HP-UX Reference

Vol 2: Sections 2 and 3

HP 9000 Series 300/800 Computers
HP-UX Release 7.0

HP Part Number 09000-90013

HEWLETT
PACKARD

Hewlett-Packard Company
3404 East Harmony Road, Fort Collins, Colorado 80525

Legal Notices

The information contained in this document is subject to change without notice.

Hewlett-Packard Company makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard Company shall not be liable for errors contained herein or direct, indirect, special, incidental, or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty: A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the U.S. Government Department of Defense is subject to restrictions as set forth in paragraph (b)(3)(ii) of the Rights in Technical Data and Software clause in FAR 52.227-7013.

This software and documentation is based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California.

Printing History

The manual printing date and part number indicate its current edition. The printing date will change when a new edition is printed. However, minor changes may be made at reprint without changing the printing date. The manual part number will change when extensive changes are made.

To ensure that you receive new editions of this manual when changes occur, you may subscribe to the appropriate product support service, available through your HP sales representative.

September 1989. First Edition. This manual replaces manual part number 09000-90009, and is valid for HP-UX Release 7.0 on both Series 300 and Series 800 systems.

Notes

Table of Contents for Volume 2

Table of Contents
Volume 2

Section 2: System Calls

Section 3: System Calls

Entry Name(Section) *name* **Description**

Entry Name(Section) *name* **Description**

Entry Name(Section) *name* **Description**

Entry Name(Section) *name* **Description**

Section 2: System Calls

NAME

intro – introduction to system calls

DESCRIPTION

This section describes all of the system calls. All of these calls return a function result. This result indicates the status of the call. Typically, a zero or positive result indicates that the call completed successfully, and −1 indicates an error. The individual descriptions specify the details. An error number is also made available in the external variable **errno** (see *errno*(2)). Note: **Errno** is not cleared on successful calls, so it should be tested only after an error has been indicated.

SEE ALSO

intro(3), errno(2), hier(5).

The introduction to this manual.

NAME

access – determine accessibility of a file

SYNOPSIS

```
#include <unistd.h>

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

DESCRIPTION

Path points to a path name naming a file. *Access* checks the named file for accessibility according to the bit pattern contained in *amode*, using the real user ID instead of the effective user ID and the real group ID instead of the effective group ID. The value of *amode* is either the bitwise inclusive OR of the access permissions to be checked or the existence test. The following symbolic constants, defined in <**unistd.h**>, test for permissions:

R_OK	read
W_OK	write
X_OK	execute (search)
F_OK	check existence of file

Access Control Lists (ACLs)

Read, write and execute/search permissions are checked against the file's access control list. Each mode is checked separately since different ACL entries might grant different permissions. The real user ID is combined with the process's real group ID and each group in its supplementary groups list, and the access control list is searched for a match. Search proceeds in order of specificity and ends when one or more matching entries are found at a specific level. More than one *u.g* or *%.g* entry can match a user if that user has a non-null supplementary groups list. If any matching entry has the appropriate permission bit set, access is permitted.

Access reports that a shared text file currently open for execution is not writable, regardless of its access control list. It also reports that a file on a read-only file system is not writable. However, *access* does not report that a shared text file open for writing is not executable, since the check is not easily done.

RETURN VALUE

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

ERRORS

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

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	Read, write, or execute (search) permission is requested for a null path name.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[EROFS]	Write access is requested for a file on a read-only file system.
[ETXTBSY]	Write access is requested for a pure procedure (shared text) file that is being executed.
[EACCES]	The access control list does not permit the requested access and the real user ID is not the superuser.
[EFAULT]	*Path* points outside the allocated address space for the process. The reliable detection of this error will be implementation dependent.

[ELOOP] Too many symbolic links were encountered in translating the path name.

[ENAMETOOLONG]
The length of the specified path name exceeds PATH_MAX bytes, or the length of a component of the path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.

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

Access reports that a file currently open for execution is not writable, regardless of the setting of its mode.

WARNINGS

If the path is valid and the real user ID is super-user, *access* always returns 0.

SEE ALSO

chmod(2), setacl(2), stat(2), acl(5), unistd(5).

STANDARDS CONFORMANCE

access: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

acct – enable or disable process accounting

SYNOPSIS

int acct (path)
char *path;

DESCRIPTION

Acct is used to enable or disable the system's process accounting routine. If the routine is enabled, an accounting record will be written on an accounting file for each process that terminates. Termination can be caused by one of two things: an *exit* call or a signal; see *exit*(2) and *signal*(5). The effective user ID of the calling process must be super-user to use this call.

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

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

When the amount of free space on the file system containing the accounting file falls below a configurable threshold, the system prints a message on the console and disables process accounting. Another message is printed and the process accounting is re-enabled when the space reaches a second configurable threshold.

RETURN VALUE

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

ERRORS

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

[EPERM] The effective user ID of the calling process is not super-user.

[EBUSY] An attempt is being made to enable accounting when it is already enabled.

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] One or more components of the accounting file path name do not exist.

[EACCES] The file named by *path* is not an ordinary file.

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

[EFAULT] *Path* points to an illegal address. The reliable detection of this error will be implementation dependent.

[ETXTBSY] *Path* points to a text file which is currently open.

[ENAMETOOLONG]
The accounting file path name exceeds PATH_MAX bytes, or the length of a component of the path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.

[ELOOP] Too many symbolic links were encountered in translating the path name.

DEPENDENCIES

Series 300

The system's process accounting routine will ignore any locks placed on the process accounting file.

If the size of the process accounting file reaches 5000 blocks, records for processes terminating after that point will be silently lost. However, in that case the *turnacct* command would still sense that process accounting is still enabled. This loss of records can be prevented by the use of *ckpacct* (see *acctsh*(1M)).

SEE ALSO

acct(1M), acctsh(1M), exit(2), acct(4), signal(5).

STANDARDS CONFORMANCE

acct: SVID2, XPG2

NAME

alarm – set a process's alarm clock

SYNOPSIS

unsigned long alarm (sec)
unsigned long sec;

DESCRIPTION

Alarm instructs the alarm clock of the calling process to send the signal SIGALRM to the calling process after the number of real-time seconds specified by *sec* have elapsed; see *signal*(5). Specific implementations might place limitations of the maximum alarm time supported. The constant MAX_ALARM defined in <*sys/param.h*> specifies the implementation-specific maximum. Whenever *sec* is greater that this maximum, it is silently rounded down to it. On all implementations, MAX_ALARM is guaranteed to be at least 31 days (in seconds).

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

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

Alarms are not inherited by a child process across a *fork*, but are inherited across an *exec*.

On systems that support the *getitimer*(2) and *setitimer* system calls, the timer mechanism used by *alarm* is the same as that used by *ITIMER_REAL.* Thus successive calls to *alarm, getitimer,* and *setitimer* set and return the state of a single timer. In addition, *alarm* sets the timer interval to zero.

RETURN VALUE

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

WARNINGS

In some implementations, error bounds for alarm are -1, +0 seconds (for the posting of the alarm, not the restart of the process). Thus a delay of 1 second can return immediately. The *setitimer* routine can be used to create a more precise delay.

SEE ALSO

sleep(1), exec(2), getitimer(2), pause(2), signal(5), sleep(3C).

STANDARDS CONFORMANCE

alarm: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

atexit – register a function to be called at program termination

SYNOPSIS

#include <stdlib.h>

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

DESCRIPTION

Atexit registers the function *func* to be called, without arguments, at normal program termination. Functions registered by *atexit* are called in reverse order of registration.

An *atexit* call during exit processing is always unsuccessful.

The number of registered functions should not exceed ATEXIT_MAX as specified in **<limits.h>**.

RETURN VALUE

Atexit returns zero if the registration is successful; non-zero if unsuccessful.

SEE ALSO

exit(2).

STANDARDS CONFORMANCE

atexit: ANSI C

NAME

audctl – start or halt the auditing system and set or get audit files

SYNOPSIS

#include <sys/audit.h>

audctl(cmd, cpath, npath, mode)
char *cpath, *npath;
int cmd, mode;

DESCRIPTION

Audctl sets or gets the auditing system "current" and "next" audit files, and starts or halts the auditing system. This call is restricted to superusers. *Cpath* and *npath* hold the absolute path names of the "current" and "next" files. *Mode* specifies the audit file's permission bits. *cmd* is one of the following specifications:

AUD_ON The caller issues the **AUD_ON** command with the required "current" and "next" files to turn on the auditing system. If the auditing system is currently off, it is turned on; the file specified by the *cpath* parameter is used as the "current" audit file, and the file specified by the *npath* parameter is used as the "next" audit file. If the audit files do not already exist, they are created with the *mode* specified. The auditing system then begins writing to the specified "current" file. An empty string or NULL *npath* can be specified, if the caller wants to designate that no "next" file be available to the auditing system. If the auditing system is already on, no action is performed; **−1** is returned and **errno** is set to EBUSY.

AUD_GET The caller issues the **AUD_GET** command to retrieve the names of the "current" and "next" audit files. If the auditing system is on, the names of the "current" and "next" audit files are returned via the *cpath* and *npath* parameters (which must point to character buffers of sufficient size to hold the file names). *Mode* is ignored. If the auditing system is on and there is no available "next" file, the "current" audit file name is returned via the *cpath* parameter, *npath* is set to an empty string; **−1** is returned, and *errno* is set to ENOENT. If the auditing system is off, no action is performed; **−1** is returned and **errno** is set to EALREADY.

AUD_SET The caller issues the **AUD_SET** command to change both the "current" and "next" files. If the audit system is on, the file specified by *cpath* is used as the "current" audit file, and the file specified by *npath* is used as the "next" audit file. If the audit files do not already exist, they are created with the specified *mode*. The auditing system begins writing to the specified "current" file. Either an empty string or NULL *npath* can be specified if the caller wants to designate that no "next" file be available to the auditing system. If the auditing system is off, no action is performed; **−1** is returned and **errno** is set to EALREADY.

AUD_SETCURR The caller issues the **AUD_SETCURR** command to change only the "current" audit file. If the audit system is on, the file specified by *cpath* is used as the "current" audit file. If the specified "current" audit file does not exist, it is created with the specified *mode*. *Npath* is ignored. The auditing system begins writing to the specified "current" file. If the audit system is off, no action is performed; **−1** is returned and **errno** is set to EALREADY.

AUD_SETNEXT The caller issues the **AUD_SETNEXT** command to change only the "next" audit file. If the auditing system is on, the file specified by *npath* is used as the "next" audit file. *Cpath* is ignored. If the "next" audit file specified does not exist, it is created with the specified *mode*. Either an empty string or NULL *npath* can be specified if the caller wants to designate that no "next" file be available to the auditing system. If the auditing system is off, no action is performed; **−1** is returned, and **errno** is set to EALREADY.

AUD_SWITCH The caller issues the **AUD_SWITCH** command to cause auditing system to switch audit files. If the auditing system is on, it uses the "next" file as the new "current" audit file and sets the new "next" audit file to NULL. *Cpath, npath,and mode* are ignored. The auditing system begins writing to the new "current" file. If the auditing system is off, no action is performed; **−1** is returned, and **errno** is set to EALREADY. If the auditing system is on and there is no available "next" file, no action is performed; **−1** is returned, and **errno** is set to ENOENT.

AUD_OFF The caller issues the **AUD_OFF** command to halt the auditing system. If the auditing system is on, it is turned off and the "current" and "next" audit files are closed. *Cpath*, *npath*, and *mode* are ignored. If the audit system is already off, **−1** is returned and **errno** is set to EALREADY.

RETURN VALUE

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

EXAMPLES

In the following example, *audctl* is used to determine whether the auditing system is on, and to retrieve the names of the audit files that are currently in use by the system.

