

x Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the `strtoul` function with the value 16 for the *base* argument. The corresponding argument should be a pointer to unsigned integer.

i Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the `strtol` function with the value 0 for the *base* argument. The corresponding argument should be a pointer to integer.

n No input is consumed. The corresponding argument should be a pointer to integer into which is to be written the number of characters read from the input stream so far by the call to the function. Execution of a %n directive does not increment the assignment count returned at the completion of execution of the function.

e,f,g Matches an optionally signed floating point number, whose format is the same as expected for the subject string of the `strtod` function. The corresponding argument should be a pointer to floating.

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 Matches a sequence of characters of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument should be a pointer to the initial character of an array large enough to accept the sequence. No null character is added. The normal skip over white space is suppressed.

[Matches a nonempty sequence of characters from a set of expected characters (the *scanset*). The corresponding argument should be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically. The conversion specifier includes all subsequent characters in the *format* string, up to and including the matching right bracket (]). The characters between the brackets (the *scanlist*) comprise the scanset, unless the character after the left bracket is a circumflex (^), in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with [] or [^], the right bracket character is in the scanlist and the next right bracket character is the matching right bracket that ends the specification; otherwise the first right bracket character is the one that ends the specification.

A range of characters in the scanset 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 character - will also stand for itself whenever it is the first or the last character in the scanlist. To include the right bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanlist and in this case it will not be syntactically interpreted as the closing bracket. At least one character must match for this conversion to be considered successful.

p Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the %p conversion of the printf function. The corresponding argument should be a pointer to void. The interpretation of the input item is implementation-defined. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise, the behavior of the %p conversion is undefined.

If an invalid conversion character follows the %, the results of the operation may not be predictable.

The conversion specifiers E, G, and X are also valid and, under the -Xa and -Xc compilation modes [see cc(1)], behave the same as e, g, and x, respectively. Under the -Xt compilation mode, E, G, and X behave the same as le, lg, and lx, respectively.

Each function allows for detection of a language-dependent decimal point character in the input string. The decimal point character is defined by the program's locale (category LC_NUMERIC). In the "C" locale, or in a locale where the decimal point character is not defined, the decimal point character defaults to a period (.).

The scanf conversion terminates at end of file, at the end of the control string, or when an input character conflicts with the control string.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. Trailing white space (including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the `%n` directive.

EXAMPLES

The call to the function `scanf`:

```
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`.

The call to the function `scanf`:

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

with the input line:

```
56789 0123 56a72
```

will assign `56` to `i`, `789.0` to `x`, skip `0123`, and place the characters `56\0` in `name`. The next character read from `stdin` will be `a`.

SEE ALSO

cc(1), printf(3S), strtod(3C), strtol(3C), strtoul(3C).

DIAGNOSTICS

These routines return the number of successfully matched and assigned input items; this number can be zero in the event of an early matching failure between an input character and the control string. If the input ends before the first matching failure or conversion, `EOF` is returned.

NAME

scanf, fscanf, sscanf – convert formatted input

SYNOPSIS

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

int scanf(const char *format [, pointer] ... );

int fscanf(FILE *stream, const char *format [, pointer] ... );

int sscanf(char *s, const char *format [, pointer] ... );
```

DESCRIPTION (International Functions)

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 (bytes), interprets them according to a control string *format*, and stores the results in its arguments.

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

A. White-space characters (characters are defined in isspace() of ctype(3C)). Except in two cases described below, these cause input to be read up to the next non-white-space character.

B. An ordinary character (any EUC character , except the ASCII character %), which must match the next byte of the input stream.

C. Conversion specifications which direct the conversion of the next input field. Only ASCII characters are allowed as conversion characters.

The conversion code indicates the interpretation of the input field, and the corresponding pointer argument must match the type being read. wc and ws are the new conversion specifications for wchar_t character control, and both may be used in all three functions.

wc A wchar_t character is expected; the character, which should be in EUC, is transformed into a wchar_t character, and stored in the location pointed to by the corresponding argument which should be a wchar_t pointer. The normal skip over white space is suppressed in this case. To read the next non-space character as the wchar_t character, %1ws should be used. If a field width is given, the corresponding argument should refer to a wchar_t array; the indicated number of wchar_t characters are read.

ws A wchar_t string is expected; characters in EUC are transformed into wchar_t characters and stored in the location pointed to by the corresponding argument. The corresponding argument should be a pointer pointing to a wchar_t array large enough to accept the string and a terminating wchar_t null character, which is added automatically. wchar_t characters are read until the number of wchar_t characters specified in the field width, if supplied, or a white-space character is read.

The conversion of these functions terminate at EOF or a NULL character in the case of sscanf(), at the end of the control string, or when an input character conflicts with the control string. In the last case, the offending character is left unread in the input stream.

These functions return the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, EOF is returned.

WARNING

A character from a supplementary code set in a `scanset` enclosed in a pair of square brackets is simply interpreted as a byte string. Each byte of the input field is compared to the byte in the `scanset`.

SEE ALSO

`printf`(3W), `vprintf`(3W), `widec`(3W).
`scanf`(3S), `stdio`(3S).

NAME

setbuf, setvbuf – assign buffering to a stream

SYNOPSIS

```
#include <stdio.h>

void setbuf (FILE *stream, char *buf);

int setvbuf (FILE *stream, char *buf, int type, size_t size);
```

DESCRIPTION

setbuf may be used after a *stream* [see intro(3)] 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.

While there is no limititation on the size of the buffer, the constant BUFSIZ, defined in the <stdio.h> header file, is typically a good buffer size:

```
char buf[BUFSIZ];
```

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

_IOFBF causes input/output to be fully buffered.

_IOLBF causes output to be line buffered; the buffer will be flushed when a newline is written, the buffer is full, or input is requested.

_IONBF causes input/output to be completely unbuffered.

If *buf* is not the NULL pointer, the array it points to will be used for buffering, instead of an automatically allocated buffer. *size* specifies the size of the buffer to be used. If input/output is unbuffered, *buf* and *size* are ignored.

For a further discussion of buffering, see stdio(3S).

SEE ALSO

fopen(3S), getc(3S), malloc(3C), putc(3S), stdio(3S).

DIAGNOSTICS

If an illegal value for *type* is provided, setvbuf returns a non-zero value. Otherwise, it returns zero.

NOTES

A common source of error is allocating buffer space as an "automatic" variable in a code block, and then failing to close the stream in the same block.

Parts of buf will be used for internal bookkeeping of the stream and, therefore, buf will contain less than *size* bytes when full. It is recommended that the automatically allocated buffer is used when using setvbuf.

NAME

setbuffer – assign a buffer to a specified stream

SYNOPSIS

```
#include <stdio.h>
. . .
FILE *fp;
char *bufptr;
int size;
void setbuffer();
. . .
setbuffer(fp, bufptr, size); (or) setbuffer(fp, NULL, size);
```

where size is the size of the character array bufptr.

DESCRIPTION

The setbuffer function assigns a buffer to a stream whose I/O you have been handling with the stdio(3S) functions. Output is sent to the file or device only when you call fflush(3S) or when the buffer fills up. This function is an alternate form of setbuf. If the buffer pointer bufptr is null (0), it specifies an unbuffered file and uses single byte I/O. Otherwise, the routine assumes that the buffer can hold at least the number of characters specified by size.

RETURN VALUE

setbuffer does not return a value.

SEE ALSO

setbuf(3S), setlinebuf(3C).

NAME

setjmp, longjmp – non-local goto

SYNOPSIS

```
#include <setjmp.h>

int setjmp (jmp_buf env);

void longjmp (jmp_buf env, int val);
```

DESCRIPTION

These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

setjmp saves its stack environment in env (whose type, *jmp_buf*, is defined in the <setjmp.h> header file) for later use by longjmp. It returns the value 0.

longjmp restores the environment saved by the last call of setjmp with the corresponding env argument. After longjmp is completed, program execution continues as if the corresponding call of setjmp had just returned the value val. (The caller of setjmp must not have returned in the interim.) longjmp cannot cause setjmp to return the value 0. If longjmp is invoked with a second argument of 0, setjmp will return 1. At the time of the second return from setjmp, all external and static variables have values as of the time longjmp is called (see example). The values of register and automatic variables are undefined.

Register or automatic variables whose value must be relied upon must be declared as volatile.

EXAMPLE

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

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

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

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

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

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

SEE ALSO

signal(2), sigsetjmp(3C).

NOTES

If longjmp is called even though env was never primed by a call to setjmp, or when the last such call was in a function that has since returned, absolute chaos is guaranteed.

NAME

setlinebuf – assign line buffering for a specified stream

SYNOPSIS

```
#include <stdio.h>
...
FILE *fp;
void setlinebuf();
. . .
setlinebuf(fp);
```

DESCRIPTION

The setlinebuf function assigns line buffering for a stream whose I/O you are handling with the stdio(3S) functions. Output will be sent to the file or device when a line terminator is printed or when you call fflush(3S).

RETURNS

setlinebuf does not return a value.

SEE ALSO

setbuf(3S), setbuffer(3C).

NAME

setlocale – modify and query a program's locale

SYNOPSIS

```
#include <locale.h>

char *setlocale (int category, const char *locale);
```

DESCRIPTION

setlocale selects the appropriate piece of the program's locale as specified by the *category* and *locale* arguments. The *category* argument may have the following values: LC_CTYPE, LC_NUMERIC, LC_TIME, LC_COLLATE, LC_MONETARY, LC_MESSAGES and LC_ALL. These names are defined in the locale.h header file. LC_CTYPE affects the behavior of the character handling functions (isdigit, tolower, etc.) and the multibyte character functions (such as mbtowc and wctomb). LC_NUMERIC affects the decimal-point character for the formatted input/output functions and the string conversion functions as well as the non-monetary formatting information returned by localeconv. [See localeconv(3C).]. LC_TIME affects the behavior of ascftime, cftime, getdate and strftime. LC_COLLATE affects the behavior of strcoll and strxfrm, and the regular expression code described in regexpr3C). LC_MONETARY affects the monetary formatted information returned by localeconv. LC_MESSAGES affects the behavior of gettxt, catopen, catclose, and catgets. [See catopen(3C) and catgets(3C).] LC_ALL names the program's entire locale.

Each category corresponds to a set of databases which contain the relevant information for each defined locale. The location of a database is given by the following path, /usr/lib/locale/*locale*/*category*, where *locale* and *category* are the names of locale and category, respectively. For example, the database for the LC_CTYPE category for the Italian locale would be found in /usr/lib/locale/it/LC_CTYPE.

A value of "C" for *locale* specifies the default environment.

A value of "" for *locale* specifies that the locale should be taken from environment variables. The order in which the environment variables are checked for the various categories is given below:

Category	1st Env. Var.	2nd Env. Var	3rd Env. Var
LC_CTYPE	LC_CTYPE	LANG	CHRCLASS
LC_COLLATE	LC_COLLATE	LANG	
LC_TIME	LC_TIME	LANG	LANGUAGE
LC_NUMERIC	LC_NUMERIC	LANG	
LC_MONETARY	LC_MONETARY	LANG	
LC_MESSAGES	LC_MESSAGES	LANG	

At program startup, the equivalent of

```
setlocale(LC_ALL, "C")
```

is executed automatically. This has the effect of initializing each category to the locale described by the environment "C".

For programs that do not depend upon the C locale, the normal use of setlocale is to execute

```
setlocale(LC_ALL, "")
```

as the first action in the application code. This has the effect of initializing each category according to the environment variables described above.

If a pointer to a string is given for *locale*, `setlocale` attempts to set the locale for the given category to *locale*. If `setlocale` succeeds, *locale* is returned. If `setlocale` fails, a null pointer is returned and the program's locale is not changed.

For category `LC_ALL`, the behavior is slightly different. If a pointer to a string is given for *locale* and `LC_ALL` is given for *category*, `setlocale` attempts to set the locale for all the categories to *locale*. The *locale* may be a simple locale, consisting of a single locale, or a composite locale. A composite locale is a string beginning with a "/" followed by the locale of each category separated by a "/". If `setlocale` fails to set the locale for any category, a null pointer is returnedand the program's locale for all categories is not changed. Otherwise, locale is returned.

A null pointer for *locale* causes `setlocale` to return the current locale associated with the *category*. The program's locale is not changed.

FILES

`/usr/lib/locale/C/LC_CTYPE` – `LC_CTYPE` database for the C locale.
`/usr/lib/locale/C/LC_MONETARY` – `LC_MONETARY` database for the C locale.
`/usr/lib/locale/C/LC_NUMERIC` – `LC_NUMERIC` database for the C locale.
`/usr/lib/locale/C/LC_TIME` – `LC_TIME` database for the C locale.
`/usr/lib/locale/C/LC_COLLATE` – `LC_COLLATE` database for the C locale.
`/usr/lib/locale/C/LC_MESSAGES` – `LC_MESSAGES` database for the C locale (this is a directory).
`/usr/lib/locale/`*locale*/*category* – files containing the locale specific information for each locale and category.

SEE ALSO

`testlocale`(1M),
`ctime`(3C), `ctype`(3C), `getdate`(3C), `gettxt`(3G), `localeconv`(3C), `mbtowc`(3C), `printf`(3S), `strcoll`(3C), `strftime`(3C), `strtod`(3C), `strxfrm`(3C), `wctomb`(3C), `environ`(5).

NAME

sigsetjmp, siglongjmp – a non-local goto with signal state

SYNOPSIS

```
#include <setjmp.h>

int sigsetjmp (sigjmp_buf env, int savemask);

void siglongjmp (sigjmp_buf env, int val);
```

DESCRIPTION

These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

sigsetjmp saves the calling process's registers and stack environment [see sigaltstack(2)] in *env* (whose type, sigjmp_buf, is defined in the <setjmp.h> header file) for later use by siglongjmp. If *savemask* is non-zero, the calling process's signal mask [see sigprocmask(2)] and scheduling parameters [see priocntl(2)] are also saved. sigsetjmp returns the value 0.

siglongjmp restores the environment saved by the last call of sigsetjmp with the corresponding *env* argument. After siglongjmp is completed, program execution continues as if the corresponding call of sigsetjmp had just returned the value *val*. siglongjmp cannot cause sigsetjmp to return the value zero. If siglongjmp is invoked with a second argument of zero, sigsetjmp will return 1. At the time of the second return from sigsetjmp, all external and static variables have values as of the time siglongjmp is called. The values of register and automatic variables are undefined. Register or automatic variables whose value must be relied upon must be declared as volatile.

If a signal-catching function interrupts sleep and calls siglongjmp to restore an environment saved prior to the sleep call, the action associated with SIGALRM and time it is scheduled to be generated are unspecified. It is also unspecified whether the SIGALRM signal is blocked, unless the process's signal mask is restored as part of the environment.

The function siglongjmp restores the saved signal mask if and only if the *env* argument was initialized by a call to the sigsetjmp function with a non-zero *savemask* argument.

SEE ALSO

getcontext(2), priocntl(2), sigaction(2), sigaltstack(2), sigprocmask(2), setjmp(3C).

NOTES

If siglongjmp is called even though *env* was never primed by a call to sigsetjmp, or when the last such call was in a function that has since returned, absolute chaos is guaranteed.

NAME

`sigemptyset`, `sigfillset`, `sigaddset`, `sigdelset`, `sigismember` – manipulate sets of signals.

SYNOPSIS

```
#include <signal.h>

int sigemptyset (sigset_t *set);

int sigfillset (sigset_t *set);

int sigaddset (sigset_t *set, int signo);

int sigdelset (sigset_t *set, int signo);

int sigismember (sigset_t *set, int signo);
```

DESCRIPTION

These functions manipulate *sigset_t* data types, representing the set of signals supported by the implementation.

`sigemptyset` initializes the set pointed to by *set* to exclude all signals defined by the system.

`sigfillset` initializes the set pointed to by *set* to include all signals defined by the system.

`sigaddset` adds the individual signal specified by the value of *signo* to the set pointed to by *set*.

`sigdelset` deletes the individual signal specified by the value of *signo* from the set pointed to by *set*.

`sigismember` checks whether the signal specified by the value of *signo* is a member of the set pointed to by *set*.

Any object of type *sigset_t* must be initialized by applying either `sigemptyset` or `sigfillset` before applying any other operation.

`sigaddset`, `sigdelset` and `sigismember` will fail if the following is true:

`EINVAL` The value of the *signo* argument is not a valid signal number.

`sigfillset` will fail if the following is true:

`EFAULT` The *set* argument specifies an invalid address.

SEE ALSO

`sigaction`(2), `sigprocmask`(2), `sigpending`(2), `sigsuspend`(2), `signal`(5).

DIAGNOSTICS

Upon successful completion, the `sigismember` function returns a value of one if the specified signal is a member of the specified set, or a value of zero if it is not. Upon successful completion, the other functions return a value of zero. Otherwise a value of -1 is returned and *errno* is set to indicate the error.

NAME

sinh, sinhf, cosh, coshf, tanh, tanhf, asinh, acosh, atanh – hyperbolic functions

SYNOPSIS

cc [*flag* ...] *file* ... -lm [*library* ...]

```
#include <math.h>

double sinh (double x);

float sinhf (float x);

double cosh (double x);

float coshf (float x);

double tanh (double x);

float tanhf (float x);

double asinh (double x);

double acosh (double x);

double atanh (double x);
```

DESCRIPTION

sinh, cosh, and tanh and the single-precision versions sinhf, coshf, and tanhf return, respectively, the hyberbolic sine, cosine, and tangent of their argument.

asinh, acosh, and atanh return, respectively, the inverse hyperolic sine, cosine, and tangent of their argument.

SEE ALSO

matherr(3M).

DIAGNOSTICS

sinh, sinhf, cosh, and coshf return HUGE (and sinh and sinhf may return -HUGE for negative *x*) when the correct value would overflow and set errno to ERANGE.

acosh returns NaN and sets errno to EDOM when the argument *x* is less than 1. A message indicating DOMAIN error is printed on the standard error output.

atanh returns NaN and sets errno to EDOM if $|x| \geq 1$. If $|x| = 1$, a message indicating SING error is printed on the standard error output; if $|x| > 1$ the message will indicate DOMAIN error.

Except when the -Xc compilation option is used, these error-handling procedures may be changed with the function matherr. When the -Xa or -Xc compilation options are used, HUGE_VAL is returned instead of HUGE and no error messages are printed.

NAME

sleep – suspend execution for interval

SYNOPSIS

```
#include <unistd.h>

unsigned sleep (unsigned seconds);
```

DESCRIPTION

The current process is suspended from execution for the number of *seconds* specified by the argument. The actual suspension time may be less than that requested because any caught signal will terminate the sleep following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount because of the scheduling of other activity in the system. The value returned by sleep will be the "unslept" amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested sleep time, or premature arousal because of another caught signal.

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

SEE ALSO

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

EXAMPLE

```
/* Program test for the sleep() function */

#include <stdio.h>
#include <unistd.h>

unsigned int  hold;

main() {
    printf("How long a nap (in seconds)?\n");
    scanf("%d", &hold);
    sleep(hold);
    printf("I'm awake again.\n");
}
```

If you execute the program test, and answer its query with 5, it pauses for 5 seconds, then generates the following:

```
I'm awake again.
```

NAME

sputl, sgetl – access long integer data in a machine-independent fashion

SYNOPSIS

cc [*flag* ...] *file* ... -lld [*library* ...]

```
#include <ldfcn.h>

void sputl (long value, char *buffer);

long sgetl (const char *buffer);
```

DESCRIPTION

sputl takes the four bytes of the long integer *value* and places them in memory starting at the address pointed to by *buffer*. The ordering of the bytes is the same across all machines.

sgetl retrieves the four bytes in memory starting at the address pointed to by *buffer* and returns the long integer value in the byte ordering of the host machine.

The combination of sputl and sgetl provides a machine-independent way of storing long numeric data in a file in binary form without conversion to characters.

A program that uses these functions must be loaded with the object-file access routine library libld.a.

SEE ALSO

a64l(3C), l3tol(3C).

NAME

ssignal, gsignal – software signals

SYNOPSIS

```
#include <signal.h>

int (*ssignal (int sig, int (*action) (int))) (int);

int gsignal (int sig);
```

DESCRIPTION

ssignal and gsignal implement a software facility similar to signal(2). This facility is made available to users for their own purposes.

Software signals made available to users are associated with integers in the inclusive range 1 through 17. A call to ssignal associates a procedure, *action*, with the software signal *sig*; the software signal, *sig*, is raised by a call to gsignal. Raising a software signal causes the action established for that signal to be *taken*.

The first argument to ssignal is a number identifying the type of signal for which an action is to be established. The second argument defines the action; it is either the name of a (user-defined) *action function* or one of the manifest constants SIG_DFL (default) or SIG_IGN (ignore). ssignal returns the action previously established for that signal type; if no action has been established or the signal number is illegal, ssignal returns SIG_DFL.

gsignal raises the signal identified by its argument, *sig*:

> If an action function has been established for *sig*, then that action is reset to SIG_DFL and the action function is entered with argument *sig*. gsignal returns the value returned to it by the action function.
>
> If the action for *sig* is SIG_IGN, gsignal returns the value 1 and takes no other action.
>
> If the action for *sig* is SIG_DFL, gsignal returns the value 0 and takes no other action.
>
> If *sig* has an illegal value or no action was ever specified for *sig*, gsignal returns the value 0 and takes no other action.

SEE ALSO

signal(2), sigset(2), raise(3C).