```
char    c_file[PATH_MAX+1], x_file[PATH_MAX+1];
int     mode=0600;

    if (audctl(AUD_GET, c_file, x_file, mode))
        switch ( errno ) {
            case ENOENT:
                strcpy(x_file,"-none-");
                break;
            case EALREADY:
                printf("The auditing system is OFF\n");
                return 0;
            case default:
                fprintf(stderr, "Audctl failed: errno=%d\n", errno);
                return 1;
        }

    printf("The auditing system is ON: c_file=%s x_file=%s\n", c_file, x_file);
    return 0;
```

ERRORS

Audctl fails if one of the following is true:

[EPERM] The caller does not have superuser privilege, or one or both of the given files are not regular files and cannot be used.

[EALREADY] The **AUD_OFF**, **AUD_SET**, **AUD_SETCURR**, **AUD_SETNEXT**, **AUD_SWITCH**, or **AUD_GET** *cmd* specified when the auditing system is off.

[EBUSY] User attempt to start the auditing system failed because auditing is already on.

[EFAULT] Bad pointer. One or more of the required function parameters are not accessible.

[EINVAL] The *cpath* or *npath* is greater than PATH_MAX in length, the *cpath* or *npath* specified is not an absolute path name.

[ENOENT] No available "next" file when *cmd* is **AUD_GETNEXT** or **AUD_SWITCH**.

AUTHOR

Audctl was developed by HP.

SEE ALSO

audit(5), audsys(1M), audomon(1M).

NAME

audswitch – suspend or resume auditing on the current process

SYNOPSIS

#include <sys/audit.h>

int audswitch (aflag)
int aflag;

DESCRIPTION

Audswitch suspends or resumes auditing within the current process. This call is restricted to superusers.

One of the following *aflags* must be used:

AUD_SUSPEND Suspend auditing on the current process.

AUD_RESUME Resume auditing on the current process.

Audswitch can be used in self-auditing privileged processes to temporarily suspend auditing during intervals where auditing is to be handled by the process itself. Auditing is suspended by a call to *audswitch* with the **AUD_SUSPEND** parameter and resumed later by a call to *audswitch* with the **AUD_RESUME** parameter.

An *audswitch* call to resume auditing serves only to reverse the action of a previous *audswitch* call to suspend auditing. A call to *audswitch* to resume auditing when auditing is not suspended has no effect.

Audswitch affects only the current process. For example, *audswitch* cannot suspend auditing for processes *exec'*ed from the current process. (Use *setaudproc*(2) to enable or disable auditing for a process and its children).

RETURN VALUE

Upon successful completion, *audswitch* returns **0**. If an error occurs, **–1** is returned and the global variable **errno** is set to indicate the error.

ERRORS

Audswitch fails if one of the following is true:

[EPERM] The user is not a superuser.

[EINVAL] The input parameter is neither **AUD_RESUME** nor **AUD_SUSPEND**.

AUTHOR

Audswitch was developed by HP.

SEE ALSO

audit(5), setaudproc(2), audusr(1M), audevent(1M).

NAME

audwrite – write an audit record for a self-auditing process

SYNOPSIS

#include <sys/audit.h>

int audwrite(audrec_p)
struct self_audit_rec *audrec_p;

DESCRIPTION

Audwrite is called by trusted self-auditing processes, which are capable of turning off the regular auditing (using *audswitch*(2)) and doing higher-level auditing on their own. *Audwrite* is restricted to superusers.

Audwrite checks to see if the auditing system is on and the calling process and the event specified are being audited. If these conditions are met, *audwrite* writes the audit record pointed to by *audrec_p* into the audit file. The record consists of an audit record body and a header with the following fields:

```
u_long  ah_time;     /* Date/time (tv_sec of timeval) */
u_short ah_pid;      /* Process ID */
u_short ah_error;    /* Success/failure */
u_short ah_event;    /* Event being audited */
u_short ah_len;      /* Length of variant part */
```

The header has the same format as the regular audit record, while the body contains additional information about the high-level audit event. The header fields *ah_error*, *ah_event*, and *ah_len* are specified by the calling process. *Audwrite* fills in *ah_time* and *ah_pid* fields with the correct values. This is done to reduce the risk of forgery. After the header is completed, the record body is attached and the entire record is written into the current audit file.

RETURN VALUE

If the write is successful, a value of **0** is returned. Otherwise, a value of **–1** is returned and **errno** is set to indicate the reason for the failure.

ERRORS

Audwrite fails if one of the following is true:

[EPERM] The caller is not a superuser.

[EINVAL] The event number in the audit record is invalid.

WARNINGS

If *audwrite* causes a file space overflow, the calling process might be suspended until the file space is cleaned up. However a returned call with the return value of 0 indicates that the audit record has been successfully written.

AUTHOR

Audwrite was developed by HP.

SEE ALSO

audswitch(2), audit(4).

NAME

brk, sbrk – change data segment space allocation

SYNOPSIS

int brk (endds)
char *endds;

char *sbrk (incr)
int incr;

DESCRIPTION

Brk and *sbrk* are used to change dynamically the amount of space allocated for the calling process's data segment; see *exec*(2). The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases. The newly allocated space is set to zero.

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

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

ERRORS

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

[ENOMEM] Such a change would result in more space being allocated than is allowed by a system-imposed maximum (see *ulimit*(2)).

[ENOMEM] Such a change would cause a conflict between addresses in the data segment and any attached shared memory segment (see *shmop*(2)).

[ENOMEM] Such a change would be impossible as there is insufficient swap space available.

WARNINGS

The pointer returned by *sbrk* is not necessarily word-aligned. Loading or storing words through this pointer could cause word alignment problems.

Care should be taken when using either *brk*(2) or *sbrk*(2) in conjunction with calls to the *malloc*(3C) or *malloc*(3X) library routines. There is only one program data segment from which all three of these routines allocate and deallocate program data memory. Although it is not recommended practice, it is possible to deallocate program data memory allocated through *malloc*(3C) with a subsequent call to *brk*().

RETURN VALUE

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

AUTHOR

Brk and *sbrk* were developed by AT&T and HP.

SEE ALSO

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

STANDARDS CONFORMANCE

brk: XPG2

sbrk: XPG2

NAME

killpg, getpgrp, setpgrp, sigvec, signal – 4.2 BSD-compatible process control facilities

SYNOPSIS

```
int killpg(pgrp, sig)
int pgrp, sig;

int getpgrp(pid)
int pid;

int setpgrp(pid, pgrp)
int pid, pgrp;

#include <signal.h>

int sigvec(sig, vec, ovec)
int sig;
struct sigvec *vec, *ovec;

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

DESCRIPTION

These calls simulate (and are provided for backward compatibility with) functions of the same name in the 4.2 Berkeley Software Distribution.

This version of *setpgrp* is equivalent to the system call *setpgid(pid, pgrp)* (see *setpgid*(2)).

This version of *getpgrp* is equivalent to the system call *getpgrp2(pid)* (see *getpid*(2)).

Killpg is equivalent to the system call *kill(−pgrp, sig)* (see *kill*(2)).

Sigvec is equivalent to the system call *sigvector(sig, vec, ovec)* (see *sigvector*(2)), except for the following:

> When SIGCHLD or SIGCLD is used and *vec* specifies a catching function, the routine acts as if the SV_BSDSIG flag were included in the *sv_flags* field of *vec*.
>
> The name *sv_onstack* can be used as a synonym for the name of the *sv_flags* field of *vec* and *ovec*.
>
> If *vec* is not a null pointer and the value of (*vec−>sv_flags* & 1) is "true", the routine acts as if the SV_ONSTACK flag were set.
>
> If *ovec* is not a null pointer, the flag word returned in *ovec−>sv_flags* (and therefore the value of *ovec−>sv_onstack*) will be equal to 1 if the system was reserving space for processing of that signal because of a call to *sigspace*(2), and 0 if not. The SV_BSDSIG bit in the value placed in *ovec−>sv_flags* is always clear.
>
> If the reception of a caught signal occurs during certain system calls, the call will always be restarted, regardless of the return value from a catching function installed with *sigvec*(). The affected calls are *wait*(2), *semop*(2), *msgsnd*(2), *msgrcv*(2), and *read*(2) or *write*(2) on a slow device (such as a terminal or pipe, but not a file). Other interrupted system calls are not restarted.

This version of *signal* has the same effect as *sigvec(sig, vec, ovec)*, where *vec−>sv_handler* is equal to *func*, *vec−>sv_mask* is equal to 0, and *vec−>sv_flags* is equal to 0. *Signal* returns the value that would be stored in *ovec−>sv_handler* if the equivalent *sigvec* call would have succeeded. Otherwise, *signal* returns −1 and **errno** is set to indicate the reason as it would have been set by the equivalent call to *sigvec*.

These functions can be linked into a program by giving the **-lBSD** option to *ld*(1).

WARNINGS

While the 4.3 BSD release defined extensions to some of the interfaces described here, only the 4.2 BSD interfaces are emulated by this package.

Bsdproc should not be used in conjunction with the facilities described under *sigset*(2V).

AUTHOR

Bsdproc was developed by HP and the University of California, Berkeley.

SEE ALSO

ld(1), kill(2), getpid(2), msgsnd(2), msgrcv(2), read(2), semop(2), setpgid(2), setsid(2), sigvector(2), wait(2), write(2), sigset(2V), sigstack(2), signal(5).

NAME

chdir – change working directory

SYNOPSIS

int chdir (path)
char *path;

DESCRIPTION

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

ERRORS

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

[ENOTDIR] A component of the path name is not a directory.

[ENOENT] The named directory does not exist.

[EACCES] Search permission is denied for any component of the path name.

[EFAULT] *Path* points outside the allocated address space of the process. The reliable detection of this error will be implementation dependent.

[ENOENT] *Path* is null.

[ENAMETOOLONG]
The length of the specified path name exceeds PATH_MAX bytes, or the length of a component of the path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.

[ELOOP] Too many symbolic links were encountered in translating the path name.

RETURN VALUE

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

AUTHOR

Chdir was developed by AT&T Bell Laboratories and the Hewlett-Packard Company.

SEE ALSO

cd(1), chroot(2).

STANDARDS CONFORMANCE

chdir: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

chmod, fchmod – change access mode of file

SYNOPSIS

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

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

int fchmod (fildes, mode)
int fildes;
mode_t mode;
```

DESCRIPTION

The *path* argument points to a path name naming a file. The *fildes* argument is a file descriptor. *Chmod* and *fchmod* set the access permission portion of the file's mode according to the bit pattern contained in *mode*.

The following symbolic constants representing the access permission bits are defined with the indicated values in **<sys/stat.h>** and are used to construct the argument *mode*. The value of the argument *mode* is the bitwise inclusive OR of the values for the desired permissions.

S_ISUID	04000	Set user ID on execution.
S_ISGID	02000	Set group ID on execution.
S_ENFMT	02000	Record locking enforced.
S_ISVTX	01000	Save text image after execution.
S_IRUSR	00400	Read by owner.
S_IWUSR	00200	Write by owner.
S_IXUSR	00100	Execute (search) by owner.
S_IRGRP	00040	Read by group.
S_IWGRP	00020	Write by group.
S_IXGRP	00010	Execute (search) by group.
S_IROTH	00004	Read by others (that is, anybody else).
S_IWOTH	00002	Write by others.
S_IXOTH	00001	Execute (search) by others.

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

If the effective-user-ID of the process is not that of the superuser, **S_ISVTX** (mode bit 01000, save text image on execution) is cleared.

If the effective-user-ID of the process is not that of the superuser, and the effective-group-ID of the process does not match the group ID of the file and none of the group IDs in the supplementary groups list match the group ID of the file, **S_ISGID, S_ENFMT** (mode bit 02000, set group ID on execution and enforced file locking mode) is cleared.

The set-group-ID on execution bit is also used to enforce file-locking mode (see *lockf*(2) and *fcntl*(2)) on files that are not group executable. This might affect future calls to *open*(2), *creat*(2), *read*(2), and *write*(2) on such files.

If an executable file is prepared for sharing, **S_ISVTX** (mode bit 01000) prevents the system from abandoning the swap-space image of the program-text portion of the file when its last user terminates. Then, when the next user of the file executes it, the text need not be read from the file system but can simply be swapped in, thus saving time.

Access Control Lists

All optional entries in a file's access control list are deleted when *chmod* is executed. (This behavior conforms to the IEEE Standard POSIX 1003.1-1988.) To preserve optional entries in a file's access control list, it is necessary to save and restore them using *getacl*(2) and *setacl*(2).

To set the permission bits of access control list entries, use *setacl*(2) instead of *chmod*.

For more information on access control list entries, see *acl*(5).

RETURN VALUE

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

ERRORS

Chmod and *fchmod* fail and the file mode is unchanged if one or more of the following is true:

[EACCES] Search permission is denied on a component of the path prefix.

[EFAULT] *Path* points outside the allocated address space of the process. The reliable detection of this error is implementation dependent.

[ELOOP] Too many symbolic links are encountered in translating *path*.

[ENAMETOOLONG] A component of *path* exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect or *path* exceeds PATH_MAX bytes.

[ENOENT] A component of *path* does not exist.

[ENOENT] The file named by *path* does not exist.

[ENOTDIR] A component of the path prefix is not a directory.

[EPERM] The effective-user-ID does not match that of the owner of the file and the effective-user-ID is not that of the super-user.

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

DEPENDENCIES

HP Clustered Environment:

If the file is a directory, the access permission bit S_CDF (04000) indicates a hidden directory (see *cdf*(4)).

RFA and NFS

Fchmod is not supported on remote files.

AUTHOR

Chmod was developed by AT&T, the University of California, Berkeley, and HP.

Fchmod was developed by the University of California, Berkeley.

SEE ALSO

chmod(1), chown(2), creat(2), fcntl(2), getacl(2), lockf(2), mknod(2), open(2), read(2), setacl(2), write(2), cdf(4), acl(5).

STANDARDS CONFORMANCE

chmod: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

chown, fchown – change owner and group of a file

SYNOPSIS

#include <sys/types.h>

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

int fchown (fildes, owner, group)
int fildes;
uid_t owner;
gid_t group;

DESCRIPTION

Chown changes the user and group ownership of a file. The *path* argument points to a path name naming a file. The *fildes* argument is a file descriptor. The *chown* and *fchown* functions set the owner ID and group ID of the file to the numeric values contained in *owner* and *group* respectively. A value of **UID_NO_CHANGE** or **GID_NO_CHANGE** can be specified in *owner* or *group* to leave unchanged the file's owner ID or group ID respectively. Note that *owner* and *group* should be less than or equal to UID_MAX (see *limits*(5)).

Only processes with effective user ID equal to the file owner or superuser can change the ownership of a file. If privilege groups are supported, the owner of a file can change the ownership only if he is a member of a privilege group allowing CHOWN, as set up by *setprivgrp*(1M). All users get CHOWN privileges by default.

The group ownership of a file can be changed to any group in the current process's access list or to the real or effective group ID of the current process. If privilege groups are supported and the user is permitted the CHOWN privilege, the file can be given to any group.

If *chown* is invoked on a regular file by other than the superuser, the set-user-ID and set-group-ID bits of the file mode are cleared. Whether *chown* preserves or clears these bits on files of other types is implementation dependent.

Access Control Lists (ACLs)

A user can allow or deny specific individuals and groups access to a file by using the file's access control list (see *acl*(5)). When using *chown*(2) in conjunction with ACLs, if the new owner and/or group does not have an optional ACL entry corresponding to *u*.% and/or %.*g* in the file's access control list, the file's access permission bits remain unchanged. However, if the new owner and/or group is already designated by an optional ACL entry of *u*.% and/or %.*g*, *chown* sets the file's permission bits (and the three basic ACL entries) to the permissions contained in that entry.

ERRORS

Chown fails and the owner and group of the file remain unchanged if one or more of the following is true:

[EBADF]	*Fildes* is not a valid file descriptor.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The file named by *path* does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[EPERM]	The effective user ID is not superuser and one or more of the following conditions exist:

The effective user ID does not match the owner of the file.

When changing the owner of the file, if the owner of the file is not a member of a privilege group allowing the CHOWN privilege.

When changing the group of the file, if the owner of the file is not a member of a privilege group allowing the CHOWN privilege and the group number is not in the current process's access list.

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

[EFAULT] *Path* points outside the allocated address space of the process. The reliable detection of this error will be implementation dependent.

[ENAMETOOLONG]
A component of *path* exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect, or *path* exceeds PATH_MAX bytes.

[ELOOP] Too many symbolic links were encountered in translating *path*.

DEPENDENCIES

HP Clustered Environment:

Chown does not clear the set-user-ID bit of a directory, because that bit indicates that the directory is hidden (see *cdf*(4)).

When *chown* is called from a diskless node, the privilege groups checked are the ones set up on the cluster server.

RETURN VALUE

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

AUTHOR

Fchown was developed by the University of California, Berkeley.

SEE ALSO

chown(1), setprivgrp(1M), chmod(2), setacl(2), acl(5), limits(5).

STANDARDS CONFORMANCE

chown: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

chroot – change root directory

SYNOPSIS

int chroot (path)
char *path;

DESCRIPTION

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

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

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

RETURN VALUE

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

ERRORS

Chroot will fail and the root directory will remain unchanged if one or more of the following are true:

[ENOTDIR] Any component of the path name is not a directory.

[ENOENT] The named directory does not exist or a component of the *path* does not exist.

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

[EFAULT] *Path* points outside the allocated address space of the process. The reliable detection of this error will be implementation dependent.

[ENAMETOOLONG]
The length of the specified path name exceeds PATH_MAX bytes, or the length of a component of the path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.

[ELOOP] Too many symbolic links were encountered in translating the path name.

SEE ALSO

chroot(1M), chdir(2).

STANDARDS CONFORMANCE

chroot: SVID2, XPG2, XPG3

NAME

close – close a file descriptor

SYNOPSIS

int close (fildes)
int fildes;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat, open, dup, fcntl,* or *pipe* system call. *Close* closes the file descriptor indicated by *fildes*. All associated file segments which have been locked by this process with the *lockf* function are released (i.e., unlocked).

ERRORS

[EBADF] *Close* will fail if *fildes* is not a valid open file descriptor.

RETURN VALUE

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

SEE ALSO

creat(2), dup(2), exec(2), fcntl(2), lockf(2), open(2), pipe(2).

STANDARDS CONFORMANCE

close: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

cnodeid – get the cnode ID of the local machine

SYNOPSIS

#include <sys/types.h>

cnode_t cnodeid ()

DESCRIPTION

Cnodeid returns the cnode ID of the local machine.

SEE ALSO

cnodes(1), cnodes(2), getccent(3C).

AUTHOR

Cnodeid was developed by HP.

NAME

cnodes – get a list of active nodes in cluster

SYNOPSIS

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

int cnodes (buf)
cnode_t *buf;

DESCRIPTION

Cnodes returns in *buf* the current number of active cnodes in the cluster. If *buf* is not a null pointer, it should be a pointer to an array of at least **MAX_CNODE** cnode IDs. This array will be filled with the cnode IDs of nodes currently in the cluster; the list of cnode IDs is terminated by the cnode ID **0**.

RETURN VALUE

Upon successful completion, *cnodes* returns the current number of active cnodes. If the value **0** is returned, the machine is not a member of a cluster. In case of an error, a value of **–1** is returned and **errno** is set to indicate the error.

ERRORS

Cnodes may fail if:

[EFAULT] *Buf* is not a null pointer and points to an illegal address. Reliable detection of this error is not guaranteed.

SEE ALSO

cnodeid(2), cnodes(1), getccent(3C).

AUTHOR

Cnodes was developed by HP.

NAME

creat – create a new file or rewrite an existing one

SYNOPSIS

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

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

DESCRIPTION

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

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

> All bits set in the process's file mode creation mask are cleared. See *umask*(2).
> The "save text image after execution" bit of the mode are cleared. See *chmod*(2).

Upon successful completion, the file descriptor is returned and the file is open for writing (only), even if the *mode* does not permit writing. The file offset is set to the beginning of the file. The file descriptor is set to remain open across *exec* system calls (see *fcntl*(2)). No process can have more than OPEN_MAX files open simultaneously. This is discussed in *open*(2). A new file can be created with a mode that forbids writing.

Access Control Lists (ACLs)

On systems that support access control lists, three base ACL entries are created corresponding to the file access permission bits. An existing file's access control list is unchanged by *creat* (see *setacl*(2), *chmod*(2), and *acl*(5)).

ERRORS

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

[ENOSPC]	Not enough space on the file system.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist (for example, *path* is null, or a component of *path* does not exist).
[EACCES]	Search permission is denied on a component of the path prefix.
[EACCES]	The file does not exist and the directory in which the file is to be created does not permit writing.
[EROFS]	The named file resides or would reside on a read-only file system.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed.
[EACCES]	The file exists and write permission is denied.
[EISDIR]	The named file is an existing directory.
[EMFILE]	More than the maximum number of file descriptors are currently open.
[EFAULT]	*Path* points outside the allocated address space of the process. The reliable detection of this error is implementation dependent.

[ENFILE] The system file table is full.

[ENXIO] The named file is a character special or block special file, and the device associated with this special file does not exist.

[ENAMETOOLONG]
The length of the specified path name exceeds PATH_MAX bytes, or the length of a component of the path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.

[EAGAIN] The file exists, enforcement mode file and record locking is set and there are outstanding record locks on the file.

[ELOOP] Too many symbolic links were encountered in translating the path name.

RETURN VALUE

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

SEE ALSO

chmod(2), close(2), dup(2), fcntl(2), lockf(2), lseek(2), open(2), read(2), setacl(2), truncate(2), umask(2), write(2), acl(5).

STANDARDS CONFORMANCE

creat: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

dup – duplicate an open file descriptor

SYNOPSIS

int dup (fildes)
int fildes;

DESCRIPTION

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

Same open file (or pipe).

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

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

Same file status flags (see *fcntl*(2), F_DUPFD).

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

The file descriptor returned is the lowest one available.

ERRORS

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

[EBADF] *Fildes* is not a valid open file descriptor.

[EMFILE] The maximum number of file descriptors are currently open.

RETURN VALUE

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

AUTHOR

Dup was developed by AT&T Bell Laboratories and the Hewlett-Packard Company.

SEE ALSO

close(2), creat(2), dup2(2), exec(2), fcntl(2), open(2), pipe(2).

STANDARDS CONFORMANCE

dup: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

dup2 – duplicate an open file descriptor to a specific slot

SYNOPSIS

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

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

Fildes2 is a non-negative integer less than the maximum value allowed for file descriptors.

Dup2 causes *fildes2* to refer to the same file as *fildes*. If *fildes2* already referred to an open file, it is closed first.

The file descriptor returned by *dup2* has the following in common with *fildes*:

Same open file (or pipe).

Same file pointer (that is, both file descriptors share one file pointer.)

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

Same file status flags (see *fcntl*(2), F_DUPFD).

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

This routine is found in the C library. Programs using *dup2* but not using other routines from the Berkeley importability library (such as the routines described in *bsdproc*(2)) should not give the **–lBSD** option to *ld*(1).

ERRORS

Dup2 will fail if the following is true:

[EBADF] *Fildes* is not a valid open file descriptor or *fildes2* is not in the range of legal file descriptors.

RETURN VALUE

Upon successful completion, *dup2* returns a non-negative integer, namely the new file descriptor *fildes2*. Otherwise, it returns **–1** and sets **errno** to indicate the error.

SEE ALSO

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

STANDARDS CONFORMANCE

dup2: SVID2, XPG3, POSIX.1, FIPS 151-1

NAME

errno – error indicator for system calls

SYNOPSIS

#include <errno.h>
extern int errno;

DESCRIPTION

The value of the external variable **errno** is set whenever an error occurs in a system call. This value can be used to obtain a more detailed description of the error. An error condition is indicated by an otherwise impossible returned value. This is almost always −1; the individual descriptions specify the details. Because **errno** is not cleared on successful system calls, its value should be checked only when an error has been indicated.

Each system call description attempts to list all possible error numbers. The following is a complete list of the error names. The numeric values can be found in **<errno.h>** but should not normally be used.

E2BIG — Arg list too long. An argument and or environment list longer than maximum supported size is presented to a member of the *exec* family. Other possibilities include: message size or number of semaphores exceeds system limit *(msgop, semop),* or too many privileged groups have been set up *(setprivgrp).*

EACCES — Permission denied. An attempt was made to access a file or IPC object in a way forbidden by the protection system.

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

EADDRNOTAVAIL — Cannot assign requested address. Normally results from an attempt to create a socket with an address not on this machine.

EAFNOSUPPORT — Address family not supported by protocol family. An address incompatible with the requested protocol was used. For example, you should not necessarily expect to be able to use PUP Internet addresses with ARPA Internet protocols.

EAGAIN — No more processes. A *fork* failed because the system's process table is full or the user is not allowed to create any more processes, or a *semop* or *msgop* call would have to block.

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

EBADF — Bad file number. Either a file descriptor refers to no open file, a read (respectively write) request is made to a file which is open only for writing (respectively reading), or the file descriptor is not in the legal range of file descriptors.

EBUSY — Device or resource busy. An attempt to mount a device that was already mounted or an attempt was made to dismount a device on which there is an active file (open file, current directory, mounted-on file, active text segment). It will also occur if an attempt is made to enable accounting when it is already enabled. The device or resource is currently unavailable, such as when a non-shareable device file is in use.

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

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

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

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

EDEADLK Resource deadlock would occur. A process which has locked a system resource would have been put to sleep while attempting to access another process' locked resource.

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

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

EEXIST File exists. An existing file was mentioned in an inappropriate context, e.g., *link*.

EFAULT Bad address. The system encountered a hardware fault in attempting to use an argument of a system call; can also result from passing the wrong number of parameters to a system call. The reliable detection of this error will be implementation dependent.

EFBIG File too large. The size of a file exceeded the maximum file size (for the file system) or ULIMIT was exceeded. (see *ulimit*(2)), or a bad semaphore number in a *semop*(2) call.

EHOSTDOWN Host is down. A socket operation encountered a dead host. Networking activity on the local host has not been intiated.

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

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

EINPROGRESS Operation now in progress. An operation which takes a long time to complete was attempted on a non-blocking object (see *ioctl*(2) and *fcntl*(2)).

EINTR Interrupted system call. An asynchronous signal (such as interrupt or quit), which the user has elected to catch, occurred during a system call. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned this error condition unless the system call is restarted (see *sigvector*(2)).

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

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

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

EISDIR	Is a directory. An attempt to open a directory for writing.
ELOOP	Too many levels of symbolic links. A path name search involved more than MAXSYMLINKS symbolic links. MAXSYMLINKS is defined in **<sys/param.h>**.
EMFILE	Too many open files. No process may have more than a system defined number of file descriptors open at a time.
EMLINK	Too many links. An attempt to make more than the maximum number of links to a file.
EMSGSIZE	Message too long. The socket requires that the message be sent atomically, and the size of the message to be sent made this impossible.
ENAMETOOLONG	File name too long. A path specified exceeds the maximum path length for the system. The maximum path length is specified by PATH_MAX and is defined in **<limits.h>**. PATH_MAX is guaranteed to be at least 1023 bytes. This error is also generated if the length of a path name component exceeds NAME_MAX and the _POSIX_NO_TRUNC option is in effect for the specified path. Currently, _POSIX_NO_TRUNC is in effect only for HFS file systems configured to allow path name components up to 255 bytes long (see *convertfs*(1M)) and therefore only path names referring to such file systems will generate the error for this case. The values of NAME_MAX, PATH_MAX, and _POSIX_NO_TRUNC for a particular path name can be queried by using the *pathconf*(2) system call.
ENET	Local area network error. An error occurred in the software or hardware associated with your local area network.
ENETDOWN	Network is down. A socket operation encountered a dead network.
ENETRESET	Network dropped connection on reset. The host you were connected to crashed and rebooted.
ENETUNREACH	Network is unreachable. A socket operation was attempted to an unreachable network.
ENFILE	File table overflow. The system's table of open files is full, and temporarily no more *opens* can be accepted.
ENOBUFS	No buffer space available. An operation on a socket was not performed because the system lacked sufficient buffer space.
ENODEV	No such device. An attempt was made to apply an inappropriate system call to a device; e.g., read a write-only device.
ENOENT	No such file or directory. This error occurs when a file name is specified and the file should exist but doesn't, or when one of the directories in a path name does not exist. It also occurs with *msgget, semget, shmget* when *key* does not refer to any object and the **IPC_CREAT** flag is not set.