NAME

stdio – standard buffered input/output package

SYNOPSIS

```
#include <stdio.h>

FILE *stdin, *stdout, *stderr;
```

DESCRIPTION

The 3S entries in this manual constitute an efficient, user-level I/O buffering scheme. The in-line macros *getc*(3S) and *putc*(3S) handle characters quickly. The macros *getchar* and *putchar*, and the higher-level routines *fgetc*, *fgets*, *fprintf*, *fputc*, *fputs*, *fread*, *fscanf*, *fwrite*, *gets*, *getw*, *printf*, *puts*, *putw*, and *scanf* all use or act as if they use *getc* and *putc*; they can be freely intermixed.

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

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

The following symbolic values in <unistd.h> define the file descriptors that will be associated with the C-language *stdin*, *stdout* and *stderr* when the application is started:

STDIN_FILENO	Standard input value, stdin. It has the value of 0.
STDOUT_FILENO	Standard output value, stdout. It has the value of 1.
STDERR_FILENO	Standard error value, stderr. It has the value of 2.

A constant null designates a null pointer.

An integer-constant EOF (–1) is returned upon end-of-file or error by most integer functions that deal with streams (see the individual descriptions for details).

An integer constant BUFSIZ specifies the size of the buffers used by the particular implementation.

An integer constant FILENAME_MAX specifies the size needed for an array of char large enough to hold the longest file name string that the implementation guarantees can be opened.

An integer constant FOPEN_MAX specifies the minimum number of files that the implementation guarantees can be open simultaneously. Note that no more than 255 files may be opened via fopen, and only file descriptors 0 through 255 are valid.

Any program that uses this package must include the header file of pertinent macro definitions, as follows:

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

The functions and constants for all 3S entries in this manual are declared in the header file and need no further declaration. The constants and the following "functions" are implemented as macros. Don't redeclare these names: getc, getchar, putc, putchar, ferror, feof, clearerr, and fileno.

There are also function versions of getc, getchar, putc, putchar, ferror, feof, clearerr, and fileno.

Output streams, with the exception of the standard error stream `stderr`, are by default buffered if the output refers to a file and line-buffered if the output refers to a terminal. The standard error output stream `stderr` is by default unbuffered, but use of `freopen` [see `fopen`(3S)] will cause it to become buffered or line-buffered. When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as written; when it is buffered, many characters are saved up and written as a block. When it is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new-line character is written or terminal input is requested). `setbuf` or `setvbuf` [both described in `setbuf`(3S)] may be used to change the stream's buffering strategy.

SEE ALSO

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

DIAGNOSTICS

Invalid *stream* pointers will usually cause problems, possibly including program termination. The description for each function lists its possible error conditions.

NAME

stdipc: ftok – standard interprocess communication package

SYNOPSIS

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

key_t ftok(const char *path, int 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. It is still possible to interface intentionally. 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 that uniquely identifies a project. Note that ftok will return the same key for linked files when called with the same *id* and that it will return different keys when called with the same file name but different *ids*.

SEE ALSO

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

DIAGNOSTICS

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

NOTES

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

str: strfind, strrspn, strtrns – string manipulations

SYNOPSIS

cc [*flag* ...] *file* ... -lgen [*library* ...]

```
#include <libgen.h>

int strfind (const char *as1, const char *as2);

char *strrspn (const char *string, const char *tc);

char * strtrns (const char *str, const char *old, const char *new,
    char *result);
```

DESCRIPTION

strfind returns the offset of the second string, *as2*, if it is a substring of string *as1*.

strrspn returns a pointer to the first character in the string to be trimmed (all characters from the first character to the end of *string* are in *tc*).

strtrns transforms str and copies it into *result*. Any character that appears in *old* is replaced with the character in the same position in *new*. The *new* result is returned.

EXAMPLES

```
/* find pointer to substring "hello" in as1 */
i = strfind(as1, "hello");

/* trim junk from end of string */
s2 = strrspn(s1, "*?#$%");
*s2 = '\0';

/* transform lower case to upper case */
a1[] = "abcdefghijklmnopqrstuvwxyz";
a2[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
s2 = strtrns(s1, a1, a2, s2);
```

SEE ALSO

string(3C).

DIAGNOSTICS

If the second string is not a substring of the first string strfind returns –1.

NAME

strccpy: streadd, strcadd, strecpy – copy strings, compressing or expanding escape codes

SYNOPSIS

cc [*flag* ...] *file* ... -lgen [*library* ...]

```
#include <libgen.h>

char *strccpy (char *output, const char *input);

char *strcadd (char *output, const char *input);

char *strecpy (char *output, const char *input, const char
    *exceptions);

char *streadd (char *output, const char *input, const char
    *exceptions);
```

DESCRIPTION

strccpy copies the *input* string, up to a null byte, to the *output* string, compressing the C-language escape sequences (for example, \n, \001) to the equivalent character. A null byte is appended to the output. The *output* argument must point to a space big enough to accommodate the result. If it is as big as the space pointed to by *input* it is guaranteed to be big enough. strccpy returns the *output* argument.

strcadd is identical to strccpy, except that it returns the pointer to the null byte that terminates the output.

strecpy copies the *input* string, up to a null byte, to the *output* string, expanding non-graphic characters to their equivalent C-language escape sequences (for example, \n, \001). The *output* argument must point to a space big enough to accommodate the result; four times the space pointed to by *input* is guaranteed to be big enough (each character could become \ and 3 digits). Characters in the *exceptions* string are not expanded. The *exceptions* argument may be zero, meaning all non-graphic characters are expanded. strecpy returns the *output* argument

streadd is identical to strecpy, except that it returns the pointer to the null byte that terminates the output.

EXAMPLES

```
/* expand all but newline and tab */
strecpy( output, input, "\n\t" );

/* concatenate and compress several strings */
cp = strcadd( output, input1 );
cp = strcadd( cp, input2 );
cp = strcadd( cp, input3 );
```

SEE ALSO

string(3C), str(3G).

NAME

strcoll – string collation

SYNOPSIS

```
#include <string.h>

int strcoll (const char *s1, const char *s2);
```

DESCRIPTION

strcoll returns an integer greater than, equal to, or less than zero in direct correlation to whether string *s1* is greater than, equal to, or less than the string *s2*. The comparison is based on strings interpreted as appropriate to the program's locale for category LC_COLLATE [see setlocale(3C)].

Both strcoll and strxfrm provide for locale-specific string sorting. strcoll is intended for applications in which the number of comparisons per string is small. When strings are to be compared a number of times, strxfrm is a more appropriate utility because the transformation process occurs only once.

FILES

/usr/lib/*locale*/LC_COLLATE	LC_COLLATE database for *locale*.

SEE ALSO

setlocale(3C), string(3C), strxfrm(3C), environ(5).
colltbl(1M) in the *System Administrator's Reference Manual*.

NAME

strerror – get error message string

SYNOPSIS

```
#include <string.h>

char *strerror (int errnum);
```

DESCRIPTION

strerror maps the error number in *errnum* to an error message string, and returns a pointer to that string. strerror uses the same set of error messages as perror. The returned string should not be overwritten.

SEE ALSO

perror(3C).

EXAMPLE

The following program prints an error message corresponding to the generic UNIX I/O message.

```
/* Program test for the strerror() function */

#include <errno.h>
#include <stdio.h>

char *strerror();

main() {
    printf("Error message is: %s\n", strerror(EIO));
    return 0;
}
```

A call to this program generates the output

```
Error message is: Error EIO(5) – Error in input/ouput.
```

FILES

/usr/lib/locale/*locale*/LC_MESSAGES/uxsyserr — message catalog.

SEE ALSO

perror(3C).

NAME

strftime, cftime, ascftime – convert date and time to string

SYNOPSIS

```
#include <time.h>

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

int cftime (char *s, char *format, const time_t *clock);

int ascftime (char *s, const char *format, const struct tm
    *timeptr);
```

DESCRIPTION

strftime, ascftime, and cftime place characters into the array pointed to by s as controlled by the string pointed to by *format*. The *format* string consists of zero or more directives and ordinary characters. All ordinary characters (including the terminating null character) are copied unchanged into the array. For strftime, no more than *maxsize* characters are placed into the array.

If *format* is (char *)0, then the locale's default format is used. For strftime the default format is the same as "%c", for cftime and ascftime the default format is the same as "%C". cftime and ascftime first try to use the value of the environment variable CFTIME, and if that is undefined or empty, the default format is used.

Each directive is replaced by appropriate characters as described in the following list. The appropriate characters are determined by the LC_TIME category of the program's locale and by the values contained in the structure pointed to by *timeptr* for strftime and ascftime, and by the time represented by *clock* for cftime.

%%	same as %
%a	locale's abbreviated weekday name
%A	locale's full weekday name
%b	locale's abbreviated month name
%B	locale's full month name
%c	locale's appropriate date and time representation
%C	locale's date and time representation as produced by date(1)
%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	locale's abbreviated month name.
%H	hour (00 - 23)
%I	hour (01 - 12)
%j	day number of year (001 - 366)
%m	month number (01 - 12)
%M	minute (00 - 59)
%n	same as \n
%p	locale's equivalent of either AM or PM
%r	time as %I:%M:%S [AM|PM]
%R	time as %H:%M
%S	seconds (00 - 61), allows for leap seconds
%t	insert a tab
%T	time as %H:%M:%S
%U	week number of year (00 - 53), Sunday is the first day of week 1

%w	weekday number (0 - 6), Sunday = 0
%W	week number of year (00 - 53), Monday is the first day of week 1
%x	locale's appropriate date representation
%X	locale's appropriate time representation
%y	year within century (00 - 99)
%Y	year as ccyy (e.g. 1986)
%Z	time zone name or no characters if no time zone exists

The difference between `%U` and `%W` lies in which day is counted as the first of the week. Week number 01 is the first week in January starting with a Sunday for `%U` or a Monday for `%W`. Week number 00 contains those days before the first Sunday or Monday in January for `%U` and `%W`, respectively.

If the total number of resulting characters including the terminating null character is not more than *maxsize*, `strftime`, `cftime` and `ascftime` return the number of characters placed into the array pointed to by *s* not including the terminating null character. Otherwise, zero is returned and the contents of the array are indeterminate. `cftime` and `ascftime` return the number of characters placed into the array pointed to by *s* not including the terminating null character.

Selecting the Output's Language

By default, the output of `strftime`, `cftime`, and `ascftime` appear in the language defined by the "C" locale. This is basically English as spoken in the United States, but technically is different from the "En_US" locale. The user can request that the output of `strftime`, `cftime` or `ascftime` be in a specific language by setting the *locale* for *category* `LC_TIME` or `LC_ALL` with `setlocale`.

Timezone

The timezone is taken from the environment variable `TZ` [see `ctime(3C)` and `environ(5)` for a description of `TZ`].

EXAMPLES

The example illustrates the use of `strftime`. It shows what the string in `str` would look like if the structure pointed to by *tmptr* contains the values corresponding to Thursday, August 28, 1986 at 12:44:36 in New Jersey.

```
strftime (str, strsize, "%A %b %d %j", tmptr)
```

This results in `str` containing "Thursday Aug 28 240".

FILES

`/usr/lib/locale/`*language*`/LC_TIME` – file containing locale specific date and time information

SEE ALSO

`ctime(3C)`, `getenv(3C)`, `setlocale(3C)`, `strftime(4)`, `timezone(4)`, `environ(5)`.

NOTE

`cftime` and `ascftime` are obsolete. `strftime` should be used instead.

NAME

string: strcat, strdup, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok, strstr – string operations

SYNOPSIS

```
#include <string.h>

char *strcat (char *s1, const char *s2);

char *strdup (const char *s1);

char *strncat (char *s1, const char *s2, size_t n);

int strcmp (const char *s1, const char *s2);

int strncmp (const char *s1, const char *s2, size_t n);

char *strcpy (char *s1, const char *s2);

char *strncpy (char *s1, const char *s2, size_t n);

size_t strlen (const char *s);

char *strchr (const char *s, int c);

char *strrchr (const char *s, int c);

char *strpbrk (const char *s1, const char *s2);

size_t strspn (const char *s1, const char *s2);

size_t strcspn (const char *s1, const char *s2);

char *strtok (char *s1, const char *s2);

char *strstr (const char *s1, const char *s2);
```

DESCRIPTION

The arguments *s*, *s1*, and *s2* point to strings (arrays of characters terminated by a null character). The functions strcat, strncat, strcpy, strncpy, and strtok. all alter *s1*. These functions do not check for overflow of the array pointed to by *s1*.

strcat appends a copy of string *s2*, including the terminating null character, to the end of string *s1*. strncat appends at most *n* characters. Each returns a pointer to the null-terminated result. The initial character of *s2* overrides the null character at the end of *s1*.

strcmp compares its arguments and returns an integer less than, equal to, or greater than 0, based upon whether *s1* is lexicographically less than, equal to, or greater than *s2*. strncmp makes the same comparison but looks at at most *n* characters. Characters following a null character are not compared.

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

strdup returns a pointer to a new string which is a duplicate of the string pointed to by *s1*. The space for the new string is obtained using malloc(3C). If the new string can not be created, a NULL pointer is returned.

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

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

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

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

strtok considers the string *s1* to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *s2*. The first call (with pointer *s1* specified) returns a pointer to the first character of the first token, and will have written a null character into *s1* immediately following the returned token. The function keeps track of its position in the string between separate calls, so that subsequent calls (which must be made with the first argument a NULL pointer) 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.

strstr locates the first occurrence in string *s1* of the sequence of characters (excluding the terminating null character) in string *s2*. strstr returns a pointer to the located string, or a null pointer if the string is not found. If *s2* points to a string with zero length (i.e., the string ""), the function returns *s1*.

SEE ALSO

ffs(3C), malloc(3C), setlocale(3C), str(3G), strxfrm(3C).

NOTES

All of these functions assume the default locale "C." For some locales, strxfrm should be applied to the strings before they are passed to the functions.

NAME

`strsave`, `strnsave` – allocate area large enough to hold string and move string into it

SYNOPSIS

```
char *string, *newstring, *strsave(const char *);
...
newstring = strsave(string);

char *string, *newstring, *strnsave(const char *, int);
int n;
...
newstring = strnsave(string, n);
```

string	A byte pointer to a character string.
newstring	A byte pointer to the area allocated to receive a copy of the string.
n	The maximum number of characters to copy.

DESCRIPTION

The `strsave` function allocates with `malloc` an area large enough to hold a specified string and moves a copy of the string into the area. It then returns a pointer to the area.

The `strsave` and `strnsave` functions are the same except that with `strnsave` you specify a maximum number of bytes to copy. The include files `string.h` and `strings.h` define these functions.

Return Value

Each function returns a pointer to the allocated area. Each returns a null if it cannot allocate the area.

EXAMPLES

```
/* Program test1 for the strsave() function */

#include <string.h>
#include <stdio.h>

char *newloc, *strsave(const char *);
int i = 1;

main(argc, argv)
int argc;
char *argv[];
{
    while (i < argc) {
         newloc = strsave(argv[i]);
         printf("\tStored argv[%d] at %o.\n", i, newloc);
         printf("argv[%d] = `%s'\n", i, newloc);
         i++;
    }
    return 0;
}
```

A call to the program `test1` with the arguments

```
Save these strings.
```

generates the output

```
          Stored argv[1] at 34003540370.
argv[1] = `Save'
          Stored argv[2] at 34003540410.
argv[2] = `these'
          Stored argv[3] at 34003540430.
argv[3] = `strings.'
```

The locations vary with execution.

```
/* Program test2 for the strnsave() function */

#include <string.h>
#include <stdio.h>
#define MAX     80

char *newloc, *strnsave(const char *, int);
int i = 1;

main(argc, argv)
int argc;
char *argv[];
{
    while (i < argc) {
        newloc = strnsave(argv[i], MAX);
        printf("Stored argv[%d] at %o.0, i, newloc);
        printf("argv[%d] = `%s'0, i, newloc);
        i++;
    }
    return 0;
}
```

A call to the program `test2` with the arguments

```
Find some addresses.
```

generates the output

```
          Stored argv[1] at 34003703644.
argv[1] = `Find'
          Stored argv[2] at 34003677500.
argv[2] = `some'
          Stored argv[3] at 34003677344.
argv[3] = `addresses.'
```

The locations vary with execution.

SEE ALSO

malloc(3C), strdup(3C).

NAME

strtod, atof, – convert string to double-precision number

SYNOPSIS

```
#include <stdlib.h>

double strtod (const char *nptr, char **endptr);

double atof (const char *nptr);
```

DESCRIPTION

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

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

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

atof(nptr) is equivalent to:

```
strtod(nptr, (char **)NULL).
```

SEE ALSO

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

DIAGNOSTICS

If the correct value would cause overflow, ±HUGE is returned (according to the sign of the value), and *errno* is set to ERANGE.
If the correct value would cause underflow, zero is returned and *errno* is set to ERANGE.
When the -Xc or -Xa compilation options are used, HUGE_VAL is returned instead of HUGE.

NAME

strtol, strtoul, atol, atoi – convert string to integer

SYNOPSIS

```
#include <stdlib.h>

long strtol (const char *str, char **ptr, int base);

unsigned long strtoul (const char *str, char **ptr, int base);

long atol (const char *str);

int atoi (const char *str);
```

DESCRIPTION

strtol returns as a long integer the value represented by the character string pointed to by str. The string is scanned up to the first character inconsistent with the base. Leading "white-space" characters [as defined by isspace in ctype(3C)] are ignored.

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

If *base* is positive (and not greater than 36), it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and "0x" or "0X" is ignored if *base* is 16.

If *base* is zero, the string itself determines the base as follows: 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.

If the value represented by *str* would cause overflow, LONG_MAX or LONG_MIN is returned (according to the sign of the value), and errno is set to the value, ERANGE.

strtoul is similar to strtol except that strtoul returns as an unsigned long integer the value represented by *str*. If the value represented by *str* would cause overflow, ULONG_MAX is returned, and errno is set to the value, ERANGE.

Except for behavior on error, atol(str) is equivalent to: strtol(str, (char **)NULL, 10).

Except for behavior on error, atoi(str) is equivalent to: (int) strtol(str, (char **)NULL, 10).

DIAGNOSTICS

If strtol is given a *base* greater than 36, it returns 0 and sets errno to EINVAL.

SEE ALSO

ctype(3C), scanf(3S), strtod(3C).

NOTES

strtol no longer accepts values greater than LONG_MAX as valid input. Use strtoul instead.

NAME

strxfrm – string transformation

SYNOPSIS

```
#include <string.h>

size_t strxfrm (char *s1, const char *s2, size_t n);
```

DESCRIPTION

strxfrm transforms the string *s2* and places the resulting string into the array *s1*. The transformation is such that if strcmp is applied to two transformed strings, it will return the same result as strcoll applied to the same two original strings. The transformation is based on the program's locale for category LC_COLLATE [see setlocale(3C)].

No more than *n* bytes will be placed into the resulting array pointed to by *s1*, including the terminating null byte. If *n* is 0, then *s1* is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

strxfrm returns the length of the transformed string (not including the terminating null byte). If the value returned is *n* or more, the contents of the array *s1* are indeterminate.

EXAMPLE

The value of the following expression is the size of the array needed to hold the transformation of the string pointed to by *s*.

```
1 + strxfrm(NULL, s, 0);
```

FILES

/usr/lib/locale/*locale*/LC_COLLATE
LC_COLLATE database for *locale*.

SEE ALSO

colltbl(1M) in the *System Administrator's Reference Manual*.
setlocale(3C), strcoll(3C), string(3C), environ(5).

DIAGNOSTICS

On failure, strxfrm returns (size_t) -1.

NAME

swab – swap bytes

SYNOPSIS

```
#include <stdlib.h>

void swab (const char *from, char *to, int nbytes);
```

DESCRIPTION

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

SEE ALSO

bcopy(3C).

NAME

swapcontext – manipulate user contexts

SYNOPSIS

```
#include <ucontext.h>

int swapcontext (ucontext_t *oucp, ucontext_t *ucp);
```

DESCRIPTION

This function is useful for implementing user-level context switching between multiple threads of control within a process.

swapcontext saves the current context in the context structure pointed to by *oucp* and sets the context to the context structure pointed to by *ucp*.

This function will fail if either of the following is true:

ENOMEM *ucp* does not have enough stack left to complete the operation.

EFAULT *ucp* or *oucp* points to an invalid address.

SEE ALSO

exit(2), getcontext(2), sigaction(2), sigprocmask(2), ucontext(5).

DIAGNOSTICS

On successful completion, swapcontext return a value of zero. Otherwise, a value of −1 is returned and errno is set to indicate the error.

NOTES

The size of the ucontext_t structure may change in future releases. To remain binary compatible, users must always use getcontext to create new instances of them.

NAME

syslog, openlog, closelog, setlogmask – control system log

SYNOPSIS

```
#include <syslog.h>

openlog(ident, logopt, facility)
char *ident;

syslog(level, message, parameters ... )
char *message;

closelog()

setlogmask(maskpri)
```

DESCRIPTION

syslog arranges to write *message* onto the system log maintained by syslogd(1M). The message is tagged with a priority *level*. The message looks like a printf(3C) string except that %m is replaced by the current error message (collected from errno). A trailing new-line is added if needed. This message will be read by syslogd(1M) and written to the system console, log files, or forwarded to syslogd on another host as appropriate.

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 should possibly be handled specially.

LOG_INFO Informational messages.

LOG_DEBUG Messages that contain information normally of use only when debugging a program.

If syslog cannot pass the message to syslogd, it will attempt to write the message on /dev/console if the LOG_CONS option is set (see below).

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: useful for identifying instantiations of servers (daemons).

LOG_CONS Force writing messages to the console if unable to send it to syslogd. This option is safe to use in server processes that have no controlling terminal since syslog will fork before opening the console.

LOG_NDELAY Open the connection to syslogd immediately. Normally the open is delayed until the first message is logged. Useful for programs that need to manage the order in which file descriptors are allocated.