ENOEXEC	Exec format error. A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid magic number (see *a.out*(4)), or the file is too small to have a valid executable file header.
ENOMEM	Not enough space. During a system call such as *exec*, *brk*, *fork*, or *sbrk*, a program asks for more space than the system is able to supply. This may not be a temporary condition; the maximum space size is a system parameter. The error may also occur if the arrangement of text, data, and stack segments requires too many segmentation registers, or if there is not enough swap space during a *fork*.

ENOMSG No message of desired type. An attempt was made to receive a message of a type that does not exist on the specified message queue; see *msgop*(2).

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

ENOSPC No space left on device. During a *write* to an ordinary file, there is no free space left on the device; or, no space in system table during *msgget*(2), *semget*(2), or *semop*(2) while **SEM_UNDO** flag is set.

ENOSYS Function is not available. The requested function or operation is not implemented or not configured in the system.

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

ENOTCONN Socket is not connected. A request to send or receive data was disallowed because the socket was not connected.

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

ENOTEMPTY Directory not empty. An attempt was made to remove a non-empty directory.

ENOTSOCK Socket operation on non-socket. An operation was attempted on something that is not a socket.

ENOTTY Not a typewriter. The (*ioctl*(2)) command is inappropriate to the selected device type.

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

EOPNOTSUPP Operation not supported. The requested operation on a socket, RFA file, or NFS file is either invalid or unsupported. For example, this might occur when an attempt to *accept* a connection on a datagram socket fails.

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

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

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

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

ERANGE Result too large. The value of a function in the math package (3M) is not representable within machine precision, or a *semop*(2) call would cause either a semaphore value or a semaphore adjust value to exceed it system-imposed maximum.

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

ESHUTDOWN Cannot send after socket shutdown. A request to send data was disallowed because the socket had already been shut down with a previous *shutdown*(2) call.

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

ESPIPE Illegal seek. An *lseek* was issued to a pipe.

ESRCH No such process. No process can be found corresponding to that specified by *pid* in *kill, rtprio* or *ptrace*, or the process is not accessible.

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

ETXTBSY Text file busy. An attempt to execute an executable file which is currently open for writing (or reading). Also, an attempt to open for writing an otherwise writable file which is currently open for execution.

EWOULDBLOCK Operation would block. An operation which would cause a process to block was attempted on a object in non-blocking mode (see *ioctl*(2) and *fcntl*(2)).

EXDEV Cross-device link. A link to a file on another device was attempted.

DEPENDENCIES

The following NFS errors are also defined:

EREFUSED The same error as. ECONNREFUSED. The external variable **errno** is defined as ECONNREFUSED for NFS compatibility.

EREMOTE Too many levels of remote in path. An attempt was made to remotely mount an NFS file system into a path which already has a remotely mounted NFS file system component.

ESTALE Stale NFS file handle. A client referenced an open file, but the file had previously been deleted.

Series 800:

In the definition of error ENOMEM, the term "segmentation registers" is invalid.

STANDARDS CONFORMANCE

errno: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

execl, execv, execle, execve, execlp, execvp – execute a file

SYNOPSIS

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

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

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

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

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

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

DESCRIPTION

Exec, in all its forms, loads a program from an ordinary, executable file onto the current process, replacing the current program. The *path* or *file* argument refers to either an executable object file or a file of data for an interpreter. In this case, the file of data is also called a script file.

An executable object file consists of a header (see *a.out*(4)), text segment, and data segment. The data segment contains an initialized portion and an uninitialized portion (bss). For *execlp* and *execvp* the shell (**/bin/sh**) can be loaded to interpret a script instead. A successful call to *exec* does not return because the new program overwrites the calling program.

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

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

where *argc* is the argument count and *argv* is the address of an array of character pointers to the arguments themselves. As indicated, *argc* usually has a value of at least one, and the first member of the array points to a string containing the name of the file. (The exit conditions from *main* are discussed in *exit*(2).)

Path points to a path name that identifies the executable file containing the new program.

File (in *execlp* or *execvp*) points to a file name identifying the executable file containing the new program. The path prefix for this file is obtained by searching the directories passed as the environment line "PATH =" (see *environ*(5)). The environment is supplied by the shell (see *sh*(1)). If *file* does not have an executable magic number (*magic*(4)), it is passed to **/bin/sh** as a shell script.

Arg0, arg1, ..., argn are pointers to null-terminated character strings. These strings constitute the argument list available to the new program. By convention, at least *arg0* must be present and point to a string identical to *path* or *path*'s last component.

Argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new program. By convention, *argv* must have at least one member, and must point to a string that is identical to *path* or *path*'s last component. *Argv* is terminated by a null pointer.

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

extern char **environ;

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

Open file descriptors remain open, except for those whose close-on-exec flag is set; see *fcntl*(2). The file offset, access mode, and status flags of open file descriptors are unchanged.

Note that normal executable files are open only briefly, when they start execution. Other executable file types can be kept open for a long time, or even indefinitely under some circumstances.

The processing of signals by the process is unchanged by *exec*, except that signals caught by the process are set to their default value; see *signal*(2).

If the set-user-ID mode bit of the executable file pointed to by *path* or *file* is set (see *chmod*(2)), *exec* sets the effective-user-ID of the new process to the user ID of the executable file. Similarly, if the set-group-ID mode bit of the executable file is set, the effective-group-ID of the process is set to the group ID of the executable file. The real-user-ID and real-group-ID of the process are unchanged. Note that the set-user(group)-ID function does not apply to scripts; thus, if *execlp* or *execvp* executes a script, the set-user(group)-ID bits are ignored, even if they are set.

The saved-user-ID and saved-group-ID of the process are always set to the effective-user-ID and effective-group-ID, respectively, of the process at the end of the *exec*, whether or not set-user(group)-ID is in effect.

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

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

The process also retains the following attributes:

current working directory
file creation mode mask (see *umask*(2))
file locks (see *fcntl*(2)), except for files closed-on-exec
file size limit (see *ulimit*(2))
interval timers (see *getitimer*(2))
nice value (see *nice*(2))
parent process ID
pending signals
process ID
process group ID
real user ID
real group ID
real-time priority (see *rtprio*(2))
root directory (see *chroot*(2))
semadj values (see *semop*(2))
session membership
signal mask (see *sigvector*(2))
supplementary group IDs
time left until an alarm clock signal (see *alarm*(2))
trace flag (see *ptrace*(2) PT_SETTRC request)
tms_utime, *tms_stime*, *tms_cutime*, and *tms_cstime* (see *times*(2))

The initial line of a script file must begin with **#!** as the first two bytes, followed by 0 or more spaces, followed by *interpreter* or *interpreter argument*. One or more space or tab must separate *interpreter* and *argument*. The first line should end with either a new line or null character.

> **#!** *interpreter*
> **#!** *interpreter argument*

When the script file is executed, the system executes the specified *interpreter* as an executable object file. Even in the case of *execlp* or *execvp*, no path searching is done of the interpreter name.

The *argument* is anything that follows the *interpreter* and tabs or spaces. If an *argument* is given, it is passed to the *interpreter* as *argv*[1] and the name of the script file is passed as *argv*[2]. Otherwise, the name of the script file is passed as *argv*[1]. The *argv*[0] is passed as specified in the *exec* call, unless either *argv* or *argv*[0] is null as specified, in which case a pointer to a null string is passed as *argv*[0]. All other arguments specified in the *exec* call are passed following the name of the script file (that is, beginning at *argv*[3] if there is an argument; otherwise at *argv*[2]).

If the initial line of the script file exceeds a system-defined maximum number of characters, *exec* fails. The minimum value for this limit is 32.

Set-user-ID and set-group-ID bits are honored for the script and not for the interpreter.

RETURN VALUE

If *exec* returns to the calling program, an error has occurred; the return value is **−1** and **errno** is set to indicate the error.

ERRORS

Exec fails and returns to the calling program if one or more of the following is true:

[E2BIG] The number of bytes in the new program's argument list is greater than the system-imposed limit. This limit is at least 5120 bytes on HP-UX systems.

[EACCES] Read permission is denied for the executable file or interpreter, and trace flag (see *ptrace*(2) request PT_SETTRC) of the process is set.

[EACCES] Search permission is denied for a directory listed in the executable file's or the interpreter's path prefix.

[EACCES] The executable file or the interpreter is not an ordinary file.

[EACCES] The file described by *path* or *file* is not executable. The superuser cannot execute a file unless at least one access permission bit or entry in its access control list has an execute bit set.

[EFAULT] *Path*, *argv*, or *envp* point to an illegal address. The reliable detection of this error is implementation dependent.

[EFAULT] The executable file is shorter than indicated by the size values in its header, or is otherwise inconsistent. The reliable detection of this error is implementation dependent.

[EINVAL] The executable file is incompatible with the architecture on which the *exec* has been performed, and is presumed to be for a different architecture. It is not guaranteed that every architecture's executable files wiill be recognized.

[ELOOP] Too many symbolic links are encountered in translating the path name.

[ENAMETOOLONG]
The executable file's path name or the interpreter's path name exceeds PATH_MAX bytes, or the length of a component of the path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.

[ENOENT] *Path* is null.

[ENOENT] One or more components of the executable file's path name or the interpreter's path name does not exist.

[ENOEXEC] The *exec* is not an *execlp* or *execvp*, and the executable file has the appropriate access permission, but there is neither a valid magic number nor the characters **#!** as the first two bytes of its initial line.

[ENOEXEC] The number of bytes in the initial line of a script file exceeds the system's maximum.

[ENOMEM] The new process requires more memory than is available or allowed by the system-imposed maximum.

[ENOTDIR] A component of the executable file's path prefix or the interpreter's path prefix is not a directory.

[ETXTBSY] The executable file is currently open for writing.

DEPENDENCIES

Series 800

Unsharable executable files (EXEC_MAGIC magic number produced via the **−N** option of *ld*(1)) are not supported.

SEE ALSO

sh(1), alarm(2), exit(2), fork(2), nice(2), ptrace(2), semop(2), signal(2), times(2), ulimit(2), umask(2), a.out(4), acl(5), environ(5), signal(5).

STANDARDS CONFORMANCE

environ: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

execl: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

execle: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

execlp: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

execv: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

execve: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

execvp: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

exit, _exit – terminate process

SYNOPSIS

#include <stdlib.h>

void exit (status)
int status;

void _exit (status)
int status;

DESCRIPTION

Exit terminates the calling process and passes *status* to the system for inspection, see *wait*(2). Returning from *main* in a C program has the same effect as *exit;* the *status* value is the function value returned by *main*. (This value will be undefined if *main* does not take care to return a value or to call *exit* explicitly.)

The *exit* function cannot return to its caller. The result of an *exit* call during exit processing is undefined.

The functions *exit* and *_exit,* are equivalent except that *exit* calls functions registered by *atexit* and flushes stdio buffers, while *_exit* does not. Both *exit* and *_exit* terminate the calling process with the following consequences:

Functions registered by *atexit*(2) are called in reverse order of registration.

All file descriptors open in the calling process are closed.

All files created by *tmpfile*(3C) are removed.

If the parent process of the calling process is executing a *wait*, *wait3*, or *waitpid*, it is notified of the calling process's termination and the low order eight bits, i.e., bits 0377, of *status* are made available to it, see *wait*(2).

If the parent process of the calling process is not executing a *wait*, *wait3*, or *waitpid*, and does not have SIGCLD set to SIG_IGN, the calling process is transformed into a *zombie* process. A *zombie* process is a process that only occupies a slot in the process table. It has no other space allocated either in user or kernel space. Time accounting information is recorded for use by *times*(2).

The parent process ID of all of the calling process's existing child processes and zombie processes is set to 1. This means the initialization process (proc1) inherits each of these processes.

Each attached shared memory segment is detached and the value of **shm_nattach** in the data structure associated with its shared memory identifier is decremented by 1, see *shmop*(2).

For each semaphore for which the calling process has set a semadj value, see *semop*(2), that semadj value is added to the semval of the specified semaphore.

If the process has a process, text, or data lock, an *unlock* is performed, see *plock*(2).

An accounting record is written on the accounting file if the system's accounting routine is enabled, see *acct*(2).

A SIGCHLD signal is sent to the parent process.

If the calling process is a controlling process, the SIGHUP signal is sent to each process in the foreground process group of the controlling terminal belonging to the calling process. The controlling terminal associated with the session is disassociated from the session, allowing it to be acquired by a new controlling process.

If the exit of the calling process causes a process group to become orphaned, and if any member of the newly-orphaned process group is stopped, all processes in the newly-orphaned process group are sent SIGHUP and SIGCONT signals.

If the current process has any child processes that are being traced, they will be sent a SIGKILL signal.

AUTHOR

Exit was developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

Exit conditions (**$?**) in sh(1), acct(2), plock(2), semop(2), shmop(2), times(2), vfork(2), wait(2), signal(5).

STANDARDS CONFORMANCE

exit: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

_exit: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

fcntl – file control

SYNOPSIS

#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

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

union {
  int val;
  struct flock *lockdes;
} arg;

DESCRIPTION

Fcntl provides for control over open files. *Fildes* is an open file descriptor.

The following are possible values of the *cmd* argument:

F_DUPFD Return a new file descriptor having the following characteristics:

Lowest numbered available file descriptor greater than or equal to *arg.***val**.

Same open file (or pipe) as the original file.

Same file pointer as the original file (that is, both file descriptors share one file pointer).

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

Same file status flags (that is, both file descriptors share the same file status flags).

The close-on-exec flag associated with the new file descriptor is set to remain open across *exec*(2) system calls.

F_GETFD Get the close-on-exec flag associated with the file descriptor *fildes*. If the low-order bit is **0** the file will remain open across *exec*(2), otherwise the file will be closed upon execution of *exec*(2).

F_SETFD Set the close-on-exec flag associated with *fildes* to the low-order bit of *arg.***val** (see F_GETFD).

F_GETFL Get file status flags and access modes; see *fcntl*(5).

F_SETFL Set file status flags to *arg.***val**. Only certain flags can be set; see *fcntl*(5). It is not possible to set both O_NDELAY and O_NONBLOCK .

F_GETLK Get the first lock that blocks the lock described by the variable of type **struct flock** pointed to by *arg*. The information retrieved overwrites the information passed to *fcntl* in the **flock** structure. If no lock is found that would prevent this lock from being created, the structure is passed back unchanged, except that the lock type is set to F_UNLCK.

F_SETLK Set or clear a file segment lock according to the variable of type **struct flock** pointed to by *arg.***lockdes** (see *fcntl*(5)). The *cmd* F_SETLK is used to establish read (F_RDLCK) and write (F_WRLCK) locks, as well as to remove either type of lock (F_UNLCK). If a read or write lock cannot be set, *fcntl* returns immediately with an error value of **–1**.

F_SETLKW This *cmd* is the same as F_SETLK except that if a read or write lock is blocked by other locks, the process will sleep until the segment is free to be locked.

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

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

The structure **flock** describes the type (**l_type**), starting offset (**l_whence**), relative offset (**l_start**), size (**l_len**), and process ID (**l_pid**) of the segment of the file to be affected. The process ID field is only used with the F_GETLK *cmd* to return the value of a block in lock. Locks can start and extend beyond the current end of a file, but cannot be negative relative to the beginning of the file. A lock can be set to always extend to the end of file by setting **l_len** to zero (0). If such a lock also has **l_start** set to zero (0), the whole file will be locked. Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments for either end. Locking a segment already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process in a *fork*(2) system call.

When enforcement-mode file and record locking is activated on a file (see *chmod*(2)), future *read*(2) and *write*(2) system calls on the file are affected by the record locks in effect.

NETWORKING FEATURES

NFS

The advisory record-locking capabilities of *fcntl*(2) are implemented throughout the network by the "network lock daemon"; see *lockd*(1M). If the file server crashes and is rebooted, the lock daemon attempts to recover all locks associated with the crashed server. If a lock cannot be reclaimed, the process that held the lock is issued a SIGLOST signal.

Record locking, as implemented for NFS files, is only advisory.

ERRORS

Under the following conditions, the function *fcntl* fails and sets the external variable **errno** accordingly:

[EBADF] *Fildes* is not a valid open file descriptor, or was not opened for reading when setting a read lock or for writing when setting a write lock.

[EMFILE] *Cmd* is F_DUPFD and the maximum number of file descriptors is currently open.

[EMFILE] *Cmd* is F_SETLK or F_SETLKW, the type of lock is a read or write lock and no more file-locking headers are available (too many files have segments locked).

[EMFILE] *Cmd* is F_DUPFD and *arg.***val** is greater than or equal to the maximum number of file descriptors.

[EMFILE] *Cmd* is F_DUPFD and *arg.***val** is negative.

[EINVAL] *Cmd* is F_GETLK, F_SETLK, or F_SETLKW and *arg.***lockdes** or the data it points to is not valid, or *fildes* refers to a file that does not support locking.

[EINVAL] *Cmd* is not a valid command.

[EINVAL] *Cmd* is F_SETFL and both O_NONBLOCK and O_NDELAY are specified.

[EINTR] *Cmd* is F_SETLKW and the call was interrupted by a signal.

[EACCES] *Cmd* is F_SETLK, the type of lock (**l_type)isaread** (F_RDLCK) or write lock (F_WRLCK) and the segment of a file to be locked is already write-locked by another process, or the type is a write lock (F_WRLCK) and the segment of a file to be locked is already read- or write-locked by another process.

[ENOLCK] *Cmd* is F_SETLK or F_SETLKW, the type of lock is a read or write lock and no more file-locking headers are available (too many files have segments locked), or no more record locks are available (too many file segments locked).

[ENOLCK] *Cmd* is F_SETLK or F_SETLKW, the type of lock (**l_type**) is a read lock (F_RDLCK) or write lock (F_WRLCK) and the file is a NFS file with access bits set for enforcement mode.

[ENOLCK] *Cmd* is F_GETLK, F_SETLK, or F_SETLKW, the file is a NFS file, and a system error occurred on the remote node.

[EDEADLK] *Cmd* is F_SETLKW, when the lock is blocked by a lock from another process and sleeping (waiting) for that lock to become free. This causes a deadlock situation.

[EFAULT] *Cmd* is either F_GETLK, F_SETLK or F_SETLKW, and *arg* points to an illegal address.

RETURN VALUE

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

F_DUPFD A new file descriptor.

F_GETFD Value of close-on-exec flag (only the low-order bit is defined).

F_SETFD Value other than −1.

F_GETFL Value of file status flags and access modes.

F_SETFL Value other than −1.

F_GETLK Value other that −1.

F_SETLK Value other than −1.

F_SETLKW Value other than −1.

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

AUTHOR

Fcntl was developed by HP, AT&T and the University of California, Berkeley.

APPLICATION USAGE

Because in the future the external 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, for example:

```
flk->l_type = F_RDLCK;
if (fcntl(fd, F_SETLK, flk) == -1)
        if ((errno == EACCES) || (errno == EAGAIN))
                /*
                 * section locked by another process,
                 * check for either EAGAIN or EACCES
                 * due to different implementations
                 */
        else if ...
                /*
                 * check for other errors
                 */
```

SEE ALSO

chmod(2), close(2), exec(2), lockf(2), open(2), read(2), write(2), fcntl(5).
lockd(1M), statd(1M), in *NFS Services Reference Pages.*

FUTURE DIRECTIONS

The error condition which currently sets **errno** to EACCES will instead set **errno** to EAGAIN (see also APPLICATION USAGE above).

STANDARDS CONFORMANCE

fcntl: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME
fork – create a new process

SYNOPSIS
#include <sys/types.h>
pid_t fork ()

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

real, effective, and saved user ID
real, effective, and saved group ID
list of supplementary group IDs (see *getgroups*(2))
process group ID
environment
file descriptors
close-on-exec flags (see *exec*(2))
signal handling settings (SIG_DFL, SIG_IGN, *address*)
signal mask (see *sigvector*(2))
profiling on/off status (see *profil*(2))
command name in the accounting record (see *acct*(4))
nice value (see *nice*(2))
all attached shared memory segments (see *shmop*(2))
current working directory
root directory (see *chroot*(2))
file mode creation mask (see *umask*(2))
file size limit (see *ulimit*(2))
real-time priority (see *rtprio*(2))

Each of the child's file descriptors shares a common open file description with the corresponding file descriptor of the parent. This implies that changes to the file offset, file access mode, and file status flags of file descriptors in the parent also affect those in the child, and vice-versa.

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

The child process has a unique process ID. The child process ID also does not match any active process group ID.

The child process has a different parent process ID (which is the process ID of the parent process).

The set of signals pending for the child process is initialized to the empty set.

The trace flag (see *ptrace*(2) PT_SETTRC request) is cleared in the child process.

The AFORK flag in the **ac_flags** component of the accounting record is set in the child process.

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

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

The child process's values of **tms_utime**, **tms_stime**, **tms_cutime**, and **tms_cstime** are set to zero; see *times*(2).

The time left until an alarm clock signal is reset to 0 (clearing any pending alarm), and all interval timers are set to 0 (disabled).

The *vfork*(2) system call can be used to fork processes more quickly than *fork*, but has some restrictions. See *vfork*(2) for details.

RETURN VALUE

Upon successful completion, *fork* returns a value of **0** to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of **−1** is returned to the parent process, no child process is created, and **errno** is set to indicate the error.

The parent and child processes resume execution immediately after the *fork* call; they are distinguished by the value returned by *fork*.

ERRORS

Fork fails and no child process is created if one or more of the following is true:

[EAGAIN] The system-imposed limit on the total number of processes under execution would be exceeded.

[EAGAIN] The system-imposed limit on the total number of processes under execution by a single user would be exceeded.

[ENOMEM] There is insufficient swap space and/or physical memory available in which to create the new process.

WARNINGS

Standard I/O streams (see *stdio*(3S)) are duplicated in the child. Therefore, if *fork* is called after a buffered I/O operation without first closing or flushing the associated standard I/O stream (see *fclose*(3S)), the buffered input or output might be duplicated.

AUTHOR

Fork was developed by AT&T, the University of California, Berkeley, and HP.

SEE ALSO

acct(2), chroot(2), exec(2), exit(2), fcntl(2), getgroups(2), lockf(2), nice(2), plock(2), profil(2), ptrace(2), rtprio(2), semop(2), setuid(2), setpgrp(2), shmop(2), signal(5), times(2), ulimit(2), umask(2), vfork(2), wait(2), fclose(3S), stdio(3S), acct(4).

STANDARDS CONFORMANCE

fork: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

fsctl – file system control

SYNOPSIS

#include <sys/cdfsdir.h>
#include <sys/cdfs.h>

int fsctl(*fildes, command, outbuf, outlen***)**
int *fildes, command, outlen;*
char **outbuf;*

DESCRIPTION

Fsctl provides for access to file-system-specific information. *Fildes* is an open file descriptor for a file in the file system of interest. The possible values for *command* depend on the type of file system. Currently, defined *commands* exist only for the *cdfs* file system (see **sys/cdfsdir.h**).

Outbuf is a pointer to the data area in which data is returned from the file system. *Outlen* gives the length of the data area pointed to by *outbuf.*

The *cdfs commands* are:

CDFS_DIR_REC
: Returns the directory record for the file or directory indicated by *fildes.* The record is returned in a structure of type *cddir,* defined in *sys/cdfsdir.h.*

CDFS_XAR
: Returns the extended attribute record, if any, for the file or directory indicated by *fildes.* Because the size of an extended attribute record varies, be sure *outbuf* points to a data area of sufficient size. To find the necessary size, do the following:

1. Use *statfs*(2). to get the logical block size of the *cdfs* volume.
2. Use an *fsctl* call with the CDFS_DIR_REC command to get the extended attribute record size (in blocks) for the file or directory of interest. The *mincdd_xar_len* field in the returned structure contains the size of the extended attribute record in logical blocks. (If this field is zero, the file or directory has no extended attribute record.)
3. Multiply *mincdd_xar_len* by the logical block size obtained in step 1 to get the total space needed.
4. Once you get the extended attribute record, cast *outbuf* into a pointer to a structure of type *cdxar_iso* (defined in **sys/cdfsdir.h**). This enables you to access those fields which are common to all extended attribute records. (See the EXAMPLES section of this manual entry for an example of this process.)

   If the extended attribute record contains additional system use or application use data, that data will have to be accessed manually.

CDFS_AFID
: Returns the abstract file identifier for the primary volume whose root directory is specified by *fildes*, terminated with a NULL character. Note that the constant CDMAXNAMELEN defined in **sys/cdfsdir.h** gives the maximum length a file identifier can have. Thus, CDMAXNAMELEN+1 can be used for *outlen* and the size of *outbuf*.

CDFS_BFID
: Returns the bibliographic file identifier for the primary volume whose root directory is specified by *fildes*, terminated with a NULL character. CDMAX-NAMELEN+1 can be used for the value of *outlen* and the size of *outbuf*.

CDFS_CFID
: Returns the copyright file identifier for the primary volume whose root directory is specified by *fildes*, terminated with a NULL character.

CDMAXNAMELEN+1 can be used for the value of *outlen* and the size of *outbuf*.

CDFS_VOL_ID
: Returns the volume ID for the primary volume specified by *fildes*, terminated with a NULL character. The maximum size of the volume ID is 32 bytes, so a length of 33 can be used for *outlen* and the size of *utbuf*.

CDFS_VOL_SET_ID
: Returns the volume set ID for the primary volume specified by *fildes*, terminated with a NULL character. The maximum size of the volume set ID is 128 bytes, so a length of 129 can be used for *outlen* and the size of *outbuf*.

EXAMPLES

The following code segment gets the extended attribute record for a file on a *cdfs* volume. The filename is passed in as the first argument to the routine. Note that error checking is omitted for brevity.