LOG_NOWAIT Do not wait for child processes forked to log messages on the console. This option should be used by processes that enable notification of child termination via SIGCHLD, as `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 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 servers, such as `ftpd`(1M), etc.

LOG_AUTH The authorization system: `login`(1), `su`(1), etc.

LOG_LPR The line printer spooling system: `lp`(1), etc.

LOG_LOCAL0 Reserved for local use. Similarly for LOG_LOCAL1 through LOG_LOCAL7.

`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

```
syslog(LOG_ALERT, "who: internal error 23");

openlog("ftpd", LOG_PID, LOG_DAEMON);
setlogmask(LOG_UPTO(LOG_ERR));
syslog(LOG_INFO, "Connection from host %d", CallingHost);

syslog(LOG_INFO|LOG_LOCAL2, "foobar error: %m");
```

SEE ALSO

`logger`(1), `syslogd`(1M), `syslog.conf`(5).

NAME

system – issue a shell command

SYNOPSIS

```
#include <stdlib.h>

int system (const char *string);
```

DESCRIPTION

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

If *string* is a NULL pointer, system checks if /sbin/sh exists and is executable. If /sbin/sh is available, system returns non-zero; otherwise it returns zero.

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

EAGAIN The system-imposed limit on the total number of processes under execution by a single user would be exceeded.

EINTR system was interupted by a signal.

ENOMEM The new process requires more memory than is allowed by the system-imposed maximum MAXMEM.

SEE ALSO

exec(2), waitpid(3C).
sh(1) in the *User's Reference Manual*.

DIAGNOSTICS

system forks to create a child process that in turn calls exec to execute /sbin/sh. This shell then executes *string*. If the fork or exec fails, system returns a value of -1 and sets errno.

NAME

sysv3_cuserid – get character login name of the user

SYNOPSIS

```
#include <stdio.h>

char *sysv3_cuserid (s)
char *s;
```

DESCRIPTION

The function sysv3_cuserid generates a character-string representation for the login name of the owner of the current process. If *s* is a NULL pointer, this representation is generated in an internal static area, the address of which is returned. Otherwise, *s* is assumed to point to an array of at least L_cuserid characters; the representation is left in this array. The constant L_cuserid is defined in the <stdio.h> header file.

DIAGNOSTICS

If the login name cannot be found, sysv3_cuserid returns a NULL pointer; if *s* is not a NULL pointer, a null character (\0) will be placed at *s[0]*.

SEE ALSO

cuserid(3S), getlogin(3C), getpwent(3C).

NAME

t_accept – accept a connect request

SYNOPSIS

```
#include <tiuser.h>

int t_accept ( int fd, int resfd, struct t_call *call);
```

DESCRIPTION

This function is issued by a transport user to accept a connect request. fd identifies the local transport endpoint where the connect indication arrived, resfd specifies the local transport endpoint where the connection is to be established, and call contains information required by the transport provider to complete the connection. call points to a t_call structure that contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

netbuf is described in intro(3). In call, addr is the address of the caller, opt indicates any protocol-specific parameters associated with the connection, udata points to any user data to be returned to the caller, and sequence is the value returned by t_listen that uniquely associates the response with a previously received connect indication.

A transport user may accept a connection on either the same, or on a different, local transport endpoint from the one on which the connect indication arrived. If the same endpoint is specified (i.e., resfd=fd), the connection can be accepted unless the following condition is true: The user has received other indications on that endpoint but has not responded to them (with t_accept or t_snddis). For this condition, t_accept 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 (e.g., a connect or disconnect) waiting to be received on that endpoint.

The values of parameters specified by opt and the syntax of those values are protocol specific. The udata argument enables the called transport user to send user data to the caller and the amount of user data must not exceed the limits supported by the transport provider as returned in the connect field of the info argument of t_open or t_getinfo. If the len [see netbuf in intro(3)] field of udata is zero, no data will be sent to the caller.

On failure, t_errno may be set to one of the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint, or the user is illegally accepting a connection on the same transport endpoint on which the connect indication arrived.

[TOUTSTATE] The function was issued in the wrong sequence on the transport endpoint referenced by fd, or the transport endpoint referred to by resfd is not in the T_IDLE state.

[TACCES] The user does not have permission to accept a connection on the responding transport endpoint or use the specified options.

[TBADOPT] The specified options were in an incorrect format or contained illegal information.

[TBADDATA] The amount of user data specified was not within the bounds allowed by the transport provider.

[TBADSEQ] An invalid sequence number was specified.

[TLOOK] An asynchronous event has occurred on the transport endpoint referenced by fd and requires immediate attention.

[TNOTSUPPORT] This function is not supported by the underlying transport provider.

[TSYSERR] A system error has occurred during execution of this function.

SEE ALSO

intro(3), t_connect(3N), t_getstate(3N), t_listen(3N), t_open(3N), t_rcvconnect(3N).

UNIX System V Network Programmer's Guide.

DIAGNOSTICS

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

NAME

t_alloc – allocate a library structure

SYNOPSIS

```
#include <tiuser.h>

char *t_alloc(fd, struct_type, fields)
int fd;
int struct_type;
int fields;
```

DESCRIPTION

The t_alloc function dynamically allocates memory for the various transport function argument structures as specified below. This function will allocate memory for the specified structure, and will also allocate memory for buffers referenced by the structure.

The structure to allocate is specified by struct_type, and can be one of the following:

T_BIND	struct t_bind
T_CALL	struct t_call
T_OPTMGMT	struct t_optmgmt
T_DIS	struct t_discon
T_UNITDATA	struct t_unitdata
T_UDERROR	struct t_uderr
T_INFO	struct t_info

where each of these structures may subsequently be used as an argument to one or more transport functions.

Each of the above structures, except T_INFO, contains at least one field of type struct netbuf. netbuf is described in intro(3). For each field of this type, the user 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 will allocate memory for the buffer associated with the field, and initialize the buf pointer and maxlen [see netbuf in intro(3) for description of buf and maxlen] field accordingly. The length of the buffer allocated will be based on the same size information that is returned to the user on t_open and t_getinfo. Thus, fd must refer to the transport endpoint through which the newly allocated structure will be passed, so that the appropriate size information can be accessed. If the size value associated with any specified field is –1 or –2 (see t_open or t_getinfo), t_alloc will be unable to determine the size of the buffer to allocate and will fail, setting t_errno to TSYSERR and errno to EINVAL. For any field not specified in fields, buf will be set to NULL and maxlen will be set to zero.

Use of t_alloc to allocate structures will help ensure the compatibility of user programs with future releases of the transport interface.

On failure, t_errno may be set to one of the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

[TSYSERR] A system error has occurred during execution of this function.

SEE ALSO

intro(3), t_free(3N), t_getinfo(3N), t_open(3N).

UNIX System V Network Programmer's Guide.

DIAGNOSTICS

On successful completion, t_alloc returns a pointer to the newly allocated structure. On failure, NULL is returned.

NAME

t_bind – bind an address to a transport endpoint

SYNOPSIS

```
#include <tiuser.h>

int t_bind (fd, req, ret)
int fd;
struct t_bind *req;
struct t_bind *ret;
```

DESCRIPTION

This function associates a protocol address with the transport endpoint specified by fd and activates that transport endpoint. In connection mode, the transport provider may begin accepting or requesting connections on the transport endpoint. In connectionless mode, the transport user may send or receive data units through the transport endpoint.

The req and ret arguments point to a t_bind structure containing the following members:

```
struct netbuf addr;
unsigned qlen;
```

netbuf is described in intro(3). The addr field of the t_bind structure specifies a protocol address and the qlen field is used to indicate the maximum number of outstanding connect indications.

req is used to request that an address, represented by the netbuf structure, be bound to the given transport endpoint. len [see netbuf in intro(3); also for buf and maxlen] specifies the number of bytes in the address and buf points to the address buffer. maxlen has no meaning for the req argument. On return, ret contains the address that the transport provider actually bound to the transport endpoint; this may be different from the address specified by the user in req. In ret, the user specifies maxlen, which is the maximum size of the address buffer, and buf, which points to the buffer where the address is to be placed. On return, len specifies the number of bytes in the bound address and buf points to the bound address. If maxlen is not large enough to hold the returned address, an error will result.

If the requested address is not available, or if no address is specified in req (the len field of addr in req is zero) the transport provider may 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 zero, and the transport provider must assign an address to the transport endpoint. Similarly, ret may be NULL if the user does not care what address was bound by the provider and is not interested in the negotiated value of qlen. It is valid to set req and ret to NULL for the same call, in which case the provider chooses the address to bind to the transport endpoint and does not return that information to the user.

The qlen field has meaning only when initializing a connection-mode service. It specifies the number of outstanding connect indications the transport provider should support for the given transport endpoint. An outstanding connect indication is one that has been passed to the transport user by the transport provider. A value of qlen greater than zero is only meaningful when issued by a passive transport user that expects other users to call it. The value of qlen 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.

This function allows more than one transport endpoint to be bound to the same protocol address (however, the transport provider must support this capability also), but it is not allowable to bind more than one protocol address to the same transport endpoint. If a user binds more than one transport endpoint to the same protocol address, only one endpoint can be used to listen for connect indications associated with that protocol address. In other words, only one t_bind for a given protocol address may specify a value of qlen greater than zero. In this way, the transport provider can identify which transport endpoint should be notified of an incoming connect indication. If a user attempts to bind a protocol address to a second transport endpoint with a value of qlen greater than zero, the transport provider will assign another address to be bound to that endpoint. If a user accepts a connection on the transport endpoint that is being used as the listening endpoint, the bound protocol address will be found to be busy for the duration of that connection. No other transport endpoints may be bound for listening while that initial listening endpoint is in the data transfer phase. This will prevent more than one transport endpoint bound to the same protocol address from accepting connect indications.

On failure, t_errno may be set to one of the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

[TOUTSTATE] The function was issued in the wrong sequence.

[TBADADDR] The specified protocol address was in an incorrect format or contained illegal information.

[TNOADDR] The transport provider could not allocate an address.

[TACCES] The user does not have permission to use the specified address.

[TBUFOVFLW] The number of bytes allowed for an incoming argument is not sufficient to store the value of that argument. The provider's state will change to T_IDLE and the information to be returned in ret will be discarded.

[TSYSERR] A system error has occurred during execution of this function.

SEE ALSO

intro(3), t_open(3N), t_optmgmt(3N), t_unbind(3N).

UNIX System V Network Programmer's Guide.

DIAGNOSTICS

t_bind returns 0 on success and −1 on failure and t_errno is set to indicate the error.

NAME

t_close – close a transport endpoint

SYNOPSIS

```
#include <tiuser.h>

int t_close(fd)
int fd;
```

DESCRIPTION

The t_close function informs the transport provider that the user is finished with the transport endpoint specified by fd, and frees any local library resources associated with the endpoint. In addition, t_close closes the file associated with the transport endpoint.

t_close should be called from the T_UNBND state [see t_getstate(3N)]. However, this function does not check state information, so it may be called from any state to close a transport endpoint. If this occurs, the local library resources associated with the endpoint will be freed automatically. In addition, close(2) will be issued for that file descriptor; the close will be abortive if no other process has that file open, and will break any transport connection that may be associated with that endpoint.

On failure, t_errno may be set to the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

SEE ALSO

t_getstate(3N), t_open(3N), t_unbind(3N).

UNIX System V Network Programmer's Guide.

DIAGNOSTICS

t_close returns 0 on success and −1 on failure and t_errno is set to indicate the error.

NAME

t_connect – establish a connection with another transport user

SYNOPSIS

```
#include <tiuser.h>

int t_connect(fd, sndcall, rcvcall)
int fd;
struct t_call *sndcall;
struct t_call *rcvcall;
```

DESCRIPTION

This function enables a transport user to request a connection to the specified destination transport user. fd identifies the local transport endpoint where communication will be established, while sndcall and rcvcall point to a t_call structure that contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

sndcall specifies information needed by the transport provider to establish a connection and rcvcall specifies information that is associated with the newly established connection.

netbuf is described in intro(3). In sndcall, addr specifies the protocol address of the destination transport user, opt presents any protocol-specific information that might be needed by the transport provider, udata points to optional user data that may be passed to the destination transport user during connection establishment, and sequence has no meaning for this function.

On return in rcvcall, addr returns the protocol address associated with the responding transport endpoint, opt presents any protocol-specific information associated with the connection, udata points to optional user data that may be returned by the destination transport user during connection establishment, and sequence has no meaning for this function.

The opt argument implies no structure on the options that may be passed to the transport provider. The transport provider is free to specify the structure of any options passed to it. These options are specific to the underlying protocol of the transport provider. The user may choose not to negotiate protocol options by setting the len field of opt to zero. In this case, the provider may use default options.

The udata argument enables the caller to pass user data to the destination transport user and receive user data from the destination user during connection establishment. However, the amount of user data must not exceed the limits supported by the transport provider as returned in the connect field of the info argument of t_open(3N) or t_getinfo(3N). If the len [see netbuf in intro(3)] field of udata is zero in sndcall, no data will be sent to the destination transport user.

On return, the addr, opt, and udata fields of rcvcall will be updated to reflect values associated with the connection. Thus, the maxlen [see netbuf in intro(3)] field of each argument must be set before issuing this function to indicate the maximum size of the buffer for each. However, rcvcall may be NULL, in which case no information is given to the user on return from t_connect.

By default, t_connect executes in synchronous mode, and will wait for the destination user's response before returning control to the local user. A successful return (i.e., return value of zero) indicates that the requested connection has been established. However, if O_NDELAY or O_NONBLOCK is set (via t_open or fcntl), t_connect executes in asynchronous mode. In this case, the call will not wait for the remote user's response, but will return control immediately to the local user and return -1 with t_errno set to TNODATA to indicate that the connection has not yet been established. In this way, the function simply initiates the connection establishment procedure by sending a connect request to the destination transport user.

On failure, t_errno may be set to one of the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

[TOUTSTATE] The function was issued in the wrong sequence.

[TNODATA] O_NDELAY or O_NONBLOCK was set, so the function successfully initiated the connection establishment procedure, but did not wait for a response from the remote user.

[TBADADDR] The specified protocol address was in an incorrect format or contained illegal information.

[TBADOPT] The specified protocol options were in an incorrect format or contained illegal information.

[TBADDATA] The amount of user data specified was not within the bounds allowed by the transport provider.

[TACCES] The user does not have permission to use the specified address or options.

[TBUFOVFLW] The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. If executed in synchronous mode, the provider's state, as seen by the user, changes to T_DATAXFER, and the connect indication information to be returned in rcvcall is discarded.

[TLOOK] An asynchronous event has occurred on this transport endpoint and requires immediate attention.

[TNOTSUPPORT] This function is not supported by the underlying transport provider.

[TSYSERR] A system error has occurred during execution of this function.

SEE ALSO

intro(3), t_accept(3N), t_getinfo(3N), t_listen(3N), t_open(3N), t_optmgmt(3N), t_rcvconnect(3N).

UNIX System V Network Programmer's Guide.

DIAGNOSTICS

t_connect returns 0 on success and −1 on failure and t_errno is set to indicate the error.

NAME

t_error – produce error message

SYNOPSIS

```
#include <tiuser.h>

void t_error(errmsg)
char *errmsg;
extern int t_errno;
extern char *t_errlist[];
extern int t_nerr;
```

DESCRIPTION

t_error produces a message on the standard error output which describes the last error encountered during a call to a transport function. The argument string errmsg is a user-supplied error message that gives context to the error.

t_error prints the user-supplied error message followed by a colon and the standard transport function error message for the current value contained in t_errno. If t_errno is TSYSERR, t_error will also print the standard error message for the current value contained in errno [see intro(2)].

t_errlist is the array of message strings, to allow user message formatting. t_errno can be used as an index into this array to retrieve the error message string (without a terminating newline). t_nerr is the maximum index value for the t_errlist array.

t_errno is set when an error occurs and is not cleared on subsequent successful calls.

EXAMPLE

If a t_connect function fails on transport endpoint fd2 because a bad address was given, the following call might follow the failure:

t_error("t_connect failed on fd2");

The diagnostic message would print as:

t_connect failed on fd2: Incorrect transport address format

where "t_connect failed on fd2" tells the user which function failed on which transport endpoint, and "Incorrect transport address format" identifies the specific error that occurred.

SEE ALSO

t_alloc(3N).
UNIX System V Network Programmer's Guide.

NAME

t_free – free a library structure

SYNOPSIS

```
#include <tiuser.h>

int t_free(ptr, struct_type)
char *ptr;
int struct_type;
```

DESCRIPTION

The t_free function frees memory previously allocated by t_alloc. This function will free memory for the specified structure, and will also free memory for buffers referenced by the structure.

ptr points to one of the six structure types described for t_alloc, and struct_type identifies the type of that structure, which can be one of the following:

T_BIND	struct t_bind
T_CALL	struct t_call
T_OPTMGMT	struct t_optmgmt
T_DIS	struct t_discon
T_UNITDATA	struct t_unitdata
T_UDERROR	struct t_uderr
T_INFO	struct t_info

where each of these structures is used as an argument to one or more transport functions.

t_free will check the addr, opt, and udata fields of the given structure (as appropriate), and free the buffers pointed to by the buf field of the netbuf [see intro(3)] structure. 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.

On failure, t_errno may be set to the following:

[TSYSERR] A system error has occurred during execution of this function.

SEE ALSO

intro(3), t_alloc(3N).

UNIX System V Network Programmer's Guide.

DIAGNOSTICS

t_free returns 0 on success and –1 on failure and t_errno is set to indicate the error.

NAME

t_getinfo – get protocol-specific service information

SYNOPSIS

```
#include <tiuser.h>

int t_getinfo(fd, info)
int fd;
struct t_info *info;
```

DESCRIPTION

This function returns the current characteristics of the underlying transport protocol associated with file descriptor fd. The info structure is used to return the same information returned by t_open. This function enables a transport user to access this information during any phase of communication.

This argument points to a t_info structure, which contains the following members:

```
long addr;     /* max size of the transport protocol address */
long options;/* max number of bytes of protocol-specific options */
long tsdu;     /* max size of a transport service data unit (TSDU) */
long etsdu;    /* max size of an expedited transport service data unit (ETSDU) */
long connect;/* max amount of data allowed on connection establishment functions */
long discon;  /* max amount of data allowed on t_snddis and t_rcvdis functions */
long servtype;/* service type supported by the transport provider */
```

The values of the fields have the following meanings:

addr — A value greater than or equal to zero indicates the maximum size of a transport protocol address; a value of −1 specifies that there is no limit on the address size; and a value of −2 specifies that the transport provider does not provide user access to transport protocol addresses.

options — A value greater than or equal to zero indicates the maximum number of bytes of protocol-specific options supported by the provider; a value of −1 specifies that there is no limit on the option size; and a value of −2 specifies that the transport provider does not support user-settable options.

tsdu — A value greater than zero specifies the maximum size of a transport service data unit (TSDU); a value of zero specifies that the transport provider does not support the concept of TSDU, although it does support the sending of a data stream with no logical boundaries preserved across a connection; a value of −1 specifies that there is no limit on the size of a TSDU; and a value of −2 specifies that the transfer of normal data is not supported by the transport provider.

etsdu — A value greater than zero specifies the maximum size of an expedited transport service data unit (ETSDU); a value of zero specifies that the transport provider does not support the concept of ETSDU, although it does support the sending of an expedited data stream with no logical boundaries preserved across a connection; a value of −1 specifies that there is no limit on the size of an ETSDU; and a value of −2 specifies that the transfer of expedited data is not supported by the transport provider.

connect — A value greater than or equal to zero specifies the maximum amount of data that 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` and `t_rcvdis` functions; a value of −1 specifies that there is no limit on the amount of data sent with these abortive release functions; and a value of −2 specifies that the transport provider does not allow data to be sent with the abortive release functions.

`servtype` This field specifies the service type supported by the transport provider, as described below.

If a transport user is concerned with protocol independence, the above sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the `t_alloc` 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.

The `servtype` field of `info` may specify one of the following values on return:

`T_COTS` The transport provider supports a connection-mode service but does not support the optional orderly release facility.

`T_COTS_ORD` The transport provider supports a connection-mode service with the optional orderly release facility.

`T_CLTS` The transport provider supports a connectionless-mode service. For this service type, `t_open` will return −2 for `etsdu`, `connect`, and `discon`.

On failure, `t_errno` may be set to one of the following:

`[TBADF]` The specified file descriptor does not refer to a transport endpoint.

`[TSYSERR]` A system error has occurred during execution of this function.

SEE ALSO

`t_open`(3N).

UNIX System V Network Programmer's Guide.

DIAGNOSTICS

`t_getinfo` returns 0 on success and −1 on failure and `t_errno` is set to indicate the error.

NAME

t_getstate – get the current state

SYNOPSIS

```
#include <tiuser.h>

int t_getstate(fd)
int fd;
```

DESCRIPTION

The t_getstate function returns the current state of the provider associated with the transport endpoint specified by fd.

On failure, t_errno may be set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TSTATECHNG]	The transport provider is undergoing a state change.
[TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

t_open(3N).

UNIX System V Network Programmer's Guide.

DIAGNOSTICS

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)

If the provider is undergoing a state transition when t_getstate is called, the function will fail.

NAME

t_listen – listen for a connect request

SYNOPSIS

```
#include <tiuser.h>

int t_listen(fd, call)
int fd;
struct t_call *call;
```

DESCRIPTION

This function listens for a connect request from a calling transport user. fd identifies the local transport endpoint where connect indications arrive, and on return, call contains information describing the connect indication. call points to a t_call structure, which contains the following members:

```
struct netbuf addr;
truct netbuf opt;
struct netbuf udata;
int sequence;
```

netbuf is described in intro(3). In call, addr returns the protocol address of the calling transport user, opt returns protocol-specific parameters associated with the connect request, udata returns any user data sent by the caller on the connect request, and sequence is a number that uniquely identifies the returned connect indication. The value of sequence enables the user to listen for multiple connect indications before responding to any of them.

Since this function returns values for the addr, opt, and udata fields of call, the maxlen [see netbuf in intro(3)] field of each must be set before issuing t_listen to indicate the maximum size of the buffer for each.

By default, t_listen executes in synchronous mode and waits for a connect indication to arrive before returning to the user. However, if O_NDELAY or O_NONBLOCK is set (via t_open or fcntl), t_listen executes asynchronously, reducing to a poll for existing connect indications. If none are available, it returns -1 and sets t_errno to TNODATA.

On failure, t_errno may be set to one of the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

[TBUFOVFLW] The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. The provider's state, as seen by the user, changes to T_INCON, and the connect indication information to be returned in call is discarded.

[TNODATA] O_NDELAY or O_NONBLOCK was set, but no connect indications had been queued.

[TLOOK] An asynchronous event has occurred on this transport endpoint and requires immediate attention.

[TNOTSUPPORT] This function is not supported by the underlying transport provider.

[TSYSERR] A system error has occurred during execution of this function.

CAVEATS

If a user issues `t_listen` in synchronous mode on a transport endpoint that was not bound for listening (i.e., `qlen` was zero on `t_bind`), the call will wait forever because no connect indications will arrive on that endpoint.

SEE ALSO

`intro`(3), `t_accept`(3N), `t_bind`(3N), `t_connect`(3N), `t_open`(3N), `t_rcvconnect`(3N).

UNIX System V Network Programmer's Guide.

DIAGNOSTICS

`t_listen` returns 0 on success and −1 on failure and `t_errno` is set to indicate the error.

NAME

`t_look` – look at the current event on a transport endpoint

SYNOPSIS

```
#include <tiuser.h>

int t_look(fd)
int fd;
```

DESCRIPTION

This function returns the current event on the transport endpoint specified by `fd`. This function enables a transport provider to notify a transport user of an asynchronous event when the user is issuing functions in synchronous mode. Certain events require immediate notification of the user and are indicated by a specific error, `TLOOK`, on the current or next function to be executed.

This function also enables a transport user to poll a transport endpoint periodically for asynchronous events.

On failure, `t_errno` may be set to one of the following:

`[TBADF]`	The specified file descriptor does not refer to a transport endpoint.
`[TSYSERR]`	A system error has occurred during execution of this function.

SEE ALSO

`t_open`(3N).
UNIX System V Network Programmer's Guide.

DIAGNOSTICS

Upon success, `t_look` returns a value that indicates which of the allowable events has occurred, or returns zero if no event exists. One of the following events is returned:

`T_LISTEN`	connection indication received
`T_CONNECT`	connect confirmation received
`T_DATA`	normal data received
`T_EXDATA`	expedited data received
`T_DISCONNECT`	disconnect received
`T_UDERR`	datagram error indication
`T_ORDREL`	orderly release indication

On failure, -1 is returned and `t_errno` is set to indicate the error.

NAME

t_open – establish a transport endpoint

SYNOPSIS

```
#include <tiuser.h>

#include <fcntl.h>

int t_open (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. This function establishes a transport endpoint by opening a UNIX file that identifies a particular transport provider (i.e., transport protocol) and returning a file descriptor that identifies that endpoint. For example, opening the file /dev/iso_cots identifies an OSI connection-oriented transport layer protocol as the transport provider.

path points to the path name of the file to open, and oflag identifies any open flags [as in open(2)]. oflag may be constructed from O_NDELAY or O_NONBLOCK OR-ed with O_RDWR. These flags are defined in the header file <fcntl.h>. t_open returns a file descriptor that will be used by all subsequent functions to identify the particular local transport endpoint.

This function also returns various default characteristics of the underlying transport protocol by setting fields in the t_info structure. This argument points to a t_info which contains the following members:

```
long addr;    /* max size of the transport protocol address */
long options;/* max number of bytes of protocol-specific options */
long tsdu;    /* max size of a transport service data unit (TSDU) */
long etsdu;   /* max size of an expedited transport service data unit (ETSDU) */
long connect;/* max amount of data allowed on connection establishment functions */
long discon; /* max amount of data allowed on t_snddis and t_rcvdis functions */
long servtype;/* service type supported by the transport provider */
```

The values of the fields have the following meanings:

addr
: A value greater than or equal to zero indicates the maximum size of a transport protocol address; a value of −1 specifies that there is no limit on the address size; and a value of -2 specifies that the transport provider does not provide user access to transport protocol addresses.

options
: A value greater than or equal to zero indicates the maximum number of bytes of protocol-specific options supported by the provider; a value of −1 specifies that there is no limit on the option size; and a value of −2 specifies that the transport provider does not support user-settable options.

tsdu
: A value greater than zero specifies the maximum size of a transport service data unit (TSDU); a value of zero specifies that the transport provider does not support the concept of TSDU, although it does support the sending of a data stream with no logical boundaries preserved across a connection; a value of −1 specifies that there is no limit on the size of a TSDU; and a value of −2 specifies that the transfer of normal data is not supported by the transport provider.

etsdu
: A value greater than zero specifies the maximum size of an expedited transport service data unit (ETSDU); a value of zero specifies that the transport provider does not support the concept of ETSDU, although it

does support the sending of an expedited data stream with no logical boundaries preserved across a connection; a value of −1 specifies that there is no limit on the size of an ETSDU; and a value of −2 specifies that the transfer of expedited data is not supported by the transport provider.

connect — A value greater than or equal to zero specifies the maximum amount of data that 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 and t_rcvdis functions; a value of −1 specifies that there is no limit on the amount of data sent with these abortive release functions; and a value of −2 specifies that the transport provider does not allow data to be sent with the abortive release functions.

servtype — This field specifies the service type supported by the transport provider, as described below.

If a transport user is concerned with protocol independence, the above sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the t_alloc 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 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 ULL by the transport user, no protocol information is returned by t_open.

On failure, t_errno may be set to the following:

[TSYSERR] — A system error has occurred during execution of this function.

[TBADFLAG] — An invalid flag is specified.

SEE ALSO

open(2).

UNIX System V Network Programmer's Guide.

DIAGNOSTICS

t_open returns a valid file descriptor on success and −1 on failure and t_errno is set to indicate the error.

NAME

t_optmgmt – manage options for a transport endpoint

SYNOPSIS

```
#include <tiuser.h>

int t_optmgmt (int fd, struct t_optmgmt *req, struct t_optmgmt *ret);
```

DESCRIPTION

The t_optmgmt function enables a transport user to retrieve, verify, or negotiate protocol options with the transport provider. fd identifies a bound transport endpoint.

The req and ret arguments point to a t_optmgmt structure containing the following members:

```
struct netbuf opt;
long flags;
```

The opt field identifies protocol options and the flags field is used to specify the action to take with those options.

The options are represented by a netbuf [see intro(3); also for len, buf, and maxlen] structure in a manner similar to the address in t_bind. req is used to request a specific action of the provider and to send options to the provider. len specifies the number of bytes in the options, buf points to the options buffer, and maxlen has no meaning for the req argument. The transport provider may return options and flag values to the user through ret. For ret, maxlen specifies the maximum size of the options buffer and buf points to the buffer where the options are to be placed. On return, len specifies the number of bytes of options returned. maxlen has no meaning for the req argument, but must be set in the ret argument to specify the maximum number of bytes the options buffer can hold. The actual structure and content of the options is imposed by the transport provider.

The flags field of req can specify one of the following actions:

T_NEGOTIATE This action enables the user to negotiate the values of the options specified in req with the transport provider. The provider will evaluate the requested options and negotiate the values, returning the negotiated values through ret.

T_CHECK This action enables the user to verify whether 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 This action enables a user to retrieve the default options supported by the transport provider into the opt field of ret. In req, the len field of opt must be zero and the buf field may be NULL.

If issued as part of the connectionless-mode service, t_optmgmt may block due to flow control constraints. The function will not complete until the transport provider has processed all previously sent data units.

On failure, t_errno may be set to one of the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

[TOUTSTATE] The function was issued in the wrong sequence.

[TACCES] The user does not have permission to negotiate the specified options.

[TBADOPT] The specified protocol options were in an incorrect format or contained illegal information.

[TBADFLAG] An invalid flag was specified.

[TBUFOVFLW] The number of bytes allowed for an incoming argument is not sufficient to store the value of that argument. The information to be returned in ret will be discarded.

[TSYSERR] A system error has occurred during execution of this function.

SEE ALSO

intro(3), t_getinfo(3N), t_open(3N).

UNIX System V Network Programmer's Guide.

DIAGNOSTICS

t_optmgmt returns 0 on success and −1 on failure and t_errno is set to indicate the error.

NAME

t_rcv – receive data or expedited data sent over a connection

SYNOPSIS

```
int t_rcv (int fd, char *buf, unsigned nbytes, int *flags);
```

DESCRIPTION

This function receives either normal or expedited data. fd identifies the local transport endpoint through which data will arrive, buf points to a receive buffer where user data will be placed, and nbytes specifies the size of the receive buffer. flags may be set on return from t_rcv and specifies optional flags as described below.

By default, t_rcv operates in synchronous mode and will wait for data to arrive if none is currently available. However, if O_NDELAY or O_NONBLOCK is set (via t_open or fcntl), t_rcv will execute in asynchronous mode and will fail if no data is available. (See TNODATA below.)

On return from the call, if T_MORE is set in flags, this indicates that there is more data and the current transport service data unit (TSDU) or expedited transport service data unit (ETSDU) must be received in multiple t_rcv calls. Each t_rcv with the T_MORE flag set indicates that another t_rcv must follow to get more data for the current TSDU. The end of the TSDU is identified by the return of a t_rcv call with the T_MORE flag not set. If the transport provider does not support the concept of a TSDU as indicated in the info argument on return from t_open or t_getinfo, the T_MORE flag is not meaningful and should be ignored.

On return, the data returned is expedited data if T_EXPEDITED is set in flags. If the number of bytes of expedited data exceeds nbytes, t_rcv will set T_EXPEDITED and T_MORE on return from the initial call. Subsequent calls to retrieve the remaining ETSDU will have T_EXPEDITED set on return. The end of the ETSDU is identified by the return of a t_rcv call with the T_MORE flag not set.

If expedited data arrives after part of a TSDU has been retrieved, receipt of the remainder of the TSDU will be suspended until the ETSDU has been processed. Only after the full ETSDU has been retrieved (T_MORE not set) will the remainder of the TSDU be available to the user.

On failure, t_errno may be set to one of the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

[TNODATA] O_NDELAY or O_NONBLOCK was set, but no data is currently available from the transport provider.

[TLOOK] An asynchronous event has occurred on this transport endpoint and requires immediate attention.

[TNOTSUPPORT] This function is not supported by the underlying transport provider.

[TSYSERR] A system error has occurred during execution of this function.

SEE ALSO

t_open(3N), t_snd(3N).

UNIX System V Network Programmer's Guide.

DIAGNOSTICS

On successful completion, t_rcv returns the number of bytes received, and it returns −1 on failure and t_errno is set to indicate the error.

NAME

t_rcvconnect – receive the confirmation from a connect request

SYNOPSIS

```
#include <tiuser.h>

int t_rcvconnect (int fd, struct t_call *call);
```

DESCRIPTION

This function enables a calling transport user to determine the status of a previously sent connect request and is used in conjunction with t_connect to establish a connection in asynchronous mode. The connection will be established on successful completion of this function.

fd identifies the local transport endpoint where communication will be established, and call contains information associated with the newly established connection. call points to a t_call structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

netbuf is described in intro(3). In call, addr returns the protocol address associated with the responding transport endpoint, opt presents any protocol-specific information associated with the connection, udata points to optional user data that may be returned by the destination transport user during connection establishment, and sequence has no meaning for this function.

The maxlen [see netbuf in intro(3)] field of each argument must be set before issuing this function to indicate the maximum size of the buffer for each. However, call may be NULL, in which case no information is given to the user on return from t_rcvconnect. By default, t_rcvconnect executes in synchronous mode and waits for the connection to be established before returning. On return, the addr, opt, and udata fields reflect values associated with the connection.

If O_NDELAY or O_NONBLOCK is set (via t_open or fcntl), t_rcvconnect executes in asynchronous mode, and reduces to a poll for existing connect confirmations. If none are available, t_rcvconnect fails and returns immediately without waiting for the connection to be established. (See TNODATA below.) t_rcvconnect must be re-issued at a later time to complete the connection establishment phase and retrieve the information returned in call.

On failure, t_errno may be set to one of the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

[TBUFOVFLW] The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument and the connect information to be returned in call will be discarded. The provider's state, as seen by the user, will be changed to DATAXFER.

[TNODATA] O_NDELAY or O_NONBLOCK was set, but a connect confirmation has not yet arrived.

[TLOOK] An asynchronous event has occurred on this transport connection and requires immediate attention.

[TNOTSUPPORT] This function is not supported by the underlying transport provider.

[TSYSERR] A system error has occurred during execution of this function.

SEE ALSO

intro(3), t_accept(3N), t_bind(3N), t_connect(3N), t_listen(3N), t_open(3N).

UNIX System V Network Programmer's Guide.

DIAGNOSTICS

t_rcvconnect returns 0 on success and −1 on failure and t_errno is set to indicate the error.

NAME

t_rcvdis – retrieve information from disconnect

SYNOPSIS

```
#include <tiuser.h>

t_rcvdis (int fd, struct t_discon *discon);
```

DESCRIPTION

This function is used to identify the cause of a disconnect, and to retrieve any user data sent with the disconnect. fd identifies the local transport endpoint where the connection existed, and discon points to a t_discon structure containing the following members:

```
struct netbuf udata;
int reason;
int sequence;
```

netbuf is described in intro(3). reason specifies the reason for the disconnect through a protocol-dependent reason code, udata identifies any user data that was sent with the disconnect, and sequence may identify an outstanding connect indication with which the disconnect is associated. sequence is only meaningful when t_rcvdis is issued by a passive transport user who has executed one or more t_listen functions and is processing the resulting connect indications. If a disconnect indication occurs, sequence can be used to identify which of the outstanding connect indications is associated with the disconnect.

If a user does not care if there is incoming data and does not need to know the value of reason or sequence, discon may be NULL and any user data associated with the disconnect will be discarded. However, if a user has retrieved more than one outstanding connect indication (via t_listen) and discon is NULL, the user will be unable to identify which connect indication the disconnect is associated with.

On failure, t_errno may be set to one of the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

[TNODIS] No disconnect indication currently exists on the specified transport endpoint.

[TBUFOVFLW] The number of bytes allocated for incoming data is not sufficient to store the data. The provider's state, as seen by the user, will change to T_IDLE, and the disconnect indication information to be returned in discon will be discarded.

[TNOTSUPPORT] This function is not supported by the underlying transport provider.

[TSYSERR] A system error has occurred during execution of this function.

SEE ALSO

intro(3), t_connect(3N), t_listen(3N), t_open(3N), t_snddis(3N).
UNIX System V Network Programmer's Guide.

DIAGNOSTICS

t_rcvdis returns 0 on success and -1 on failure and t_errno is set to indicate the error.

NAME

t_rcvrel – acknowledge receipt of an orderly release indication

SYNOPSIS

```
#include <tiuser.h>

t_rcvrel (int fd);
```

DESCRIPTION

This function is used to acknowledge receipt of an orderly release indication. fd identifies the local transport endpoint where the connection exists. After receipt of this indication, the user should not attempt to receive more data because such an attempt will block forever. However, the user may continue to send data over the connection if t_sndrel has not been issued by the user.

This function is an optional service of the transport provider, and is only supported if the transport provider returned service type T_COTS_ORD on t_open or t_getinfo.

On failure, t_errno may be set to one of the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

[TNOREL] No orderly release indication currently exists on the specified transport endpoint.

[TLOOK] An asynchronous event has occurred on this transport endpoint and requires immediate attention.

[TNOTSUPPORT] This function is not supported by the underlying transport provider.

[TSYSERR] A system error has occurred during execution of this function.

SEE ALSO

t_open(3N), t_sndrel(3N).

UNIX System V Network Programmer's Guide.

DIAGNOSTICS

t_rcvrel returns 0 on success and −1 on failure t_errno is set to indicate the error.

NAME

t_rcvudata – receive a data unit

SYNOPSIS

```
#include <tiuser.h>

int t_rcvudata (int fd, struct t_unitdata *unitdata, int *flags);
```

DESCRIPTION

This function is used in connectionless mode to receive a data unit from another transport user. fd identifies the local transport endpoint through which data will be received, unitdata holds information associated with the received data unit, and flags is set on return to indicate that the complete data unit was not received. unitdata points to a t_unitdata structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

The maxlen [see netbuf in intro(3)] field of addr, opt, and udata must be set before issuing this function to indicate the maximum size of the buffer for each.

On return from this call, addr specifies the protocol address of the sending user, opt identifies protocol-specific options that were associated with this data unit, and udata specifies the user data that was received.

By default, t_rcvudata operates in synchronous mode and will wait for a data unit to arrive if none is currently available. However, if O_NDELAY or O_NONBLOCK is set (via t_open or fcntl), t_rcvudata will execute in asynchronous mode and will fail if no data units are available.

If the buffer defined in the udata field of unitdata is not large enough to hold the current data unit, the buffer will be filled and T_MORE will be set in flags on return to indicate that another t_rcvudata should be issued to retrieve the rest of the data unit. Subsequent t_rcvudata call(s) will return zero for the length of the address and options until the full data unit has been received.

On failure, t_errno may be set to one of the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

[TNODATA] O_NDELAY or O_NONBLOCK was set, but no data units are currently available from the transport provider.

[TBUFOVFLW] The number of bytes allocated for the incoming protocol address or options is not sufficient to store the information. The unit data information to be returned in unitdata will be discarded.

[TLOOK] An asynchronous event has occurred on this transport endpoint and requires immediate attention.

[TNOTSUPPORT] This function is not supported by the underlying transport provider.

[TSYSERR] A system error has occurred during execution of this function.

SEE ALSO

intro(3), t_rcvuderr(3N), t_sndudata(3N).

UNIX System V Network Programmer's Guide.

DIAGNOSTICS

`t_rcvudata` returns 0 on successful completion and −1 on failure and `t_errno` is set to indicate the error.

NAME

t_rcvuderr – receive a unit data error indication

SYNOPSIS

```
#include <tiuser.h>

int t_rcvuderr (int fd, struct t_uderr *uderr);
```

DESCRIPTION

This function is used in connectionless mode to receive information concerning an error on a previously sent data unit, and should be issued only after a unit data error indication. It informs the transport user that a data unit with a specific destination address and protocol options produced an error. fd identifies the local transport endpoint through which the error report will be received, and uderr points to a t_uderr structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
long error;
```

netbuf is described in intro(3). The maxlen [see netbuf in intro(3)] field of addr and opt must be set before issuing this function to indicate the maximum size of the buffer for each.

On return from this call, the addr structure specifies the destination protocol address of the erroneous data unit, the opt structure identifies protocol-specific options that were associated with the data unit, and error specifies a protocol-dependent error code.

If the user does not care to identify the data unit that produced an error, uderr may be set to NULL and t_rcvuderr will simply clear the error indication without reporting any information to the user.

On failure, t_errno may be set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TNOUDERR]	No unit data error indication currently exists on the specified transport endpoint.
[TBUFOVFLW]	The number of bytes allocated for the incoming protocol address or options is not sufficient to store the information. The unit data error information to be returned in uderr will be discarded.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

intro(3), t_rcvudata(3N), t_sndudata(3N).

UNIX System V Network Programmer's Guide.

DIAGNOSTICS

t_rcvuderr returns 0 on successful completion and -1 on failure and t_errno is set to indicate the error.

NAME

t_snd – send data or expedited data over a connection

SYNOPSIS

```
#include <tiuser:h>

int t_snd (int fd, char *buf, unsigned nbytes, int flags);
```

DESCRIPTION

This function is used to send either normal or expedited data. fd identifies the local transport endpoint over which data should be sent, buf points to the user data, nbytes specifies the number of bytes of user data to be sent, and flags specifies any optional flags described below.

By default, t_snd operates in synchronous mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if O_NDELAY or O_NONBLOCK is set (via t_open or fcntl), t_snd will execute in asynchronous mode, and will fail immediately if there are flow control restrictions.

Even when there are no flow control restrictions, t_snd will wait if STREAMS internal resources are not available, regardless of the state of O_NDELAY or O_NONBLOCK.

On successful completion, t_snd returns the number of bytes accepted by the transport provider. Normally this will equal the number of bytes specified in nbytes. However, if O_NDELAY or O_NONBLOCK is set, it is possible that only part of the data will be accepted by the transport provider. In this case, t_snd will set T_MORE for the data that was sent (see below) and will return a value less than nbytes. If nbytes is zero and sending of zero bytes is not supported by the underlying transport provider, t_snd() will return –1 with t_errno set to TBADDATA. A return value of zero indicates that the request to send a zero-length data message was sent to the provider.

If T_EXPEDITED is set in flags, the data will be sent as expedited data, and will be subject to the interpretations of the transport provider.

If T_MORE is set in flags, or is set as described above, an indication is sent to the transport provider that the transport service data unit (TSDU) or expedited transport service data unit (ETSDU) is being sent through multiple t_snd calls. Each t_snd with the T_MORE flag set indicates that another t_snd will follow with more data for the current TSDU. The end of the TSDU (or ETSDU) is identified by a t_snd call with the T_MORE flag not set. Use of T_MORE enables a user to break up large logical data units without losing the boundaries of those units at the other end of the connection. The flag implies nothing about how the data is packaged for transfer below the transport interface. If the transport provider does not support the concept of a TSDU as indicated in the info argument on return from t_open or t_getinfo, the T_MORE flag is not meaningful and should be ignored.