```
#include <sys/types.h>
#include <sys/vfs.h>
#include <fcntl.h>
#include <sys/cdfsdir.h>
main(argc, argv)
int argc;
char *argv[];
{
   int fildes, size = 0;
   char *malloc(), *outbuf;
   struct statfs buf;
   struct cddir cdrec;
   struct cdxar_iso *xar;
   .
   .
   .
   statfs(argv[1], &buf);   /* get logical block size */

   fildes = open(argv[1], O_RDONLY);  /* open file arg */

   /* get directory record for file arg */
   fsctl(fildes, CDFS_DIR_REC, &cdrec, sizeof(cdrec));

   size = buf.f_bsize * cdrec.cdd_min.mincdd_xar_len;   /* compute size */

   if(size) {   /* if size != 0 then there is an xar */
      outbuf = malloc(size);   /* malloc sufficient memory */

      fsctl(fildes, CDFS_XAR, outbuf, size);  /* get xar */

      xar = (struct cdxar_iso *)outbuf;  /* cast outbuf to access fields */
         .
         .
         .
   }
```

```
        .
        .
        .
    }
```

RETURN VALUE

Fsctl returns the number of bytes read if successful. If an error occurs, -1 is returned and errno is set to indicate the error:

[EBADF]	*Fildes* is not a valid open file descriptor.
[EFAULT]	*Outbuf* points to an invalid address.
[ENOENT]	The requested information does not exist.
[EINVAL]	*Command* is not a valid command.
[EINVAL]	*Outlen* is negative, or *fildes* does not refer to a CDFS file system.

SEE ALSO

statfs(2), cdfs(4), cdfsdir(4), cdnode(4), cdrom(4).

NAME

fsync – synchronize a file's in-core state with its state on disk

SYNOPSIS

int fsync(fildes)
int fildes;

DESCRIPTION

Fsync causes all modified data and attributes of *fildes* to be moved to a permanent storage device. This normally results in all in-core modified copies of buffers for the associated file to be written to a disk. *Fsync* applies to ordinary files, and applies to block special devices on systems which permit I/O to block special devices.

Fsync should be used by programs which require a file to be in a known state; for example in building a simple transaction facility.

ERRORS

Fsync will fail if one of the following conditions is true and *errno* will be set accordingly:

[EBADF] *Fildes* is not a valid descriptor.

[EINVAL] *Fildes* refers to a file type to which *fsync* does not apply.

RETURN VALUE

A 0 value is returned on success. A −1 value indicates an error.

BUGS

The current implementation of this call is expensive for large files.

AUTHOR

Fsync was developed by the Hewlett-Packard Company, and the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

fcntl(2), fcntl(5), open(2), select(2), sync(2), sync(1M).

STANDARDS CONFORMANCE

fsync: XPG3

NAME

ftime – get date and time more precisely

SYNOPSIS

#include <sys/types.h>
#include <sys/timeb.h>
ftime(tp)
struct timeb *tp;

REMARKS

This facility is provided for backwards compatibility with Version 7 systems. Either *time* or *gettimeofday* should be used for all new code.

DESCRIPTION

Ftime entry 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 in seconds since 00:00:00 GMT, January 1, 1970, up to 1000 milliseconds of more-precise interval, the local timezone (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. *Gettimeofday* should be consulted for more details on the meaning of the timezone field.

This call can be accessed by giving the **-lV7** option to *ld*(1).

Ftime can fail for exactly the same reasons as *gettimeofday*(2).

SEE ALSO

date(1), gettimeofday(2), stime(2), time(2), ctime(3C).

BUGS

The millisecond value usually has a granularity greater than one due to the resolution of the system clock. Depending on any granularity (particularly of one) will render code non-portable.

NAME

getaccess – get a user's effective access rights to a file

SYNOPSIS