The size of each TSDU or ETSDU must not exceed the limits of the transport provider as returned by t_open or t_getinfo. If the size is exceeded, a TSYSERR with system error EPROTO will occur. However, the t_snd may not fail because EPROTO errors may not be reported immediately. In this case, a subsequent call that accesses the transport endpoint will fail with the associated TSYSERR.

If t_snd is issued from the T_IDLE state, the provider may silently discard the data. If t_snd is issued from any state other than T_DATAXFER, T_INREL or T_IDLE, the provider will generate a TSYSERR with system error EPROTO (which may be reported in the manner described above).

On failure, t_errno may be set to one of the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

[TFLOW] O_NDELAY or O_NONBLOCK was set, but the flow control mechanism prevented the transport provider from accepting data at this time.

[TNOTSUPPORT] This function is not supported by the underlying transport provider.

[TSYSERR] A system error [see intro(2)] has been detected during execution of this function.

[TBADDATA] nbytes is zero and sending zero bytes is not supported by the transport provider.

SEE ALSO

t_open(3N), t_rcv(3N).

UNIX System V Network Programmer's Guide.

DIAGNOSTICS

On successful completion, t_snd returns the number of bytes accepted by the transport provider, and it returns −1 on failure and t_errno is set to indicate the error.

NAME

t_snddis – send user-initiated disconnect request

SYNOPSIS

```
#include <tiuser.h>

int t_snddis (int fd, struct t_call *call):
```

DESCRIPTION

This function is used to initiate an abortive release on an already established connection or to reject a connect request. fd identifies the local transport endpoint of the connection, and call specifies information associated with the abortive release. call points to a t_call structure that contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

netbuf is described in intro(3). The values in call have different semantics, depending on the context of the call to t_snddis. When rejecting a connect request, call must be non-NULL and contain a valid value of sequence to identify uniquely 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 in the discon field of the info argument of t_open or t_getinfo. If the len field of udata is zero, no data will be sent to the remote user.

On failure, t_errno may be set to one of the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

[TOUTSTATE] The function was issued in the wrong sequence. The transport provider's outgoing queue may be flushed, so data may be lost.

[TBADDATA] The amount of user data specified was not within the bounds allowed by the transport provider. The transport provider's outgoing queue will be flushed, so data may be lost.

[TBADSEQ] An invalid sequence number was specified, or a NULL call structure was specified when rejecting a connect request. The transport provider's outgoing queue will be flushed, so data may be lost.

[TLOOK] An asynchronous event has occurred on this transport endpoint and requires immediate attention.

[TNOTSUPPORT] This function is not supported by the underlying transport provider.

[TSYSERR] A system error has occurred during execution of this function.

SEE ALSO

intro(3), t_connect(3N), t_getinfo(3N), t_listen(3N), t_open(3N).

UNIX System V Network Programmer's Guide.

DIAGNOSTICS

`t_snddis` returns 0 on success and −1 on failure and `t_errno` is set to indicate the error.

NAME

t_sndrel – initiate an orderly release

SYNOPSIS

```
#include <tiuser.h>

int t_sndrel (int fd);
```

DESCRIPTION

This function is used to initiate an orderly release of a transport connection and indicates to the transport provider that the transport user has no more data to send. fd identifies the local transport endpoint where the connection exists. After issuing t_sndrel, the user may not send any more data over the connection. However, a user may continue to receive data if an orderly release indication has not been received.

This function is an optional service of the transport provider, and is only supported if the transport provider returned service type T_COTS_ORD on t_open or t_getinfo.

If t_sndrel is issued from an invalid state, the provider will generate an EPROTO protocol error; however, this error may not occur until a subsequent reference to the transport endpoint.

On failure, t_errno may be set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TFLOW]	O_NDELAY or O_NONBLOCK was set, but the flow control mechanism prevented the transport provider from accepting the function at this time.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

t_open(3N), t_rcvrel(3N).

UNIX System V Network Programmer's Guide.

DIAGNOSTICS

t_sndrel returns 0 on success and –1 on failure and t_errno is set to indicate the error.

NAME

t_sndudata – send a data unit

SYNOPSIS

```
#include <tiuser.h>

int t_sndudata (int fd, struct t_unitdata *unitdata);
```

DESCRIPTION

This function is used in connectionless mode to send a data unit to another transport user. fd identifies the local transport endpoint through which data will be sent, and unitdata points to a t_unitdata structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

netbuf is described in intro(3). In unitdata, addr specifies the protocol address of the destination user, opt identifies protocol-specific options that the user wants associated with this request, and udata specifies the user data to be sent. The user may choose not to specify what protocol options are associated with the transfer by setting the len field of opt to zero. In this case, the provider may use default options.

If the len field of udata is zero, and the sending of zero bytes is not supported by the underlying transport provider, t_sndudata will return -1 with t_errno set to TBADDATA.

By default, t_sndudata operates in synchronous mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if O_NDELAY or O_NONBLOCK is set (via t_open or fcntl), t_sndudata will execute in asynchronous mode and will fail under such conditions.

If t_sndudata is issued from an invalid state, or if the amount of data specified in udata exceeds the TSDU size as returned in the tsdu field of the info argument of t_open or t_getinfo, the provider will generate an EPROTO protocol error. (See TSYSERR below.) If the state is invalid, this error may not occur until a subsequent reference is made to the transport endpoint.

On failure, t_errno may be set to one of the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

[TFLOW] O_NDELAY or O_NONBLOCK was set, but the flow control mechanism prevented the transport provider from accepting data at this time.

[TNOTSUPPORT] This function is not supported by the underlying transport provider.

[TSYSERR] A system error has occurred during execution of this function.

[TBADDATA] nbytes is zero and sending zero bytes is not supported by the transport provider.

SEE ALSO

intro(3), t_rcvudata(3N), t_rcvuderr(3N).
UNIX System V Network Programmer's Guide.

DIAGNOSTICS

t_sndudata returns 0 on successful completion and −1 on failure t_errno is set to indicate the error.

NAME

t_sync – synchronize transport library

SYNOPSIS

```
#include <tiuser.h>

int t_sync (int fd);
```

DESCRIPTION

For the transport endpoint specified by fd, t_sync synchronizes the data structures managed by the transport library with information from the underlying transport provider. In doing so, it can convert a raw file descriptor [obtained via open(2), dup(2), or as a result of a fork(2) and exec(2)] to an initialized transport endpoint, assuming that file descriptor referenced.a transport provider. This function also allows two cooperating processes to synchronize their interaction with a transport provider.

For example, if a process forks a new process and issues an exec, the new process must issue a t_sync to build the private library data structure associated with a transport endpoint and to synchronize the data structure with the relevant provider information.

It is important to remember that the transport provider treats all users of a transport endpoint as a single user. If multiple processes are using the same endpoint, they should coordinate their activities so as not to violate the state of the provider. t_sync returns the current state of the provider to the user, thereby enabling the user to verify the state before taking further action. This coordination is only valid among cooperating processes; it is possible that a process or an incoming event could change the provider's state *after* a t_sync is issued.

If the provider is undergoing a state transition when t_sync is called, the function will fail.

On failure, t_errno may be set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TSTATECHNG]	The transport provider is undergoing a state change.
[TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

dup(2), exec(2), fork(2), open(2).

UNIX System V Network Programmer's Guide.

DIAGNOSTICS

t_sync returns the state of the transport provider on successful completion and -1 on failure and t_errno is set to indicate the error. The state returned 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)

NAME

t_unbind – disable a transport endpoint

SYNOPSIS

```
#include <tiuser.h>

int t_unbind (int fd);
```

DESCRIPTION

The t_unbind function disables the transport endpoint specified by fd which was previously bound by t_bind(3N). On completion of this call, no further data or events destined for this transport endpoint will be accepted by the transport provider.

On failure, t_errno may be set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TOUTSTATE]	The function was issued in the wrong sequence.
[TLOOK]	An asynchronous event has occurred on this transport endpoint.
[TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

t_bind(3N).

UNIX System V Network Programmer's Guide.

DIAGNOSTICS

t_unbind returns 0 on success and −1 on failure and t_errno is set to indicate the error.

NAME

`tcflush`: `tcsendbreak`, `tcdrain`, `tcflush`, `tcflow` – control data transmission

SYNOPSIS

```
#include <termios.h>

int tcsendbreak (fildes, duration)
int fildes;
int duration;

int tcdrain (fildes)
int fildes;

int tcflush (fildes, queue_selector)
int fildes;
int queue_selector;

int tcflow (fildes, action)
int fildes;
int action;
```

DESCRIPTION

If the terminal is using asynchronous serial data transmission, the `tcsendbreak()` function shall cause transmission of a continuous stream of zero-valued bits for a specific duration. If *duration* is zero, it shall cause transmission of zero-valued bits for at least 0.25 seconds, and not more that 0.5 seconds. If *duration* is not zero, it shall send zero-valued bits for an implementation-defined period of time.

If the terminal is not using asynchronous serial data transmission, it is implementation-defined whether the `tcsendbreak()` function sends data to generate a break condition (as defined by the implementation) or returns without taking any action.

The `tcdrain()` function shall wait until all output written to the object referred to by *fildes* has been transmitted.

The `tcflush()` function shall discard data written to the object referred to by *fildes* but not transmitted, or data received but not read, depending on the value of *queue_selector*:

(1) If *queue_selector* is TCIFLUSH, it shall flush data received but not read.

(2) If *queue_selector* is TCOFLUSH, it shall flush data written but not transmitted.

(3) If *queue_selector* is TCIOFLUSH, it shall flush both data received but not read, and data written but not transmitted.

The `tcflow()` function shall suspend transmission or reception of data on the object referred to by *fildes*, depending on the value of *action*:

(1) If *action* is TCOOFF, it shall suspend output.

(2) If *action* is TCOON, it shall restart suspended output.

(3) If *action* is TCIOFF, the system shall transmit a STOP character, which is intended to cause the terminal device to stop transmitting data to the system.

(4) If *action* is TCION, the system shall transmit a START character, which is intended to cause the terminal device to start transmitting data to the system.

The symbolic constants for the values of *queue_selector* and *action* are defined in `<termios.h>`.

The default on open of a terminal file is that neither its input nor its output is suspended.

RETURNS

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

DIAGNOSTICS

If any of the following conditions occur, the `tcsendbreak()` function shall return -1 and set `errno` to the corresponding value:

[EBADF]
The *fildes* argument is not a valid file descriptor.

[ENOTTY]
The file associated with *fildes* is not a terminal.

If any of the following conditions occur, the `tcdrain()` function shall return -1 and set `errno` to the corresponding value:

[EBADF]
The *fildes* argument is not a valid file descriptor.

[EINTR]
A signal interrupted the `tcdrain()` function.

[ENOTTY]
The file associated with *fildes* is not a terminal.

If any of the following conditions occur, the `tcflush()` function shall return -1 and set `errno` to the corresponding value:

[EBADF]
The *fildes* argument is not a valid file descriptor.

[EINVAL]
The *queue_selector* argument is not a proper value.

[ENOTTY]
The file associated with *fildes* is not a terminal.

If any of the following conditions occur, the `tcflow()` function shall return -1 and set `errno` to the corresponding value:

[EBADF]
The *fildes* argument is not a valid file descriptor.

[EINVAL]
The *action* argument is not a proper value.

[ENOTTY]
The file associated with *fildes* is not a terminal.

SEE ALSO

`tcsetattr(3C)`, `<termios.h>`.

COPYRIGHTS

Portions of this text are reprinted from IEEE Std 1003.1-1988, *Portable Operating System Interface for Computer Environment*, copyright © 1988 by the Institute of Electrical and Electronics Engineers, Inc., with the permission of the IEEE Standards Department. To purchase IEEE Standards, call 800/678-IEEE.

In the event of a discrepancy between the electronic and the original printed version, the original version takes precedence.

STANDARDS

If the *duration* argument to the `tcsendbreak()` function is not zero, the function will send zero-valued bits for *duration* microseconds.

When `tcsendbreak()` is invoked on a pseudo-terminal device file, no data will be sent unless the pseudo-terminal is in packet mode. See `pty(7)` for details.

NAME

tcgetpgrp – get foreground process group ID

SYNOPSIS

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

pid_t tcgetpgrp (fildes)
int fildes;
```

DESCRIPTION

If {_POSIX_JOB_CONTROL} is defined:

(1) The tcgetpgrp() function shall return the value of the process group ID of the foreground process group associated with the terminal.

(2) The tcgetpgrp() function 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.

Otherwise:

The implementation shall either support the tcgetpgrp() function as described above, or the tcgetpgrp() call shall fail.

RETURNS

Upon successful completion, tcgetpgrp() returns the process group ID of the foreground process group associated with the terminal. Otherwise, a value of −1 is returned and errno is set to indicate the error.

DIAGNOSTICS

If any of the following conditions occur, the tcgetpgrp() function shall return −1 and set errno to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[ENOSYS]	The tcgetpgrp() function is not supported in this implementation.
[ENOTTY]	The calling process does not have a controlling terminal or the file is not the controlling terminal.

SEE ALSO

setpgid(2), setsid(2), tcsetpgrp(3C).

COPYRIGHTS

Portions of this text are reprinted from IEEE Std 1003.1-1988, *Portable Operating System Interface for Computer Environment*, copyright © 1988 by the Institute of Electrical and Electronics Engineers, Inc., with the permission of the IEEE Standards Department. To purchase IEEE Standards, call 800/678-IEEE.

In the event of a discrepancy between the electronic and the original printed version, the original version takes precedence.

STANDARDS

The tcgetpgrp() function is fully supported, regardless of whether _POSIX_JOB_CONTROL is defined.

NAME

`tcgetattr`, `tcsetattr` – get and set state

SYNOPSIS

```
#include <termios.h>

int tcgetattr (fildes, termios_p)
int fildes;
struct termios *termios_p;

int tcsetattr (fildes, optional_actions, termios_p)
int fildes, optional_actions;
struct termios *termios_p;
```

DESCRIPTION

The `tcgetattr()` function shall get the parameters associated with the object referred to by *fildes* and store them in the *termios* structure referenced by *termios_p*. This function is allowed from a background process; however, the terminal attributes may be subsequently changed by a foreground process.

The `tcsetattr()` function shall set the parameters associated with the terminal (unless support is required from the underlying hardware that is not available) from the *termios* structure referenced by *termios_p* as follows:

(1) If *optional_actions* is TCSANOW, the change shall occur immediately.

(2) If *optional_actions* is TCSADRAIN, the change shall occur after all output written to *fildes* has been transmitted. This function should be used when changing parameters that affect output.

(3) If *optional_actions* is TCSAFLUSH, the change shall occur after all output written to the object referred to by *fildes* has been transmitted, and all input that has been received but not read shall be discarded before the change is made.

The symbolic constants for the values of *optional_actions* are defined in `<termios.h>`.

RETURNS

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

DIAGNOSTICS

If any of the following conditions occur, the `tcgetattr()` function shall return –1 and set `errno` to the corresponding value:

[EBADF]
The *fildes* argument is not a valid file descriptor.

[ENOTTY]
The file associated with *fildes* is not a terminal.

If any of the following conditions occur, the `tcsetattr()` function shall return –1 and set `errno` to the corresponding value:

[EBADF]
The *fildes* argument is not a valid file descriptor.

[EINVAL]
The *optional_actions* argument is not a proper value, or an attempt was made

to change an attribute represented in the *termios* structure to an unsupported value.

[ENOTTY]
The file associated with *fildes* is not a terminal.

SEE ALSO

`tcflush`(3C), `tcsetpgrp`(3C), `<termios.h>`.

COPYRIGHTS

In the event of a discrepancy between the electronic and the original printed version, the original version takes precedence.

NAME

tcsetpgrp – set terminal foreground process group id

SYNOPSIS

```
#include <unistd.h>

int tcsetpgrp (int fildes, pid_t pgid)
```

DESCRIPTION

tcsetpgrp sets the foreground process group ID of the terminal specified by *fildes* to *pgid*. The file associated with *fildes* must be the controlling terminal of the calling process and the controlling terminal must be currently associated with the session of the calling process. The value of *pgid* must match a process group ID of a process in the same session as the calling process.

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

EBADF The *fildes* argument is not a valid file descriptor.

EINVAL The *fildes* argument is a terminal that does not support tcsetpgrp, or *pgid* is not a valid process group ID.

ENOTTY The calling process does not have a controlling terminal, or the file is not the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.

EPERM *pgid* does not match the process group ID of an existing process in the same session as the calling process.

SEE ALSO

tcsetpgrp(3C), tcsetsid(3C).
termio(7) in the *System Administrator's Reference Manual.*

DIAGNOSTICS

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

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;
int destcol, destline;

tputs(cp, affcnt, outc)
register char *cp;
int affcnt;
int (*outc)();
```

DESCRIPTION

These functions extract and use routines from the terminal capability data base termcap(5). These are obsolete low level routines; see terminfo(4) for an equivalent but more modern package, and curses(3X) for a higher level package.

tgetent extracts the entry for terminal *name* into the buffer at *bp*. *bp* should be a character buffer of size 1024 and must be retained through all subsequent calls to *tgetnum, tgetflag,* and *tgetstr*. *tgetent* returns –1 if it cannot open the *termcap* file, 0 if the terminal name given does not have an entry, and 1 if all goes well.

tgetent looks in the environment for a TERMCAP variable. If it is found, and its 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*; it will also help you debug new terminal descriptions or to make one for your terminal if you can't write the file */etc/termcap*.

tgetnum gets the numeric value of capability *id,* returning –1 if 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,* and advancing the *area* pointer. It decodes the abbreviations for this field described in termcap(5), except for cursor addressing and padding information.

tgoto returns a cursor addressing string decoded from *cm* to go to column *destcol* in line *destline*. It uses the external variables UP (from the up capability) and BC (if bc is given rather than bs) if necessary to avoid placing \n, \r, ^D, ^H, or ^@ in the returned string.

Programs which call tgoto should be sure to turn off tab expansion into spaces in the terminal driver since *tgoto* may now output a tab. Note that programs using *termcap* should turn off tab expansion anyway since some terminals use the tab character (^I) for other functions. If a % sequence is given which is not understood, then *tgoto* returns "OOPS".

tputs decodes the leading padding information of the string *cp; affcnt* is the number of lines affected by the operation, or 1 if this is not applicable. *outc* is a routine that is called with each character in turn. The external variable *ospeed* should contain the output speed of the terminal as encoded by *ioctl (2)* The external variable PC should contain a pad character to be used (from the pc capability) if a null (^@) is inappropriate.

FILES

/lib/libtermcap.a	-l*termcap* library
/etc/termcap	terminal information data base

SEE ALSO

curses(3X), terminfo(4), termcap(5).
captoinfo(1M), infocmp(1M) in the *System Manager's Reference for the DG/UX System*.

NAME

termios: tcgetattr, tcsetattr, tcsendbreak, tcdrain, tcflush, tcflow, cfgetospeed, cfgetispeed, cfsetispeed, cfsetospeed, tcgetpgrp, tcsetpgrp, tcgetsid – general terminal interface

SYNOPSIS

```
#include <termios.h>

int tcgetattr(int fildes, struct termios *termios_p);

int tcsetattr(int fildes, int optional_actions,
      const struct termios *termios_p);

int tcsendbreak(int fildes, int duration);

int tcdrain(int fildes);

int tcflush(int fildes, int queue_selector);

int tcflow(int fildes, int action);

speed_t cfgetospeed(struct termios *termios_p);

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

speed_t cfgetispeed(struct termios *termios_p);

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

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

pid_t tcgetpgrp(int fildes);

int tcsetpgrp(int fildes, pid_t pgid);

pid_t tcgetsid(int fildes);
```

DESCRIPTION

These functions describe a general terminal interface for controlling asynchronous communications ports. A more detailed overview of the terminal interface can be found in termio(7), which also describes an ioctl(2) interface that provides 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*(7)*.

Get and Set Terminal Attributes

The tcgetattr function gets the parameters associated with the object referred by *fildes* 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.

The tcsetattr function sets the parameters associated with the terminal (unless support is required from the underlying hardware that is not available) from the termios structure referenced 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 *fildes* 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 *fildes* has been transmitted, and all input that has been received but not read is discarded before the change is made.

The symbolic constants for the values of *optional_actions* are defined in <termios.h>.

Line Control

If the terminal is using asynchronous serial data transmission, the tcsendbreak function causes transmission of a continuous stream of zero-valued bits for a specific duration. If *duration* is zero, it causes transmission of zero-valued bits for at least 0.25 seconds, and not more than 0.5 seconds. If *duration* is not zero, it behaves in a way similar to tcdrain.

If the terminal is not using asynchronous serial data transmission, the tcsendbreak function sends data to generate a break condition or returns without taking any action.

The tcdrain function waits until all output written to the object referred to by *fildes* has been transmitted.

The tcflush function discards data written to the object referred to by *fildes* 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 tcflow function suspends transmission or reception of data on the object referred to by *fildes*, depending on the value of *action*:

> If *action* is TCOOFF, it suspends output.
>
> If *action* is TCOON, it restarts suspended output.
>
> If *action* if TCIOFF, the system transmits a STOP character, which causes the terminal device to stop transmitting data to the system.
>
> If *action* is TCION, the system transmits a START character, which causes the terminal device to start transmitting data to the system.

Get and Set Baud Rate

The baud rate functions get and set the values of the input and output baud rates in the termios structure. The effects on the terminal device described below do not become effective until the tcsetattr function is 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 gets the output baud rate and stores it 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 are no longer be asserted. Normally, this disconnects the line.

cfgetispeed gets the input baud rate and stores it in the termios structure pointed to by *termios_p*.

cfsetispeed sets the input baud rate stored in the termios structure pointed to by *termios_p* to *speed*. If the input baud rate is set to zero, the input baud rate is specified by the value of the output baud rate. Both cfsetispeed and cfsetospeed return a value of zero if successful and −1 to indicate an error. Attempts to set unsupported baud rates are ignored. This refers both to changes to baud rates not supported by the hardware, and to changes setting the input and output baud rates to different values if the hardware does not support this.

Get and Set Terminal Foreground Process Group ID

tcsetpgrp sets the foreground process group ID of the terminal specified by *fildes* to *pgid*. The file associated with *fildes* must be the controlling terminal of the calling process and the controlling terminal must be currently associated with the session of the calling process. *pgid* must match a process group ID of a process in the same session as the calling process.

tcgetpgrp returns the foreground process group ID of the terminal specified by *fildes*. 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.

Get Terminal Session ID

tcgetsid returns the session ID of the terminal specified by *fildes*.

DIAGNOSTICS

On success, tcgetpgrp returns the process group ID of the foreground process group associated with the specified terminal. Otherwise, it returns −1 and sets errno to indicate the error.

On success, tcgetsid returns the session ID associated with the specified terminal. Otherwise, it returns −1 and sets errno to indicate the error.

On success, all other functions return a value of 0. Otherwise, they return −1 and set errno to indicate the error.

All of the functions fail if one of more of the following is true:

EBADF The *fildes* argument is not a valid file descriptor.

ENOTTY The file associated with *fildes* is not a terminal.

tcsetattr also fails if the following is true:

EINVAL The *optional_actions* argument is not a proper value, or an attempt was made to change an attribute represented in the termios structure to an unsupported value.

tcsendbreak also fails if the following is true:

EINVAL The device does not support the tcsendbreak function.

tcdrain also fails if one or more of the following is true:

EINTR A signal interrupted the tcdrain function.

EINVAL The device does not support the tcdrain function.

tcflush also fails if the following is true:

EINVAL The device does not support the tcflush function or the *queue_selector* argument is not a proper value.

tcflow also fails if the following is true:

EINVAL The device does not support the tcflow function or the *action* argument is not a proper value.

tcgetpgrp also fails if the following is true:

ENOTTY the calling process does not have a controlling terminal, or *fildes* does not refer to the controlling terminal.

tcsetpgrp also fails if the following is true:

EINVAL *pgid* is not a valid process group ID .

ENOTTY the calling process does not have a controlling terminal, or *fildes* does not refer to the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.

EPERM *pgid* does not match the process group of an existing process in the same session as the calling process.

tcgetsid also fails if the following is true:

EACCES *fildes* is a terminal that is not allocated to a session.

SEE ALSO

setsid(2), setpgid(2), termio(7).

NAME

tmpfile – create a temporary file

SYNOPSIS

```
#include <stdio.h>

FILE *tmpfile (void);
```

DESCRIPTION

tmpfile creates a temporary file using a name generated by the tmpnam routine and returns a corresponding FILE pointer. If the file cannot be opened, a NULL pointer is returned. The file is automatically deleted when the process using it terminates or when the file is closed. The file is opened for update ("w+").

SEE ALSO

creat(2), open(2), unlink(2), fopen(3S), mktemp(3C), perror(3C), stdio(3S), tmpnam(3S).

NAME

tmpnam, tempnam – create a name for a temporary file

SYNOPSIS

```
#include <stdio.h>

char *tmpnam (char *s);

char *tempnam (const char *dir, const char *pfx);
```

DESCRIPTION

These functions generate file names that can safely be used for a temporary file.

tmpnam always generates a file name using the path-prefix defined as P_tmpdir in the <stdio.h> header file. If *s* is NULL, tmpnam leaves its result in an internal static area and returns a pointer to that area. The next call to tmpnam will destroy the contents of the area. If *s* is not NULL, it is assumed to be the address of an array of at least L_tmpnam bytes, where L_tmpnam is a constant defined in <stdio.h>; tmpnam places its result in that array and returns *s*.

tempnam allows the user to control the choice of a directory. The argument *dir* points to the name of the directory in which the file is to be created. If *dir* is NULL or points to a string that is not a name for an appropriate directory, the path-prefix defined as P_tmpdir in the <stdio.h> header file is used. If that directory is not accessible, /tmp will be used as a last resort. This entire sequence can be up-staged by providing an environment variable TMPDIR in the user's environment, whose value is the name of the desired temporary-file directory.

Many applications prefer their temporary files to have certain favorite initial letter sequences in their names. Use the *pfx* argument for this. This argument may be NULL or point to a string of up to five characters to be used as the first few characters of the temporary-file name.

tempnam uses malloc to get space for the constructed file name, and returns a pointer to this area. Thus, any pointer value returned from tempnam may serve as an argument to free [see malloc(3C)]. If tempnam cannot return the expected result for any reason—e.g., malloc failed—or none of the above mentioned attempts to find an appropriate directory was successful, a NULL pointer will be returned.

tempnam fails if there is not enough space.

FILES

p_tmpdir and /var/tmp

SEE ALSO

creat(2), unlink(2), fopen(3S), malloc(3C), mktemp(3C), tmpfile(3S).

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 remove the file when its use is ended.

If called more than TMP_MAX (defined in stdio.h) times in a single process, these functions start recycling previously used names.

Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using these functions or mktemp and the file names are chosen to render duplication by other means unlikely.

NAME

trig: sin, sinf, cos, cosf, tan, tanf, asin, asinf, acos, acosf, atan, atanf, atan2, atan2f – trigonometric functions

SYNOPSIS

cc [*flag* ...] *file* ... -lm [*library* ...]

cc -O -Ksd [*flag* ...] *file* ... -J sfm [*library* ...]

```
#include <math.h>

double sin (double x);

float sinf (float x);

double cos (double x);

float cosf (float x);

double tan (double x);

float tanf (float x);

double asin (double x);

float asinf (float x);

double acos (double x);

float acosf (float x);

double atan (double x);

float atanf (float x);

double atan2 (double y, double x);

float atan2f (float y, float x);
```

DESCRIPTION

sin, cos, and tan and the single-precision versions sinf, cosf, and tanf return, respectively, the sine, cosine, and tangent of their argument, *x*, measured in radians.

In the following paragraphs, π represents pi.

asin and asinf return the arcsine of *x*, in the range $[-\pi/2,+\pi/2]$.

acos and acosf return the arccosine of *x*, in the range $[0,+\pi]$.

atan and atanf return the arctangent of *x*, in the range $(-\pi/2,+\pi/2)$.

atan2 and atan2f return the arctangent of *y*/*x*, in the range $(-\pi,+\pi]$, using the signs of both arguments to determine the quadrant of the return value.

SEE ALSO

matherr(3M).

DIAGNOSTICS

If the magnitude of the argument of asin, asinf, acos, or acosf is greater than 1, or if both arguments of atan2 or atan2f are 0, 0 is returned and errno is set to EDOM. In addition, a message indicating DOMAIN error is printed on the standard error output.

Except when the -Xc compilation option is used, these error-handling procedures may be changed with the function matherr. When the -Xa or -Xc compilation options are used, no error messages are printed.

NAME

tsearch, tfind, tdelete, twalk – manage binary search trees

SYNOPSIS

```
#include <search.h>

void *tsearch (const void *key, void **rootp, int (*compar)
    (const void *, const void *));

void *tfind (const void *key, void * const *rootp, int (*compar)
    (const void *, const void *));

void *tdelete (const void *key, void **rootp, int (*compar)
    (const void *, const void *));

void twalk (void *root, void(*action) (void *, VISIT, int));
```

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.

Each node of these trees contains a pointer to a datum supplied by the user. These routines manipulate trees without any knowledge of the data these pointers point to. You will need to supply two routines of your own (described later in this section) that understand the data: one for comparing two data, and another for acting on each datum during a traversal of the tree. Pointers to these routines are passed as parameters to tsearch, tfind, tdelete, and twalk.

All but twalk return pointers to nodes; everthing about the structure of a node is hidden except that its first element is the pointer to the node's datum, so that

```
*((char **) node)
```

can be used as if it were a pointer to the node's datum.

You must provide storage for a pointer, root, that these routines use to keep track of the root of the tree. It should be initialized to NULL before any routines are called.

tsearch is used to build and access the tree. key is a pointer to a datum to be accessed or stored. If a datum already in the tree is equal to *key (as determined by the user-supplied comparison routine), a pointer to its node is returned. Otherwise a node containing key is inserted into the tree, and a pointer to the new node is returned; this returned pointer will point to key, so that

```
key == *((char **) returned)
```

Only the pointer key is copied, so the calling routine must store what key points to. You must provide the root pointer, *rootp.

Like tsearch, tfind will search for a datum in the tree, returning a pointer to its node 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 searches for a datum in the tree and deletes its node if found. If the datum was not found, NULL is returned. If the datum not found, a pointer to the node's parent is returned. The arguments for tdelete are the same as for tsearch.

twalk traverses the tree. *root* points to the root of the tree to be traversed. (Any node in the tree may be used as the root for a walk beneath that node.) action is a pointer to the user-supplied action routine that will be invoked at each node. This user-supplied comparison routine is as follows:

```
compar(datum1p, datum2p)
char *datum1p;
char *datum2p;
```

The user-supplied comparison routine above returns an integer that is less than, equal to, or greater than 0, according to whether the datum pointed to by `datum1p` should be considered less than, equal to, or greater than the datum pointed to by `datum2p`. The comparison function need not compare every byte, so arbitrary information may be contained in each datum in addition to the values being compared.

The user-supplied action routine is as follows:

```
void action(node, order, level)
char *node;
VISIT order;
```

The user-supplied action routine above is called each time a node is encountered during a traversal of the tree. node is a pointer to the datum for the node; thus, if the call tsearch(key, rootp, compar) created the node, then key == *((char **) node).

The enumeration VISIT is defined in <search.h>.

`Order` is leaf if the node is a leaf; if the node is not a leaf, `order` is preorder the first time the node is encountered, postorder the second, and endorder the third time. `Level` is the level of the node in the tree, with the root being level zero.

EXAMPLES

```
#Include <string.h>
#Include <search.h>
#Include <stdio.h>

struct datum {              /*pointers to these are stored */
     char *string;               */in the tree*/
     int length;
};
char string_space [10000];  /* space to store stings */
struct datum data [500];    /* data to store */
char *root = NULL;          /* this points to root */

main()
{

    char *strptr = string_space;
    struct datum *datumptr = data;
    void print_datum(), twalk();
    int i = 0, compare_data();
    char *tsearch();

    while (gets(strptr) != NULL && i++ < 500) {
         /* set datum */
         datumptr ->string = strptr;
         datumptr ->length = strlen(strptr);
         /* put datum into the tree */
         tsearch((char *) datumptr, &root, compare_data);
         /* adjust pointers so we don't overwrite tree*/
         strptr =+ datumptr->length + 1;
         datumptr++;
}
twalk(root, print_datum);
}
/*
     This routine compares two data, based on an alphabetical
     ordering of the string field.
*/
int
compare_data(datum1, datum2)
char *datum1, *datum2;
{
     return(strcmp(((struct datum *) datum1)->string,
                  (((struct datum *) datum2)->string;
}
/*
     This routine prints out a datum the first time
     twalk encounters it.
*/
void
print_datum(datum, order, level)
char *datum;
VISIT order;
int level;
{
     if (order == preorder || order == leaf) {
          printf ("string = %20s, length = %d0
               ((struct datum *) *((char **) datum))->string,
               ((struct datum *) *((char **) datum))->length;
     }
}
```

SEE ALSO

bsearch(3C), hsearch(3C), lsearch(3C).

DIAGNOSTICS

A NULL pointer is returned by *tsearch* if there is not enough space available to create a new node.
A NULL pointer is returned by *tsearch*, *tfind*, and *tdelete* if rootp (which should be &root) is NULL.

If the datum is found both *tsearch* and *tfind* return a pointer to its node. If not, *tfind* returns NULL, and *tsearch* returns a pointer to the inserted datum's node, such that

```
     key == *((char **) returned)
```

NOTES

The root argument to *twalk* is one level of indirection less than the rootp arguments to *tsearch* and *tdelete*.

There are two nomenclatures that refer to the order in which tree nodes are visited. *tsearch* uses preorder, postorder and endorder to refer respectively to visiting a node before any of its children, after its left child and before its right, and after both its children. The alternate nomenclature uses preorder, inorder, and postorder to refer to the same visits. This could result in some confusion over the meaning of postorder.

There are two cases in which `tsearch` and `tfind` return a NULL pointer. The first case is relatively normal: `tsearch` cannot allocate more space, or `tfind` did not find the datum. The second case, that `rootp` is NULL, is not normal and should never occur unless `tsearch` and `tfind` have been called incorrectly.

`tsearch`, `tfind`, and `tdelete` return pointers to nodes, not pointers to data; the caller must dereference and cast the node pointer to get a datum pointer.

If the calling function alters the pointer to the root, results are unpredictable.

NAME

ttyname, isatty – find name of a terminal

SYNOPSIS

```
#include <stdlib.h>

char *ttyname (int fildes);

int isatty (int fildes);
```

DESCRIPTION

ttyname returns a pointer to a string containing the null-terminated path name of the terminal device associated with file descriptor *fildes*.

isatty returns 1 if *fildes* is associated with a terminal device, 0 otherwise.

FILES

/dev/*

DIAGNOSTICS

ttyname returns a NULL pointer if *fildes* does not describe a terminal device in directory /dev.

SEE ALSO

ttyslot(3C).

NOTES

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

NAME

ttyslot – find the slot in the utmp file of the current user

SYNOPSIS

```
#include <stdlib.h>

int ttyslot (void);
```

DESCRIPTION

ttyslot returns the index of the current user's entry in the /var/adm/utmp file. The returned index is accomplished by scanning files in /dev for the name of the terminal associated with the standard input, the standard output, or the standard error output (0, 1, or 2).

FILES

/var/adm/utmp

SEE ALSO

getut(3C), ttyname(3C).

DIAGNOSTICS

A value of –1 is returned if an error was encountered while searching for the terminal name or if none of the above file descriptors are associated with a terminal device.

NAME

ungetc – push character back onto input stream

SYNOPSIS

```
#include <stdio.h>

int ungetc (int c, FILE *stream);
```

DESCRIPTION

ungetc inserts the character specified by *c* (converted to an unsigned char) into the buffer associated with an input *stream* [see intro(3)]. That character, *c*, will be returned by the next getc(3S) call on that *stream*. ungetc returns *c*, and leaves the file corresponding to *stream* unchanged. A successful call to ungetc clears the EOF indicator for stream.

Four bytes of pushback are guaranteed.

The value of the file position indicator for *stream* after reading or discarding all pushed-back characters will be the same as it was before the characters were pushed back.

If *c* equals EOF, ungetc does nothing to the buffer and returns EOF.

fseek, rewind [both described on fseek(3S)], and fsetpos erase the memory of inserted characters for the stream on which they are applied.

SEE ALSO

fseek(3S), fsetpos(3C), getc(3S), setbuf(3S), stdio(3S).

DIAGNOSTICS

ungetc returns EOF if it cannot insert the character.

NAME

ungetwc – push wchar_t character back into input stream

SYNOPSIS

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

int ungetwc(wchar_t c, FILE *stream);
```

DESCRIPTION (International Functions)

ungetwc() inserts the wchar_t character *c* into the buffer associated with the input *stream*. That character, *c*, will be returned by the next *getwc* call on that stream. ungetwc() returns *c*.

One character of pushback is guaranteed, provided something has already been read from the stream and the stream is actually buffered.

If *c* equals (wchar_t) EOF, ungetwc() does nothing to the buffer and returns EOF.

fseek() erases all memory of inserted characters.

DIAGNOSTICS

ungetwc() returns EOF if it cannot insert a wchar_t character.

SEE ALSO

getwc(3W), widec(3W).
fseek(3S), setbuf(3S), stdio(3S).

NAME

`unlockpt` – unlock a pseudo-terminal master/slave pair

SYNOPSIS

```
int unlockpt(int fildes);
```

DESCRIPTION

The function `unlockpt()` clears a lock flag associated with the slave pseudo-terminal device associated with its master pseudo-terminal counterpart so that the slave pseudo-terminal device can be opened. *fildes* is a file descriptor returned from a successful open of a master pseudo-terminal device.

RETURN VALUE

Upon successful completion, the function `unlockpt()` returns `0`; otherwise it returns `-1`. A failure may occur if *fildes* is not an open file descriptor or is not associated with a master pseudo-terminal device.

SEE ALSO

`open`(2)

`grantpt`(3C), `ptsname`(3C)
in the *Programmer's Guide: STREAMS*.

NAME

vlimit – control maximum system resource consumption

SYNOPSIS

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

vlimit(resource, value)
```

DESCRIPTION

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`	Maximum number of cpu-seconds to be used by each process
`LIM_FSIZE`	Size of the largest single file that can be created
`LIM_DATA`	Maximum growth beyond the end of program text of the data+stack region via `sbrk`(2)
`LIM_STACK`	Maximum size of the automatically-extended stack region
`LIM_CORE`	Size of the largest core dump that may be created.
`LIM_MAXRSS`	Soft limit for the amount of physical memory (in bytes) to be given to the program. This information is specified for the system's benefit; if memory is tight, the system will prefer to take memory from processes that are exceeding their declared LIM_MAXRSS.

Because specifications from this call are 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 that would violate file-size limits during creation will cause a signal SIGXFSZ to be generated. This signal 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 the process time to handle the signal, it adds five seconds to the CPU time limit.

SEE ALSO

`csh`(1).

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`.

NAME

`vprintf`, `vfprintf`, `vsprintf` – print formatted output of a variable argument list

SYNOPSIS

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

int vprintf(const char *format, va_list ap);

int vfprintf(FILE *stream, const char *format, va_list ap);

int vsprintf(char *s, const char *format, va_list ap);
```

DESCRIPTION

`vprintf`, `vfprintf` and `vsprintf` are the same as `printf`, `fprintf`, and `sprintf` respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by the `<stdarg.h>` header file.

The `<stdarg.h>` header file defines the type `va_list` and a set of macros for advancing through a list of arguments whose number and types may vary. The argument *ap* to the vprint family of routines is of type `va_list`. This argument is used with the `<stdarg.h>` header file macros `va_start`, `va_arg` and `va_end` [see `va_start`, `va_arg`, and `va_end` in `stdarg`(5)]. The EXAMPLE section below shows their use with `vprintf`.

EXAMPLE

The following demonstrates how `vfprintf` could be used to write an `error` routine:

```
#include <stdio.h>
#include <stdarg.h>
/*
 *   error should be called like
 *        error(function_name, format, arg1, ...);
 */
void error(char *function_name, char *format, ...)

{
    va_list ap;

    va_start(ap, format);
    /* print out name of function causing error */
    (void) fprintf(stderr, "ERR in %s: ", function_name);
    va_arg(ap, char*);
    /* print out remainder of message */
    (void) vfprintf(stderr, format, ap);
    va_end(ap);
    (void) abort;
}
```

SEE ALSO

`printf`(3S), `stdarg`(5).

DIAGNOSTICS

`vprintf` and `vfprintf` return the number of characters transmitted, or return `-1` if an error was encountered.

NAME

vprintf, vfprintf, vsprintf – print formatted output of a variable argument list

SYNOPSIS

```
#include <stdio.h>
#include <stdarg.h>
#include <widec.h>
```

int vprintf (const char **format*, va_list *ap*);

int vfprintf (FILE **stream*, const char **format*, va_list *ap*);

int vsprintf (char **s*, const char **format*, va_list *ap*);

DESCRIPTION (International Functions)

vprintf(), vfprint(), and vsprintf() are the same as printf(), fprintf(), and sprintf() respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by the <stdarg.h> header file.

wc and ws are the new conversion specifications for wchar_t character control. Both *wc* and *ws* may be used in all three functions.

wc The wchar_t character *arg* is transformed into EUC, and then printed. If a field width is specified and the transformed EUC has fewer bytes than the field width, it will by padded to the given width. A precision specification is ignored, if specified.

ws The *arg* is taken to be a wchar_t string and the wchar_t characters from the string are transformed into EUC, and printed until a wchar_t null character is encountered or the number of bytes indicated by the precision specification is printed. If the precision specification is missing, it is taken to be infinite, and all wchar_t characters up to the first wchar_t null character are transformed into EUC and printed. If a field width is specified and the transformed EUC have fewer bytes than the field width, they are padded to the given width.

The ASCII space character (0x20) is used as a padding characters.

SEE ALSO

printf(3W), scanf(3W), stdio(3S), vprintf(3S), widec(3W), stdarg(5).

NAME

vscanf, vfscanf, vsscanf – convert formatted input using varargs argument list

SYNOPSIS

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

int vscanf (format, ap)
char *format;
va_list ap;

int vfscanf (stream, format, ap)
FILE *stream;
char *format;
va_list ap;

int vsscanf (s, format, ap)
char *s, *format;
va_list ap;
```

DESCRIPTION

Vscanf, vfscanf, and vsscanf are the same as scanf, fscanf, and sscanf respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by varargs(5).

SEE ALSO

scanf(3S), varargs(5).

NAME

vtimes – get information about resource usage

SYNOPSIS

```
vtimes(par_vm, ch_vm)
struct vtimes *par_vm, *ch_vm;
```

DESCRIPTION

vtimes returns accounting information for the current process and for the terminated child processes of the current process. *par_vm*, *ch_vm*, or both may be 0, in which case only the information for non-zero pointers is returned.