```
#include <unistd.h>
#include <limits.h>
#include <sys/getaccess.h>

int getaccess (path, uid, ngroups, gidset, label, privs)
char *path;
int uid;
int ngroups;
int gidset[];
void *label;
void *privs;
```

Remarks:

To ensure continued conformance with emerging industry standards, features described in this manual entry are likely to change in a future release.

DESCRIPTION

Getaccess identifies the access rights (read, write, execute/search) a specific user ID has to an existing file. *Path* points to a path name of a file. If the call succeeds, it returns a value of zero or greater, representing the specified user's effective access rights (modes) to the file. The rights are expressed as the OR of bits (R_OK, W_OK, and X_OK) whose values are defined in the header **<unistd.h>**. A return of zero means that access is denied.

The *uid* parameter is a user ID. Special values, defined in **<sys/getaccess.h>**, represent the calling process's effective, real, or saved user ID:

UID_EUID	Effective user ID.
UID_RUID	Real user ID.
UID_SUID	Saved user ID.

Ngroups is the number of group IDs in *gidset*, not to exceed NGROUPS_MAX + 1 (NGROUPS_MAX is defined in **<limits.h>**). If the *ngroups* parameter is positive, the *gidset* parameter is an array of group ID values to use in the check. If *ngroups* is a recognized negative value, *gidset* is ignored. Special negative values of *ngroups*, defined in **<sys/getaccess.h>**, represent various combinations of the process's effective, real, or saved user ID and its supplementary groups list:

NGROUPS_EGID	Use process's effective group ID only.
NGROUPS_RGID	Use process's real group ID only.
NGROUPS_SGID	Use process's saved group ID only.
NGROUPS_SUPP	Use process's supplementary groups only.
NGROUPS_EGID_SUPP	Use process's effective group ID plus supplementary groups.
NGROUPS_RGID_SUPP	Use process's real group ID plus supplementary groups.
NGROUPS_SGID_SUPP	Use process's saved group ID plus supplementary groups.

The *label* and *privs* parameters are placeholders for future extensions. For now, the values of these parameters must be **(void *) 0**.

The access check rules for access control lists are described in *acl*(5). In addition, the W_OK bit is cleared for files on read-only file systems or shared-text programs being executed. Note that as in *access*(2), the X_OK bit is not turned off for shared-text programs open for writing because there is no easy way to know that a file open for writing is a shared-text program.

If the caller's user ID is 0, or if it is UID_EUID, UID_RUID, or UID_SUID (see **<sys/getaccess.h>**) and the process's respective user ID is 0, then R_OK and W_OK are always set, except when

W_OK is cleared for files on read-only file systems or shared-text programs being executed. X_OK is set if and only if the file is not a regular file or the execute bit is set in any of the file's ACL entries.

Getaccess checks each directory component of *path* by first using the caller's effective user ID, effective group ID, and supplementary groups list, regardless of the user ID specified. An error occurs, distinct from "no access allowed," if the caller cannot search the path to the file. (In this case it is inappropriate for the caller to learn anything about the file.)

Comparison of *access*(2) and *getaccess*(2)

The following table compares various attributes of *access* and *getaccess*.

access()	getaccess()
checks all ACL entries	same
uses real uid, real gid, and supplementary groups list	uses specified uid and groups list; macros available for typical values
checks specific mode value, returns succeed or fail	returns all mode bits, each on or off
checks path to file using caller's effective IDs	same
W_OK false if shared-text file currently being executed	same
W_OK false if file on read-only file system	same
X_OK not modified for file currently open for writing	same
R_OK and W_OK always true for superuser (except as above)	same
X_OK always true for superuser	X_OK true for super-user if file is not a regular file OR execute is set in any ACL entry

RETURN VALUE

Upon successful completion, *getaccess* returns a non-negative value representing the access rights of the specified user to the specified file. If an error occurs, a value of −1 is returned and the error code is stored in the global variable **errno**.

ERRORS

Getaccess fails if any of the following is true:

[EACCES] A component of the *path* prefix denies search permission to the caller.

[EFAULT] *Path* or *gidset* points outside the allocated address space of the process. The reliable detection of this error is implementation dependent.

[EINVAL] *Ngroups* is invalid; *ngroups* is either zero, an unrecognized negative value, or a value larger than NGROUPS + 1.

[EINVAL] *Gidset* contains an invalid group ID value.

EINVAL] The value of *label* or *privs* is not a null pointer.

[ELOOP] Too many symbolic links were encountered in translating the *path* name.

[ENAMETOOLONG]
The length of the specified path name exceeds PATH_MAX bytes, or the length

of a component of the path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.

[ENOENT] The named file does not exist (for example, *path* is null or a component of *path* does not exist).

[ENOTDIR] A component of the *path* prefix is not a directory.

[EOPNOTSUPP] *getaccess*() is not supported on some types of remote files.

EXAMPLES

The following call determines the caller's effective access rights to file "test," and succeeds if the user has read access:

```
#include <unistd.h>
#include <sys/getaccess.h>

int mode;
mode = getaccess ("test", UID_EUID, NGROUPS_EGID_SUPP,
(int *) 0, (void *) 0, (void *) 0);

if ((mode >= 0) && (mode & R_OK)) ...
```

Here's one way to test access rights to file "/tmp/hold" for user ID 23, group ID 109:

```
int gid = 109;
int mode;

mode = getaccess ("/tmp/hold", 23, 1, & gid,
(void *) 0, (void *) 0);
```

Should the need arise, the following code builds a *gidset* that includes the process's effective group ID:

```
#include <limits.h>

int gidset [NGROUPS_MAX + 1];
int ngroups;

gidset [0] = getegid();
ngroups = 1 + getgroups (NGROUPS_MAX, & gidset [1]);
```

AUTHOR

Getaccess was developed by HP.

SEE ALSO

access(2), chmod(2), getacl(2), setacl(2), stat(2), acl(5), unistd(5).

NAME

getacl, fgetacl – get access control list (ACL) information

SYNOPSIS

```
#include <unistd.h>
#include <sys/acl.h>

int getacl (path, nentries, acl)
char    *path;
int     nentries;
struct  acl_entry acl[];

int fgetacl (fildes, nentries, acl)
int     fildes;
int     nentries;
struct  acl_entry acl[];
```

Remarks:

To ensure continued conformance with emerging industry standards, features described in this manual entry are likely to change in a future release.

DESCRIPTION

Getacl returns a complete listing of all ACL entries (*uid.gid, mode*) in an existing file's access control list. *Path* points to a path name of a file.

Similarly, *fgetacl* returns a complete listing of all ACL entries for an open file known by the file descriptor *fildes*.

The *nentries* parameter is the number of entries being reported on, and is never more than the constant NACLENTRIES defined in **<sys/acl.h>**. If *nentries* is non-zero, it must be at least as large as the number of entries in the file's ACL, including base entries (see *setacl*(2)). *Getacl* returns the number of entries in the file's ACL, as well as the ACL entries themselves in the array of structures *acl* declared by the calling program.

If *nentries* is zero, *getacl* returns the number of entries in the file's ACL, including base ACL entries, and *acl* is ignored.

Entries are reported in groups of decreasing order of specificity (see *setacl*(2)), then sorted in each group by user ID and group ID. The content of array entries beyond the number of defined entries for the file is undefined.

RETURN VALUE

Upon successful completion, *getacl* and *fgetacl* return a non-negative value. If an error occurs, a value of −1 is returned, and the global variable **errno** is set to indicate the error.

ERRORS

Getacl or *fgetacl* fail to modify the *acl* array if any of the following is true:

[ENOTDIR] A component of the *path* prefix is not a directory.

[ENOENT] The named file does not exist (for example, *path* is null or a component of *path* does not exist).

[EBADF] *Fildes* is not a valid file descriptor.

[EACCES] A component of the *path* prefix denies search permission.

[EFAULT] *Path* or a portion of *acl* to be written points outside the allocated address space of the process.

[EINVAL] *Nentries* is non-zero and less than the number of entries in the file's ACL, or it is greater than NACLENTRIES.

[EOPNOTSUPP] *Getacl* is not supported on remote files by some networking services.

[ENFILE] The system file table is full.

[ENAMETOOLONG]
The length of *path* exceeds PATH_MAX bytes, or the length of a component of *path* exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.

[ELOOP] Too many symbolic links were encountered in translating the *path* name.

EXAMPLES

The following call returns the number of entries in the ACL on file "/users/bill/mcfile".

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

entries = getacl ("/users/bill/mcfile", 0, (struct acl_entry *) 0);
```

The following call returns in *acl* all entries in the ACL on the file opened with file descriptor 5.

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

int nentries;
struct acl_entry acl [NACLENTRIES];

entries = fgetacl (5, NACLENTRIES, acl);
```

DEPENDENCIES

RFA and NFS

Getacl and *fsetacl* are not supported on remote files.

AUTHOR

Getacl and *fgetacl* were developed by HP.

SEE ALSO

access(2), chmod(2), getaccess(2), setacl(2), stat(2), unistd(5).

NAME

getaudid – get the audit ID (aid) for the current process

SYNOPSIS

#include <sys/audit.h>

int getaudid ()

DESCRIPTION

Getaudid returns the audit ID (*aid*) for the current process. This call is restricted to the superuser.

RETURN VALUE

Upon successful completion, the audit ID is returned; otherwise, a **–1** is returned.

ERRORS

Getaudid fails if the following is true:

[EPERM] The caller is not a superuser.

AUTHOR

Getaudid was developed by HP.

SEE ALSO

setaudid(2).

NAME

getaudproc – get the audit process flag for the calling process

SYNOPSIS

#include <sys/audit.h>

int getaudproc ()

DESCRIPTION

Getaudproc returns the audit process flag for the calling process. the audit process flag (*u_audproc*) determines whether the process, run by a given user, should be audited. The process is audited if the returned flag is 1. If the returned flag is 0, the process is not audited. This call is restricted to the superuser.

RETURN VALUE

Upon successful completion, the audit process flag is returned; otherwise, a **–1** is returned.

ERRORS

Getaudproc fails if the following is true:

[EPERM] The caller is not a superuser.

AUTHOR

Getaudproc was developed by HP.

SEE ALSO

setaudproc(2).

NAME

getcontext – return the process context for context dependent file search

SYNOPSIS

int getcontext(contextbuf,length)
char *contextbuf;
int length;

DESCRIPTION

Getcontext reads the per-process context (see *context*(5)) into the buffer pointed to by *contextbuf*. The context is returned as a null-terminated string containing a blank-separated list of names. The function value returned by *getcontext* is the length of this string, including the null terminator. If this string, including the null terminator, is less than *length* bytes, a truncated, null-terminated string of *length* bytes is returned. In particular, if *length* is zero, only the function value is returned.

RETURN VALUE

Upon successful completion, the length of the context string including the null terminator is returned. Otherwise, a value of −1 is returned and **errno** is set to indicate the error.

ERRORS

Getcontext may fail if the following is true:

[EFAULT] *Contextbuf* points to an illegal address. Reliable detection of this error is not guaranteed.

EXAMPLES

In the following example *getcontext* is called once with a *length* parameter of zero to determine the size of a buffer to allocate for the context.

```
int length;
char *contextbuf;

        length = getcontext ((char *)0, 0);
        contextbuf = malloc (length);
        (void) getcontext (contextbuf, length);
```

AUTHOR

Getcontext was developed by HP.

SEE ALSO

context(5), cdf(4), getcontext(1).

NAME

getdirentries – get entries from a directory in a filesystem-independent format

SYNOPSIS

```
#include <ndir.h>

int getdirentries(fildes, buf, nbytes, basep)
int fildes;
char *buf;
int nbytes;
long *basep;
```

DESCRIPTION

Getdirentries places directory entries from the directory referenced by the file descriptor *fildes* into the buffer pointed to by *buf*, in a filesystem-independent format. Up to *nbytes* of data are transferred. *Nbytes* must be greater than or equal to the block size associated with the file; see *stat*(2). Smaller block sizes can cause errors on certain file systems.

The data in the buffer consists of a series of **direct** structures, each containing the following entries:

```
unsigned long  d_fileno;
unsigned short d_reclen;
unsigned short d_namlen;
char           d_name[MAXNAMLEN + 1];
```

The **d_fileno** entry is a number unique for each distinct file in the file system. Files linked by hard links (see *link*(2)) have the same **d_fileno**. The **d_reclen** entry identifies the length, in bytes, of the directory record. The **d_name** entry contains a null-terminated file name. The **d_namlen** entry specifies the length of the file name. Thus the actual size of **d_name** can vary from 2 to MAXNAMLEN + 1. Note that the **direct** structures in the buffer are not necessarily tightly packed. The **d_reclen** entry must be used as an offset from the beginning of a **direct** structure to the next structure, if any.

The return value of the system call is the actual number of bytes transferred. The current position pointer associated with *fildes* is set to point to the next block of entries. The pointer is not necessarily incremented by the number of bytes returned by *getdirentries*. If the value returned is zero, the end of the directory has been reached.

The current position pointer is set and retrieved by *lseek*(2). *Getdirentries* writes the position of the block read into the location pointed to by *basep*. The current position pointer can be set safely only to a value previously returned by *lseek*(2), to a value previously returned in the location pointed to by *basep*, or to zero. Any other manipulation of the position pointer causes undefined results.

RETURN VALUE

If successful, the number of bytes actually transferred is returned. Otherwise, **–1** is returned and the global variable **errno** is set to indicate the error.

ERRORS

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

[EBADF]	*Fildes* is not a valid file descriptor open for reading.
[EFAULT]	Either *buf* or *basep* points outside the allocated address space.
[EINTR]	A read from a slow device was interrupted by the delivery of a signal before any data arrived.
[EIO]	An I/O error occurred while reading from or writing to the file system.

AUTHOR

Getdirentries was developed by Sun Microsystems, Inc.

SEE ALSO

open(2), lseek(2).

NAME

getevent – get events and system calls that are currently being audited

SYNOPSIS

#include <sys/audit.h>

int getevent (a_syscall, a_event)
struct aud_type *a_syscall;
struct aud_event_tbl *a_event;

DESCRIPTION

Getevent gets the events and system calls being audited. The events are returned in a table pointed to by *a_event*. The system calls are returned in a table pointed to by *a_syscall*. This call is restricted to the superuser.

RETURN VALUE

Upon successful completion, a value of **0** is returned; otherwise, a **−1** is returned.

ERRORS

Getevent fails if the following is true:

[EPERM] The caller is not a superuser.

AUTHOR

Getevent was developed by HP.

SEE ALSO

setevent(2), audevent(1M).

NAME

getgroups – get group access list

SYNOPSIS

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

int getgroups(ngroups, gidset)
int ngroups;
gid_t  *gidset;
```

DESCRIPTION

Getgroups gets the current group access list of the user process and stores it in the array *gidset*. The parameter *ngroups* indicates the number of entries which may be placed in *gidset.* No more than NGROUPS, as defined in *<sys/param.h>*, will ever be returned.