After the call, each buffer contains information as defined by the contents of the include file /usr/include/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 100ths of a second. The vm_idrss and vm_ixrss measure memory usage. They are computed by integrating the number of memory pages in use over CPU time. They are reported as though computed discretely, adding the current memory usage (in 2048 byte pages) each time the clock ticks.

For example, if a process used five main memory pages over one CPU-second for its data and stack, then vm_idsrss would have the value 5*100, where vm_utime+vm_stime would be the 100. 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 that 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 that 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

time(2), wait3(2), ftime(3C).

NAME

wconv: towupper, towlower – translate characters

SYNOPSIS

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

wchar_t towupper(wchar_t c);

wchar_t towlower(wchar_t c);
```

DESCRIPTION

If the argument to towupper() represents a lower-case letter of the ASCII or supplementary code sets, the result is the corresponding upper-case letter. If the argument to towlower() represents an upper-case letter of the ASCII or supplementary code sets, the result is the corresponding lower-case letter.

In the case of all other arguments, the return value in unchanged. The table which is used for translation is generated by wchrtbl(1M).

SEE ALSO

wchrtbl(1M), ctype(3C), wctype(3W).

NAME

wctype: iswalpha, iswupper, iswlower, iswdigit, iswxdigit, iswalnum, iswspace, iswpunct, iswprint, iswgraph, iswcntrl, iswascii, isphonogram, isideogram, isenglish, isnumber, isspecial – classify ASCII and supplemetary code set characters

SYNOPSIS

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

int iswalpha(wchar_t c);

...
```

DESCRIPTION

These functions classify character-coded `wchar_t` values by table lookup. Each is a predicate returning nonzero for true, zero for false. The lookup table is generated by `wchrtbl(1M)`. Each of these functions operates on both ASCII and supplementary code sets unless otherwise indicated.

`iswalpha(`*c*`)` *c* is an English letter.

`iswupper(`*c*`)` *c* is an English upper-case letter.

`iswlower(`*c*`)` *c* is an English lower-case letter.

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

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

`iswalnum(`*c*`)` *c* is an alphanumeric (letter or digit).

`iswspace(`*c*`)` *c* is a space character or a tab, carriage return, new line, vertical tab or form-feed.

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

`iswprint(`*c*`)` *c* is a printing character including space.

`iswgraph(`*c*`)` *c* is a printing character, like `iswprint()` except false for space.

`iswcntrl(`*c*`)` *c* is a delete character (`0177`), an ordinary control character (less than `040`) or other control character of a supplementary code set.

`iswascii(`*c*`)` *c* is an ASCII character code less than `0200`.

`isphonogram(`*c*`)` *c* is a phonogram in a supplementary code set.

`isideogram(`*c*`)` *c* is an ideogram in a supplementary code set.

`isenglish(`*c*`)` *c* is an English letters in a supplementary code set.

`isnumber(`*c*`)` *c* is a digit of a supplementary code set.

`isspecial(`*c*`)` *c* is a special character in a supplementary code set.

SEE ALSO

`wchrtbl`(1M).
`ctype`(3C) in the *System V Release 4.0 Programmer's Reference Manual.*

NAME

widec – multibyte character I/O routines

SYNOPSIS

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

DESCRIPTION (International Functions)

The functions that the multibyte character library provides for `wchar_t` string operations correspond to those provided by the `stdio`(3S) as shown in the figure below:

	character based function	*byte based function*	*character and byte based*

character I/O	`getwc`
	`getwchar`
	`fgetwc`
	`ungetwc`
	`putwc`
	`putwchar`
	`fputwc`
		`getc`
		`getchar`
		`fgetc`
		`ungetc`
		`putc`
		`putchar`
		`fputc`	

string I/O	`getws`
	`fgetws`
	`putws`
	`fputws`
		`gets`
		`fgets`
		`puts`
		`fputs`	

formatted I/O			`printf`
			`fprintf`
			`sprintf`
			`vprintf`
			`vfprintf`
			`vsprintf`
			`scanf`
			`fscanf`
			`sscanf`

The character based input and output routines provides the ability to work in units of a characters instead of bytes. C programs using these routines can handle any character, from any of the four EUC code sets as the same size by using the `wchar_t` representation.

`getwc()` returns a value of type `wchar_t`, which corresponds to the EUC representation of a character read from the input stream. `getwc()` uses the `cswidth` parameter in the *character class table* to determin the width of the character in its EUC form.

`putwc()` transforms a `wchar_t` character into the EUC, and writes it to the named output stream. `putwc()` also uses the `cswidth` parameter for determining the widths of characters in EUC.

The macros `getwchar()` and `putwchar()`; the functions `fgetwc()`, `fputwc()`, `getws()`, `fgetws()`, `putws()` and `fputws()`; and the format specifications `%wc` and `%ws` of the functions `printf()`, `fprintf()`, `sprintf()`, `vprintf()`, `vfprintf()`, `vsprintf()`, `scanf()`, `fscanf()`, and `sscanf()`; act as if they had made successive calls to either `getwc()` or `putwc()`.

The character based routines use the existing byte based routines internally, so the buffering scheme is the same.

Any program that uses these routines must include the following header files:

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

SEE ALSO

getwc(3W), getws(3W), mbchar(3W), mbstring(3W), printf(3W), putwc(3W), putws(3W), scanf(3W), ungetwc(3W), vprintf(3W), wstring(3W).

open(2), close(2), lseek(2), pipe(2), read(2), write(2), ctermid(3S), cuserid(3S), fclose(3S), ferror(3S), fopen(3S), fread(3S), fseek(3S), popen(3S), printf(3S), scanf(3S), setbuf(3S), stdio(3S), system(3S), tmpfile(3S), tmpnam(3S).

NAME

wstring: wscat, wsncat, wscmp, wsncmp, wscpy, wsncpy, wslen, wschr, wsrchr, wspbrk, wsspn, wscspn, wstok, wstostr, strtows – wchar_t string operations and type transformation

SYNOPSIS

```
#include <widec.h>

wchar_t *wscat(wchar_t *s1, wchar_t *s2);

wchar_t *wsncat(wchar_t *s1, wchar_t *s2, int n);

int wscmp(wchar_t *s1, wchar_t *s2);

int wsncmp(wchar_t *s1, wchar_t *s2, int n);

wchar_t *wscpy(wchar_t *s1, wchar_t *s2);

wchar_t *wsncpy(wchar_t *s1, wchar_t *s2, int n);

int wslen(wchar_t *s);

wchar_t *wschr(wchar_t *s, int c);

wchar_t *wsrchr(wchar_t *s, int c);

wchar_t *wspbrk(wchar_t *s1, wchar_t *s2);

int wsspn(wchar_t *s1, wchar_t *s2);

int wscspn(wchar_t *s1, wchar_t *s2);

wchar_t *wstok(wchar_t *s1, wchar_t *s2);

char *wstostr(char *s1, wchar_t *s2);

wchar_t *strtows(wchar_t *s1, char *s2);
```

DESCRIPTION (International Functions)

The arguments *s1*, *s2* and *s* point to wchar_t strings (that is, arrays of wchar_t characters terminated by a wchar_t null character). The functions wscat(), wsncat(), wscpy() and wsncpy() all modify *s1*. These functions do not check for an overflow condition of the array pointed to by *s1*.

wscat() appends a copy of the wchar_t string *s2* to the end of the wchar_t string *s1*. wsncat() appends at most *n* wchar_t characters. Each function returns *s1*.

wscmp() compares its arguments and returns an integer less than, equal to, or greater than 0, depending on whether *s1* is less than, equal to, or greater than *s2*. wsncmp() makes the same comparison but looks at most *n* wchar_t characters.

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

wslen() returns the number of wchar_t characters in *s*, not includng the termnating wchar_t null character.

wschr() [wsrchr()] returns a pointer to the first [last] occurrence of wchar_t character *c* in wchar_t string *s*, or a null pointer, if *c* does not occur in the string. The wchar_t null character terminating a string is considered to be part of the string.

`wspbrk()` returns a pointer to the first occurrence in `wchar_t` string *s1* of any `wchar_t` character from `wchar_t` string *s2*, or a null pointer if there is no `wchar_t` character from *s2* in *s1*.

`wsspn()` [`wscspn()`] returns the length of the initial segment of `wchar_t` string *s1*, which consists [does not consist] entirely of `wchar_t` characters from `wchar_t` string *s2*.

`wstok()` considers the `wchar_t` string *s1* to consist of a sequence of zero or more text tokens, separated by spans of one or more `wchar_t` characters from the separator `wchar_t` string *s2*. The first call (with the pointer *s1* specified) returns a pointer to the first `wchar_t` character of the first token, and writes a `wchar_t` null character into *s1* immediately following the returned token. The function keeps track of its position in the `wchar_t` string between separate calls, so that subsequent calls (which must be made with the first argument a null pointer) will progress through the `wchar_t` string *s1* immediately following that token. Similarly, subsequent calls will progress through the `wchar_t` string *s1* until no tokens remain. The `wchar_t` separator string *s2* may be different from call to call. A null pointer is returned when no token remains in *s1*.

`wstostr()` transforms `wchar_t` characters in `wchar_t` string *s2* into EUC, and transfers them to character string *s1*, stopping after the `wchar_t` null character has been processed.

`strtows()` transforms EUC in character string *s2* into the `wchar_t` characters, and transfers those to `wchar_t` string *s1*, stopping after the null character has been processed.

DIAGNOSTICS

On success, `wstostr()` and `strtows()` return *s1*. If an illegal byte sequence is detected, a null pointer is returned and `EILSEQ` is set to `errno`.

SEE ALSO

`malloc`(3C), `malloc`(3X), `widec`(3W).

NAME

xdr_array, xdr_bool, xdr_bytes, xdr_char, xdr_destroy, xdr_double, xdr_int, xdr_long, xdrmem_create, xdr_opaque, xdr_pointer, xdrrec_create, xdrrec_endofrecord, xdrrec_eof, xdrrec_skiprecord, xdr_reference, xdr_setpos, xdr_short, xdrstdio_create, xdr_string, xdr_u_char, xdr_u_int, xdr_u_long, xdr_u_short, xdr_union, xdr_vector, xdr_void, xdr_wrapstring – library routines for external data representation

SYNOPSIS AND DESCRIPTION

The XDR library has filter routines for strings (null-terminated arrays of bytes), structures, unions, arrays, and the basic C language data types. These routines, allow C programmers to describe arbitrary data structures in a machine-independent fashion.

XDR routines a are direction-independent; that is the same routines are called to serialize data to, or deserialize data from, an XDR stream. XDR streams are created by the use of the xdr*_create() functions, and then the pointer to these streams are passed to many of the other XDR functions.

Data for remote procedure calls are transmitted using these routines.

```
#include <rpc/rpc.h>
```

```
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 variable-length arrays and their 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 element count cannot exceed *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 one if it succeeds, zero otherwise.

```
xdr_bool(xdrs, bp)
XDR *xdrs;
bool_t *bp;
```

A filter primitive that translates between booleans (C integers) and their external representations. When encoding data, this filter produces values of either one or zero. This routine returns one if it succeeds, zero otherwise.

```
xdr_bytes(xdrs, sp, sizep, maxsize)
XDR *xdrs;
char **sp;
u_int *sizep, maxsize;
```

A filter primitive that translates between counted byte strings and their external representations. The parameter *sp* is the address of the string pointer. The length of the string is located at address *sizep*; strings cannot be longer than *maxsize*. This routine returns one if it succeeds, zero otherwise.

```
xdr_char(xdrs, cp)
XDR *xdrs;
char *cp;
```

A filter primitive that translates between C characters and their external representations. This routine returns one if it succeeds, zero 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( )`.

```
void
xdr_destroy(xdrs)
XDR *xdrs;
```

A macro that invokes 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.

```
xdr_double(xdrs, dp)
XDR *xdrs;
double *dp;
```

A filter primitive that translates between C `double` precision numbers and their external representations. This routine returns one if it succeeds, zero otherwise.

```
xdr_enum(xdrs, ep)
XDR *xdrs;
enum_t *ep;
```

A filter primitive that translates between C `enum`s (actually integers) and their external representations. This routine returns one if it succeeds, zero otherwise.

```
xdr_float(xdrs, fp)
XDR *xdrs;
float *fp;
```

A filter primitive that translates between C `float`s and their external representations. This routine returns one if it succeeds, zero 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).

```
u_int
xdr_getpos(xdrs)
XDR *xdrs;
```

A macro that invokes 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;
```

A macro that invokes 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: pointer is cast to `long *`.

Warning: `xdr_inline( )` may return NULL (0) 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.

```
xdr_int(xdrs, ip)
XDR *xdrs;
int *ip;
```

A filter primitive that translates between C integers and their external representations. This routine returns one if it succeeds, zero otherwise.

```
xdr_long(xdrs, lp)
XDR *xdrs;
long *lp;
```

A filter primitive that translates between C `long` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

```
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. The *op* determines the direction of the XDR stream (either `XDR_ENCODE`, `XDR_DECODE`, or `XDR_FREE`).

```
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 one if it succeeds, zero otherwise.

```
xdr_pointer(xdrs, objpp, objsize, xdrobj)
XDR *xdrs;
char **objpp;
u_int objsize;
xdrproc_t xdrobj;
```

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.

```
void
xdrrec_create(xdrs, sendsize, recvsize, handle, readit, writeit)
XDR *xdrs;
u_int sendsize, recvsize;
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 *sendsize*; a value of zero indicates the system should use a suitable default. The stream's data is read from a buffer of size *recvsize*; 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 the system calls `read` and `write`, 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.

Warning: this XDR stream implements an intermediate record stream. Therefore there are additional bytes in the stream to provide record boundary information.

```
xdrrec_endofrecord(xdrs, sendnow)
XDR *xdrs;
int sendnow;
```

This routine can be invoked only on streams created by `xdrrec_create( )`. 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 one if it succeeds, zero otherwise.

```
xdrrec_eof(xdrs)
XDR *xdrs;
int empty;
```

This routine can be invoked only on streams created by `xdrrec_create( )`. After consuming the rest of the current record in the stream, this routine returns one if the stream has no more input, zero otherwise.

```
xdrrec_skiprecord(xdrs)
XDR *xdrs;
```

This routine can be invoked only on streams created by `xdrrec_create( )`. It tells the XDR implementation that the rest of the current record in the stream's input buffer should be discarded. This routine returns one if it succeeds, zero otherwise.

```
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 one if it succeeds, zero otherwise.

Warning: this routine does not understand NULL pointers. Use `xdr_pointer( )` instead.

```
xdr_setpos(xdrs, pos)
XDR *xdrs;
u_int pos;
```

A macro that invokes the set position routine associated with the XDR stream *xdrs*. The parameter *pos* is a position value obtained from `xdr_getpos( )`. This routine returns one if the XDR stream could be repositioned, and zero 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.

```
xdr_short(xdrs, sp)
XDR *xdrs;
short *sp;
```

A filter primitive that translates between C `short` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

```
void
xdrstdio_create(xdrs, file, op)
XDR *xdrs;
FILE *file;
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 *file*. 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( ).

```
xdr_string(xdrs, sp, maxsize)
XDR
*xdrs;
char **sp;
u_int maxsize;
```

A filter primitive that translates between C strings and their corresponding external representations. Strings cannot be longer than *maxsize*. Note: *sp* is the address of the string's pointer. This routine returns one if it succeeds, zero otherwise.

```
xdr_u_char(xdrs, ucp)
XDR *xdrs;
unsigned char *ucp;
```

A filter primitive that translates between unsigned C characters and their external representations. This routine returns one if it succeeds, zero otherwise.

```
xdr_u_int(xdrs, up)
XDR *xdrs;
unsigned *up;
```

A filter primitive that translates between C unsigned integers and their external representations. This routine returns one if it succeeds, zero otherwise.

```
xdr_u_long(xdrs, ulp)
XDR *xdrs;
unsigned long *ulp;
```

A filter primitive that translates between C unsigned long integers and their external representations. This routine returns one if it succeeds, zero otherwise.

```
xdr_u_short(xdrs, usp)
XDR *xdrs;
unsigned short *usp;
```

A filter primitive that translates between C unsigned short integers and their external representations. This routine returns one if it succeeds, zero otherwise.

```
xdr_union(xdrs, dscmp, unp, choices, dfault)
XDR *xdrs;
int *dscmp;
char *unp;
struct xdr_discrim *choices;
bool_t (*defaultarm) (); /* may equal 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 the associated *value*, then the *proc* is called to translate the union. The end of the `xdr_discrim( )` structure array is denoted by a routine of value NULL. If the discriminant is not found in the *choices* array, then the *defaultarm* procedure is called (if it is not NULL). Returns one if it succeeds, zero otherwise.

```
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 pointer to the array, while *size* is 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 one if it succeeds, zero otherwise.

```
xdr_void( )
```

This routine always returns one. It may be passed to RPC routines that require a function parameter, where nothing is to be done.

```
xdr_wrapstring(xdrs, sp)
XDR *xdrs;
char **sp;
```

A primitive that calls **xdr_string(xdrs, sp, MAX.UNSIGNED");"** where MAX.UNSIGNED 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. Returns one if it succeeds, zero otherwise.

SEE ALSO

rpc(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 – Network Information Service client interface

SYNOPSIS AND DESCRIPTION

This package of functions provides an interface to the Network Information Service (NIS) network lookup service. 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 Network Information Service, including the definitions of map and domain, and a description of the various servers, databases, and commands that comprise the NIS.

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(3C), 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 NULL, respectively, but these two bytes are not reflected in *outkeylen* or *outvallen*. *indomain* and *inmap* strings must be non-NULL and 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 invalllen;
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

```
/usr/include/rpcsvc/ypclnt.h
/usr/include/rpcsvc/yp_prot.h
/usr/lib/libc.a
```

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

#define YPERR_RPC
2 /* RPC failure - domain

#define YPERR_DOMAIN
3 /* can't bind to server

#define YPERR_MAP
4 /* no such map in

#define YPERR_KEY
5 /* no such key in

#define YPERR_YPERR
6 /* internal yp server or

#define YPERR_RESRC
7 /* resource allocation failure */

#define YPERR_NOMORE
8 /* no more records in

#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

#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 */

SEE ALSO

`malloc`(3C), `ypupdate`(3N), `ypfiles`(5), `ypserv`(8).

End of Chapter

Index

Note: Boldfaced page numbers (e.g., **1-5**) indicate definitions of terms or other key information.

A

a641(3C) **3-9**
abort(3C) **3-10**
abs(3C) **3-11**
addseverity(3C) **3-12**
assert(3X) **3-13**
atexit(3C) **3-14**

B

basename(3G) **3-15**
bcmp(3C) **3-16**
bcopy(3C) **3-17**
berk_regex(3C) **3-18**
berk_signal(3C) **3-19**
bessel(3M) **3-21**
bgets(3G) **3-22**
bsearch(3C) **3-23**
bufsplit(3G) **3-25**
byteorder(3N) **3-26**
bzero(3C) **3-27**

C

catgets(3C) **3-28**
catopen(3C) **3-29**
cfsetospeed(3C) **3-31**
clock(3C) **3-33**
conv(3C) **3-34**
copylist(3G) **3-35**
crypt(3C) **3-36**
crypt(3X) **3-37**
ctermid(3S) **3-39**
ctime(3C) **3-40**
ctype(3C) **3-43**
curs_addch(3X) **3-45**
curs_addchstr(3X) **3-47, 3-48**
curs_addstr(3X) **3-49**
curs_addwch(3X) **3-50**
curs_addwchstr(3X) **3-51**
curs_attr(3X) **3-52**
curs_beep(3X) **3-54**
curs_bkgd(3X) **3-55**
curs_border(3X) **3-56**
curs_clear(3X) **3-57**
curs_color(3X) **3-58**
curs_delch(3X) **3-60**
curs_deleteln(3X) **3-61**
curs_getch(3X) **3-62**
curs_getstr(3X) **3-66**
curs_getwch(3X) **3-67**
curs_getwstr(3X) **3-71**
curs_getyx(3X) **3-72**
curs_inch(3X) **3-73**
curs_inchstr(3X) **3-74**
curs_initscr(3X) **3-75**
curs_inopts(3X) **3-77**
curs_insch(3X) **3-80**
curs_insstr(3X) **3-81**
curs_instr(3X) **3-82**
curs_inswch(3X) **3-83**
curs_inswstr(3X) **3-84**
curs_inwch(3X) **3-85**
curs_inwchstr(3X) **3-86**
curs_inwstr(3X) **3-87**
curs_kernel(3X) **3-88**
curs_move(3X) **3-90**
curs_outopts(3X) **3-91**
curs_overlay(3X) **3-93**
curs_pad(3X) **3-94**
curs_printw(3X) **3-96**
curs_refresh(3X) **3-97**
curs_scanw(3X) **3-99**
curs_scr_dump(3X) **3-100**
curs_scroll(3X) **3-101**
curs_slk(3X) **3-102**
curs_termattrs(3X) **3-104**
curs_termcap(3X) **3-106**
curs_terminfo(3X) **3-107**
curs_touch(3X) **3-110**
curs_util(3X) **3-111**
curs_window(3X) **3-113**

curses(3X) **3-115**
cuserid(3S) **3-125**

D

dbm(3X) **3-126**
dg_flock(3C) **3-128**
dg_seek(3C) **3-130**
dial(3C) **3-131**
directory(3X) **3-133**
dirname(3G) **3-135**
div(3C) **3-136**
doconfig(3N) **3-137**
Documention
  AViiON and DG/UX, Guide to RD-1
  related **RD-1**
drand48(3C) **3-139**
drem(3M) **3-141**

E

ecvt(3C) **3-142**
elf(3E) **3-143**
elf_begin(3E) **3-147**
elf_cntl(3E) **3-152**
elf_end(3E) **3-153**
elf_error(3E) **3-154**
elf_fill(3E) **3-155**
elf_flag(3E) **3-156**
elf_fsize(3E) **3-158**
elf_getarhdr(3E) **3-159**
elf_getarsym(3E) **3-160**
elf_getbase(3E) **3-161**
elf_getdata(3E) **3-162**
elf_getehdr(3E) **3-166**
elf_getident(3E) **3-167**
elf_getphdr(3E) **3-168**
elf_getscn(3E) **3-169**
elf_getshdr(3E) **3-171**
elf_hash(3E) **3-172**
elf_kind(3E) **3-173**
elf_next(3E) **3-174**
elf_rand(3E) **3-175**
elf_rawfile(3E) **3-176**
elf_strptr(3E) **3-177**
elf_update(3E) **3-178**
elf_version(3E) **3-181**
elf_xlate(3E) **3-182**
end(3C) **3-184**
erf(3M) **3-185**
ethers(3N) **3-186**
exp(3M) **3-188**
exportent(3C) **3-190**
extended_perror(3C) **3-192**
extended_strerror(3C) **3-194**

F

fattach(3C) **3-195**
fclose(3S) **3-196**
fdetach(3C) **3-197**
ferror(3S) **3-198**
ffs(3C) **3-199**
floor(3M) **3-200**
fmtmsg(3C) **3-201**
fopen(3S) **3-205**
form_cursor(3X) **3-207**
form_data(3X) **3-208**
form_driver(3X) **3-209**
form_field(3X) **3-211**
form_field_attributes(3X) **3-212**
form_field_buffer(3X) **3-213**
form_field_info(3X) **3-214**
form_field_just(3X) **3-215**
form_field_new(3X) **3-216**
form_field_opts(3X) **3-217**
form_field_userptr(3X) **3-218**
form_field_validation(3X) **3-219**
form_fieldtype(3X) **3-220**
form_hook(3X) **3-222**
form_new(3X) **3-223**
form_new_page(3X) **3-224**
form_opts(3X) **3-225**
form_page(3X) **3-226**
form_post(3X) **3-227**
form_userptr(3X) **3-228**
form_win(3X) **3-229**
forms(3X) **3-230**
fpgetround(3C) **3-233**
fread(3S) **3-235**
frexp(3C) **3-236**
fseek(3S) **3-238**
ftime(3C) **3-239**
ftw(3C) **3-240**

G

gamma(3M) **3-242**
getc(3S) **3-243**
getcwd(3C) **3-244**

getdate(3C) **3-245**
getenv(3C) **3-248**
getfsent(3C) **3-249**
getgrent(3C) **3-250**
gethostent(3N) **3-252**
getlogin(3C) **3-254**
getmntent(3C) **3-255**
getnetconfig(3N) **3-256**
getnetent(3N) **3-258**
getnetgrent(3N) **3-260**
getnetpath(3N) **3-261**
getopt(3C) **3-262**
getpass(3C) **3-264**
getprotoent(3N) **3-265**
getpw(3C) **3-266**
getpwent(3C) **3-267**
getrpcent(3N) **3-269**
getrpcport(3R) **3-270**
gets(3S) **3-271**
getservent(3N) **3-272**
getspent(3C) **3-274**
getsubopt(3C) **3-276**
gettxt(3C) **3-279**
getut(3C) **3-281**
getwc(3W) **3-283**
getwd(3C) **3-284**
getwidth(3W) **3-285**
getws(3W) **3-286**
gmatch(3G) **3-287**
grantpt(3C) **3-288**

H

hsearch(3C) **3-289**
hypot(3M) **3-291**

I

ieeefp(3C) **3-292**
index(3C) **3-293**
inet(3N) **3-295**
initgroups(3C) **3-297**
insque(3C) **3-298**
intro(3) **3-2**
intro(3N) **3-8**
isalphanum(3C) **3-299**
isastream(3C) **3-300**
isencrypt(3G) **3-301**
ishex(3C) **3-302**
isnan(3C) **3-303**
itoa(3C) **3-304**

J

jobs(3C) **3-305**

L

l3tol(3C) **3-310**
ldahread(3X) **3-311**
ldclose(3X) **3-312**
ldfhread(3X) **3-313**
ldgetname(3X) **3-314**
ldlread(3X) **3-315**
ldlseek(3X) **3-316**
ldohseek(3X) **3-317**
ldopen(3X) **3-318**
ldrseek(3X) **3-320**
ldshread(3X) **3-321**
ldsseek(3X) **3-322**
ldtbindex(3X) **3-323**
ldtbread(3X) **3-324**
ldtbseek(3X) **3-325**
Line control 3-547
localeconv(3C) **3-326**
lockf(3C) **3-329**
logname(3X) **3-331**
lsearch(3C) **3-332**

M

main(3C) **3-334**
malloc(3C) **3-336**
malloc(3X) **3-338**
matherr(3M) **3-340**
mbchar(3C) **3-343**
mbchar(3W) **3-345**
mbstring(3C) **3-346**
mbstring(3W) **3-347**
memory(3C) **3-348**
menu_attributes(3X) **3-349**
menu_cursor(3X) **3-350**
menu_driver(3X) **3-351**
menu_format(3X) **3-353**
menu_hook(3X) **3-354**
menu_item_current(3X) **3-355**
menu_item_name(3X) **3-356**
menu_item_new(3X) **3-357**
menu_item_opts(3X) **3-358**
menu_item_userptr(3X) **3-359**

menu_item_value(3X) **3-360**
menu_item_visible(3X) **3-361**
menu_items(3X) **3-362**
menu_mark(3X) **3-363**
menu_new(3X) **3-364**
menu_opts(3X) **3-365**
menu_pattern(3X) **3-366**
menu_post(3X) **3-367**
menu_userptr(3X) **3-368**
menu_win(3X) **3-369**
menus(3X) **3-370**
mkdirp(3G) **3-373**
mkfifo(3C) **3-374**
mkstemp(3C) **3-375**
mktemp(3C) **3-376**
mlock(3C) **3-377**
mlockall(3C) **3-378**
monitor(3C) **3-379**
mp(3X) **3-381**
msync(3C) **3-383**

N

ndbm(3C) **3-384**
netdir(3N) **3-386**
Networking functions, *see* xdr(3N)
nl_langinfo(3C) **3-390**
nlist(3C) **3-391**
nlsgetcall(3N) **3-392**
nlsprovider(3N) **3-393**
nlsrequest(3N) **3-394**

P

p2open(3G) **3-395**
panel_above(3X) **3-397**
panel_move(3X) **3-398**
panel_new(3X) **3-399**
panel_show(3X) **3-400**
panel_top(3X) **3-401**
panel_update(3X) **3-402**
panel_userptr(3X) **3-403**
panel_window(3X) **3-404**
panels(3X) **3-405**
pathfind(3G) **3-406**
perror(3C) **3-407**
popen(3S) **3-408**
printf(3S) **3-410**
printf(3W) **3-415**
psignal(3C) **3-416**
ptsname(3C) **3-417**
putc(3S) **3-418**
putenv(3C) **3-419**
putpwent(3C) **3-420**
puts(3S) **3-421**
putspent(3C) **3-422**
putwc(3W) **3-423**
putws(3W) **3-424**

Q

qsort(3C) **3-425**

R

raise(3C) **3-426**
rand(3C) **3-427**
random(3C) **3-428**
rcmd(3X) **3-430**
re_comp(3C) **3-18**
re_exec(3C) **3-18**
realpath(3C) **3-431**
regcmp(3G) **3-432**
regcmp(3X) **3-434**
regex(3C) **3-18**
regexpr(3G) **3-437**
Related documents **RD-1**
remove(3C) **3-440**
remque(3C) **3-441**
resolver(3C) **3-443**
rexec(3X) **3-445**
rindex(3C) **3-446**
rpc(3N) **3-448**
rtime(3N) **3-464**

S

scandir(3C) **3-465**
scanf(3S) **3-466**
scanf(3W) **3-470**
setbuf(3S) **3-472**
setbuffer(3C) **3-473**
setjmp(3C) **3-474**
setlinebuf(3C) **3-476**
setlocale(3C) **3-477**
signal(3C) **3-19**
sigsetjmp(3C) **3-479**
sigsetops(3C) **3-480**
sinh(3M) **3-481**
sleep(3C) **3-482**

sputl(3X) **3-483**
ssignal(3C) **3-484**
stdio(3S) **3-485**
stdipc(3C) **3-487**
str(3G) **3-488**
strccpy(3G) **3-489**
strcoll(3C) **3-490**
strerror(3C) **3-491**
strftime(3C) **3-492**
string(3C) **3-494**
strsave(3C) **3-496**
strtod(3C) **3-498**
strtol(3C) **3-499**
strxfrm(3C) **3-500**
swab(3C) **3-501**
swapcontext(3C) **3-502**
syslog(3C) **3-503**
system(3S) **3-505**
sysv3_cuserid(3S) **3-506**

T

t_accept(3N) **3-507**
t_alloc(3N) **3-509**
t_bind(3N) **3-511**
t_close(3N) **3-513**
t_connect(3N) **3-514**
t_error(3N) **3-516**
t_free(3N) **3-517**
t_getinfo(3N) **3-518**
t_getstate(3N) **3-520**
t_listen(3N) **3-521**
t_look(3N) **3-523**
t_open(3N) **3-524**
t_optmgmt(3N) **3-526**
t_rcv(3N) **3-528**
t_rcvconnect(3N) **3-529**
t_rcvdis(3N) **3-531**
t_rcvrel(3N) **3-533**
t_rcvudata(3N) **3-534**
t_rcvuderr(3N) **3-536**
t_snd(3N) **3-537**
t_snddis(3N) **3-539**
t_sndrel(3N) **3-541**
t_sndudata(3N) **3-542**
t_sync(3N) **3-544**
t_unbind(3N) **3-546**
tcflush(3C) **3-547**
tcgetpgrp(3C) **3-550**
tcsetattr(3C) **3-551**
tcsetpgrp(3C) **3-553**
termcap(3X) **3-554**
termios(3C) **3-556**
tmpfile(3S) **3-560**
tmpnam(3S) **3-561**
trig(3M) **3-562**
tsearch(3C) **3-563**
ttyname(3C) **3-567**
ttyslot(3C) **3-568**

U

ungetc(3S) **3-569**
ungetwc(3W) **3-570**
unlockpt(3C) **3-571**

V

vlimit(3C) **3-572**
vprintf(3S) **3-573**
vprintf(3W) **3-574**
vscanf(3S) **3-575**
vtimes(3C) **3-576**

W

wconv(3W) **3-577**
wctype(3W) **3-578**
widec(3W) **3-579**
wstring(3W) **3-581**

X

xdr(3N) **3-583**

Y

ypclnt(3N) **3-590**

Related Documents

The following list of related manuals gives titles of Data General manuals followed by nine-digit numbers used for ordering. You can order any of these manuals via mail or telephone (see the TIPS Order Form in the back of this manual).

For a complete list of AViiON® and DG/UX™ manuals, see the *Guide to AViiON® and DG/UX™ Documentation* (069-701085). The on-line version of this manual found in **/usr/release/doc_guide** contains the most current list.

Data General Software Manuals

User's Manuals

User's Reference for the DG/UX™ System
Contains an alphabetical listing of manual pages for commands relating to general system operation. Ordering Number — 093-701054

Using the DG/UX™ Editors
Describes the text editors **vi** and **ed**, the batch editor **sed**, and the command line editor **editread**. Ordering Number — 069-701036

Using the DG/UX™ System
Describes the DG/UX system and its major features, including the C and Bourne shells, typical user commands, the file system, and communications facilities such as **mailx**. Ordering Number — 069-701035

Installation and Administration Manuals

System Manager's Reference for the DG/UX™ System
Contains an alphabetical listing of manual pages for commands relating to system administration or operation. Ordering Number — 093-701050

Programming Manuals

Porting and Developing Applications on the DG/UX™ System
A compendium of useful information for experienced programmers developing or porting applications to the DG/UX™ system. It includes information on how to: set up your environment, use the software development tools, compile and link programs, port to the windowing environment, and build BCS applications. It also describes available debuggers and the various industry standards the DG/UX system supports. Ordering Number — 069-701059

Programmer's Guide: ANSI C and Programming Support Tools (UNIX System V Release 4)
Describes the standard tools of the UNIX program development environment including compiling, linking, debugging, and analysis and revision control. An accompanying supplement, *Supplement for Programmer's Guide: ANSI C and Programming Support Tools* (086-000180) describes the DG/UX system enhancements and differences. Ordering Number — 093-701104

Programmer's Guide: Systems Services and Application Packaging Tools (UNIX System V Release 4)
Describes standard programming procedures and interfaces available to the C application developer in the UNIX environment. Topics include interprocess communications, memory management, file and record locking and application packaging. **Note:** Chapters 5 and 9 of this Prentice Hall manual discuss topics that do not apply to the DG/UX system. Ordering Number — 093-701103

Programmer's Reference for the DG/UX™ System, (Volume 1)
Alphabetical listing of manual pages for DG/UX programming commands and system calls. This is part of a three-volume set. Ordering Number — 093-701055

Programmer's Reference for the DG/UX™ System, (Volume 3)
Alphabetical listing of manual pages for DG/UX file formats, miscellaneous features, and networking protocols. Part of a three-volume set, this volume contains the table of contents and index (**contents** (0) and **index** (0)) for man pages. Ordering Number — 093-701102

Writing a Standard Device Driver for the DG/UX™ System
Describes how to write a device driver for a DG/UX system running on an AViiON computer. Describes the drivers written to address specific devices or adapters that manage secondary bus access to specific devices. Information on kernel-level programming in the DG/UX system and descriptions of important kernel-level utility routines are found in *Programming in the DG/UX™ Kernel Environment* (093-701083). Ordering Number — 093-701053

End of Related Documents

TIPS ORDERING PROCEDURES

TO ORDER

1. An order can be placed with the TIPS group in two ways:

   a) MAIL ORDER – Use the order form on the opposite page and fill in all requested information. Be sure to include shipping charges and local sales tax. If applicable, write in your tax exempt number in the space provided on the order form.

   Send your order form with payment to:

   Data General Corporation
   ATTN: Educational Services/TIPS G155
   4400 Computer Drive
   Westboro, MA 01581–9973

   b) TELEPHONE – Call TIPS at (508) 870–1600 for all orders that will be charged by credit card or paid for by purchase orders over $50.00. Operators are available from 8:30 AM to 5:00 PM EST.

METHOD OF PAYMENT

2. As a customer, you have several payment options:

   a) Purchase Order – Minimum of $50. If ordering by mail, a hard copy of the purchase order must accompany order.

   b) Check or Money Order – Make payable to Data General Corporation.

   c) Credit Card – A minimum order of $20 is required for Mastercard or Visa orders.

SHIPPING

3. To determine the charge for UPS shipping and handling, check the total quantity of units in your order and refer to the following chart:

Total Quantity	Shipping & Handling Charge
1–4 Units	$5.00
5–10 Units	$8.00
11–40 Units	$10.00
41–200 Units	$30.00
Over 200 Units	$100.00

If overnight or second day shipment is desired, this information should be indicated on the order form. A separate charge will be determined at time of shipment and added to your bill.

VOLUME DISCOUNTS

4. The TIPS discount schedule is based upon the total value of the order.

Order Amount	Discount
$1–$149.99	0%
$150–$499.99	10%
Over $500	20%

TERMS AND CONDITIONS

5. Read the TIPS terms and conditions on the reverse side of the order form carefully. These must be adhered to at all times.

DELIVERY

6. Allow at least two weeks for delivery.

RETURNS

7. Items ordered through the TIPS catalog may not be returned for credit.
8. Order discrepancies must be reported within 15 days of shipment date. Contact your TIPS Administrator at (508) 870–1600 to notify the TIPS department of any problems.

INTERNATIONAL ORDERS

9. Customers outside of the United States must obtain documentation from their local Data General Subsidiary or Representative. Any TIPS orders received by Data General U.S. Headquarters will be forwarded to the appropriate DG Subsidiary or Representative for processing.

TIPS ORDER FORM

Mail To: Data General Corporation
Attn: Educational Services/TIPS G155
4400 Computer Drive
Westboro, MA 01581 - 9973

BILL TO:	SHIP TO: (No P.O. Boxes - Complete Only If Different Address)
COMPANY NAME ____	COMPANY NAME ____
ATTN: ____	ATTN: ____
ADDRESS ____	ADDRESS (NO PO BOXES) ____
CITY ____	CITY ____
STATE ____ ZIP ____	STATE ____ ZIP ____

Priority Code ____ (See label on back of catalog)

Authorized Signature of Buyer (Agrees to terms & conditions on reverse side) ____ Title ____ Date ____ Phone (Area Code) ____ Ext. ____

ORDER #	QTY	DESCRIPTION	UNIT PRICE	TOTAL PRICE

A SHIPPING & HANDLING

□ UPS	ADD
1-4 Items	$ 5.00
5-10 Items	$ 8.00
11-40 Items	$ 10.00
41-200 Items	$ 30.00
200+ Items	$100.00

Check for faster delivery

Additional charge to be determined at time of shipment and added to your bill.

□ UPS Blue Label (2 day shipping)

□ Red Label (overnight shipping)

B VOLUME DISCOUNTS

Order Amount	Save
$0 - $149.99	0%
$150 - $499.99	10%
Over $500.00	20%

Tax Exempt # or Sales Tax (if applicable) ____

ORDER TOTAL	
Less Discount See B	-
SUB TOTAL	
Your local* sales tax	+
Shipping and handling - See A	+
TOTAL - See C	

C PAYMENT METHOD

□ Purchase Order Attached ($50 minimum)
P.O. number is ____. (Include hardcopy P.O.)
□ Check or Money Order Enclosed
□ Visa □ MasterCard ($20 minimum on credit cards)

Account Number □□□□□□□□□□□□□□□□

Expiration Date □□□□

Authorized Signature ____
(Credit card orders without signature and expiration date cannot be processed.)

THANK YOU FOR YOUR ORDER

PRICES SUBJECT TO CHANGE WITHOUT PRIOR NOTICE.
PLEASE ALLOW 2 WEEKS FOR DELIVERY.
NO REFUNDS NO RETURNS.

* Data General is required by law to collect applicable sales or use tax on all purchases shipped to states where DG maintains a place of business, which covers all 50 states. Please include your local taxes when determining the total value of your order. If you are uncertain about the correct tax amount, please call 508-870-1600.

Form 702
Rev. 8/87

DATA GENERAL CORPORATION .
TECHNICAL INFORMATION AND PUBLICATIONS SERVICE
TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form. These terms and conditions apply to all orders, telephone, telex, or mail. By accepting these products the Customer accepts and agrees to be bound by these terms and conditions.

1. CUSTOMER CERTIFICATION

Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

2. TAXES

Customer shall be responsible for all taxes, including taxes paid or payable by DGC for products or services supplied under this Agreement, exclusive of taxes based on DGC's net income, unless Customer provides written proof of exemption.

3. DATA AND PROPRIETARY RIGHTS

Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer sh abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Custome and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any dat by itself does not constitute or evidence a publication or public disclosure.

4. LIMITED MEDIA WARRANTY

DGC warrants the CLI Macros media, provided by DGC to the Customer under this Agreement, against physical defects for period of ninety (90) days from the date of shipment by DGC. DGC will replace defective media at no charge to you, provide it is returned postage prepaid to DGC within the ninety (90) day warranty period. This shall be your exclusive remedy and DGC's sole obligation and liability for defective media. This limited media warranty does not apply if the media has been damaged by accident, abuse or misuse.

5. DISCLAIMER OF WARRANTY

EXCEPT FOR THE LIMITED MEDIA WARRANTY NOTED ABOVE, DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS, CLI MACROS OR MATERIALS SUPPLIED HEREUNDER.

6. LIMITATION OF LIABILITY

A. CUSTOMER AGREES THAT DGC'S LIABILITY, IF ANY, FOR DAMAGES, INCLUDING BUT NOT LIMITED TO LIABILITY ARISING OUT OF CONTRACT, NEGLIGENCE, STRICT LIABILITY IN TORT OR WARRANTY SHALL NOT EXCEED THE CHARGES PAID BY CUSTOMER FOR THE PARTICULAR PUBLICATION OR CLI MACRO INVOLVED. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO CLAIMS FOR PERSONAL INJURY CAUSED SOLELY BY DGC'S NEGLIGENCE. OTHER THAN THE CHARGES REFERENCED HEREIN, IN NO EVENT SHALL DGC BE LIABL FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOST PROFITS AND DAMAGES RESULTING FROM LOSS OF USE, OR LOST DATA, OR DELIVERY DELAYS, EVEN IF DGC HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILIT THEREOF; OR FOR ANY CLAIM BY ANY THIRD PARTY.

B. ANY ACTION AGAINST DGC MUST BE COMMENCED WITHIN ONE (1) YEAR AFTER THE CAUSE OF ACTION ACCRUES.

7. GENERAL

A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts, excluding its conflict law rules. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer. DGC hereby rejects all such different, conflicting, or additional terms.

8. IMPORTANT NOTICE REGARDING AOS/VS INTERNALS SERIES (ORDER #1865 & #1875)

Customer understands that information and material presented in the AOS/VS Internals Series documents may be specific t a particular revision of the product. Consequently user programs or systems based on this information and material may be revision-locked and may not function properly with prior or future revisions of the product. Therefore, Data General makes r representations as to the utility of this information and material beyond the current revision level which is the subject of the manual. Any use thereof by you or your company is at your own risk. Data General disclaims any liability arising from any su use and I and my company (Customer) hold Data General completely harmless therefrom.

Cut here and insert in binder spine pocket

Data General

Data General Corporation, Westboro, Massachusetts 01580