As a special case, if the *ngroups* argument is zero, *getgroups* returns the number of group entries for the process. In this case, the array pointed to by the *gidset* argument is not modified.

EXAMPLES

The following call to *getgroups*(2) retrieves the group access list of the calling process and stores the group ids in array mygidset:

```
int ngroups = NGROUPS;
gid_t mygidset[NGROUPS];
int ngrps;

ngrps = getgroups (ngroups, mygidset);
```

RETURN VALUE

A non-negative value indicates that the call succeeded, and is the number of elements returned in *gidset*. A value of −1 indicates that an error occurred, and the error code is stored in the global variable **errno**.

ERRORS

The possible errors for *getgroups* are:

[EFAULT] *Gidset* specifies an invalid address. The reliable detection of this error will be implementation dependent.

[EINVAL] The argument *ngroups* is not zero and is less than the number of groups in the current group access list of the process.

AUTHOR

Getgroups was developed by HP and the University of California, Berkeley

SEE ALSO

setgroups(2), initgroups(3C)

STANDARDS CONFORMANCE

getgroups: XPG3, POSIX.1, FIPS 151-1

NAME

gethostname – get name of current host

SYNOPSIS

int gethostname(hostname, size)
char *hostname;
unsigned int size;

DESCRIPTION

Gethostname returns in the array to which *hostname* points, the standard host name for the current processor as set by *sethostname*(2). *Size* specifies the length of the *hostname* array. *Hostname* is null-terminated unless insufficient space is provided.

RETURN VALUE

Gethostname returns **0** if successful. Otherwise, **–1** is returned and **errno** is set to indicate the error.

ERRORS

Gethostname can fail if the following is true:

[EFAULT] *Hostname* points to an illegal address. The reliable detection of this error is implementation dependent.

DEPENDENCIES

Series 300

Gethostname returns a non-negative integer if successful.

AUTHOR

Gethostname was developed by the University of California, Berkeley.

SEE ALSO

hostname(1), uname(1), sethostname(2), uname(2).

NAME

getitimer, setitimer – get/set value of interval timer

SYNOPSIS

```
#include <time.h>

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

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

DESCRIPTION

The system provides each process with three interval timers, defined in **<time.h>**. The *getitimer* call returns the current value for the timer specified in *which*, while the *setitimer* call sets the value of a timer (optionally returning the previous value of the timer).

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

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

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

Time values smaller than the resolution of the system clock are rounded up to this resolution. The machine-dependent clock resolution is *1/HZ* seconds, where the constant *HZ* is defined in **<sys/param.h>**. Time values larger than an implementation-specific maximum value are rounded down to this maximum. The maximum values for the three interval timers are specified by the constants MAX_ALARM, MAX_VTALARM, and MAX_PROF defined in **<sys/param.h>**. On all implementations, these values are guaranteed to be at least 31 days (in seconds).

The *which* parameter specifies which timer to use. The possible values are ITIMER_REAL, ITIMER_VIRTUAL, and ITIMER_PROF.

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

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

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

Interval timers are not inherited by a child process across a *fork*, but are inherited across an *exec*.

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

The timer used with ITIMER_REAL is also used by *alarm*(2). Thus successive calls to *alarm*, *getitimer*, and *setitimer* set and return the state of a single timer. In addition, a call to *alarm* sets the timer interval to zero.

RETURN VALUE

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

ERRORS

Getitimer or *setitimer* can fail if any of the following is true:

[EFAULT] The *value* structure specified a bad address. The reliable detection of this error will be implementation dependent.

[EINVAL] A *value* structure specified a microsecond value less that zero or greater than or equal to one million.

[EINVAL] *Which* does not specify one of the three possible timers.

EXAMPLES

The following call to *setitimer*(2) sets the real-time interval timer to expire initially after 10 seconds and every 0.5 seconds thereafter:

```
struct itimerval rttimer;
struct itimerval old_rttimer;

rttimer.it_value.tv_sec     = 10;
rttimer.it_value.tv_usec    = 0;
rttimer.it_interval.tv_sec  = 0;
rttimer.it_interval.tv_usec = 500000;

setitimer (ITIMER_REAL, &rttimer, &old_rttimer);
```

AUTHOR

Getitimer was developed by the University of California, Berkeley.

SEE ALSO

alarm(2), exec(2), gettimeofday(2), signal(5).

NAME

getpid, getpgrp, getppid, getpgrp2 – get process, process group, and parent process ID

SYNOPSIS

#include <sys/types.h>
pid_t getpid ()
pid_t getpgrp ()
pid_t getppid ()
pid_t getpgrp2 (pid)
pid_t pid;

DESCRIPTION

Getpid returns the process ID of the calling process.

Getpgrp returns the process group ID of the calling process.

Getppid returns the parent process ID of the calling process.

Getpgrp2 returns the process group ID of the specified process. If *pid* is zero, the call applies to the current process. For this to be allowed, the current process and the referenced process must be in the same session.

ERRORS

Getpgrp2 will fail if any of the following are true:

[EPERM] The current process and the specified process are not in the same session.

[ESRCH] No process can be found corresponding to that specified by *pid*.

AUTHOR

Getpid, *getppid*, *getpgrp*, and *getpgrp2* were developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

exec(2), fork(2), setpgrp(2), setpgid(2), signal(5).

STANDARDS CONFORMANCE

getpid: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

getpgrp: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

getppid: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

getprivgrp, setprivgrp – get and set special attributes for group

SYNOPSIS

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

int getprivgrp(grplist)
struct privgrp_map grplist[PRIV_MAXGRPS];

int setprivgrp(grpid, mask)
gid_t grpid;
int mask[PRIV_MASKSIZ];

DESCRIPTION

Setprivgrp associates a kernel capability with a group id. This allows subsetting of super–user like privileges for members of a particular group or groups. *Setprivgrp* takes two arguments: the integer group id and a mask of permissions. The mask is created by treating the access types defined in <sys/privgrp.h> as bit numbers (using 1 for the least significant bit). Thus, privilege number 5 would be represented by the bit 1<<(5-1) or 16. More generally, privilege **p** is represented by:

mask[((**p**-1) / BITS_PER_INT)] & (1 << ((**p**-1) % BITS_PER_INT)).

As it is possible to have more than **word size** distinct privileges, mask is a pointer to an integer array of size **PRIV_MASKSIZ**.

Setprivgrp privileges include those specified in the file **<sys/privgrp.h>**. A process may access the system call protected by a specific privileged group if it belongs to or has an effective group id of a group having access to the system call. All processes are considered to belong to the pseudo-group **PRIV_GLOBAL**.

Specifying a *grpid* of **PRIV_NONE** causes privileges to be revoked on all privileged groups having any of the privileges specified in *mask*. Specifying a *grpid* of **PRIV_GLOBAL** causes privileges to be granted to all processes.

The constant **PRIV_MAXGRPS** in <sys/privgrp.h> defines the system limit on the number of groups which can be assigned privileges. One of these is always the psuedo-group **PRIV_GLOBAL**, allowing for **PRIV_MAXGRPS**-1 actual groups.

Getprivgrp returns a table of the privileged group assignments into a user supplied structure. *Grplist* points to an array of structures of type **privgrp_map** associating a groupid with a privilege mask. Privilege masks are formed by *or*ing together elements from the access types specified in <sys/privgrp.h>. The array may have gaps in it distinguished as having a **priv_groupno** field of **PRIV_NONE**. The group number **PRIV_GLOBAL** gives the global privilege mask. Only information about groups which are in the user's group access list, or about his real or effective group id, is returned to an ordinary user. The complete set is returned to the super-user.

EXAMPLES

The following example prints out PRIV_GLOBAL and the group ids of the privilege groups to which the user belongs:

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

struct privgrp_map pgrplist[PRIV_MAXGRPS];
int i;
gid_t pgid;
```

```
getprivgrp (pgrplist);
for (i=0; i<PRIV_MAXGRPS; i++) {
        if ((pgid = pgrplist[i].priv_groupno) != PRIV_NONE) {
                if (pgid == PRIV_GLOBAL)
                        printf ("(PRIV_GLOBAL) ");
                printf ("privilege group id = %d\n", pgid);
        }
}
```

NOTES

Only the super-user may use *setprivgrp*.

ERRORS

Setprivgrp returns -1 and an error code in **errno** if:

[EPERM] The caller is not super user.

[EFAULT] *Mask* points to an illegal address. The reliable detection of this error will be implementation dependent.

[EINVAL] *Mask* has bits set for one or more unknown privileges.

[E2BIG] The request would require assigning privileges to more than **PRIV_MAXGRPS** groups.

Getprivgrp returns -1 and an error code in **errno** if:

[EFAULT] *Grplist* points to an illegal address. The reliable detection of this error will be implementation dependent.

Both calls return 0 on success.

DEPENDENCIES

HP Clustered Environment:

In a clustered environment privilege groups are maintained separately on each machine in the cluster. The CHOWN privilege from diskless nodes is determined by the privilege groups set up on the cluster server.

AUTHOR

Getprivgrp was developed by HP.

SEE ALSO

getprivgrp(1), setgroups(2), setprivgrp(1M), privgrp(4).

NAME

gettimeofday, settimeofday – get/set date and time

SYNOPSIS

```
#include <time.h>

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

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

DESCRIPTION

Gettimeofday returns and *settimeofday* sets the system's notion of the current Greenwich time and the system's notion of the current time zone. Time is expressed in seconds and microseconds since midnight January 1, 1970.

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

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

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

The **timezone** structure indicates the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Savings time applies locally during the appropriate part of the year. Programs should use this timezone information only in the absence of the TZ environment variable.

Only the super-user may set the time of day.

EXAMPLES

The following example calls *gettimeofday*(2) twice. It then computes the lapsed time between the calls in seconds and microseconds and stores the result in a timeval structure:

```
struct timeval    first,
                  second,
                  lapsed;
struct timezone tzp;


        gettimeofday (&first, &tzp);

        /* lapsed time */

        gettimeofday (&second, &tzp);

        if (first.tv_usec > second.tv_usec) {
                  second.tv_usec += 1000000;
                  second.tv_sec--;
        }
        lapsed.tv_usec = second.tv_usec - first.tv_usec;
```

lapsed.tv_sec = second.tv_sec - first.tv_sec;

RETURN VALUE

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

ERRORS

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

[EFAULT] An argument address referenced invalid memory. The reliable detection of this error will be implementation dependent.

[EPERM] A user other than the super-user attempted to set the time.

Clustered Systems

In an HP Clustered Environment, setting the time of day sets the date and timezone on all systems in the cluster.

WARNINGS

The microsecond value usually has a granularity much greater than one due to the resolution of the system clock. Depending on any granularity (particularly of one) will render code non-portable.

DEPENDENCIES

Series 300

Gettimeofday has a granularity of 4 microseconds.

AUTHOR

Gettimeofday was developed by the University of California, Berkeley.

SEE ALSO

date(1), stime(2), time(2), ctime(3C).

NAME

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

SYNOPSIS

#include <sys/types.h>

uid_t getuid ()

uid_t geteuid ()

gid_t getgid ()

gid_t getegid ()

DESCRIPTION

Getuid returns the real-user-ID of the calling process.

Geteuid returns the effective-user-ID of the calling process.

Getgid returns the real-group-ID of the calling process.

Getegid returns the effective-group-ID of the calling process.

There is no way to ascertain the saved-user-ID or saved-group-ID of a process.

SEE ALSO

setuid(2).

STANDARDS CONFORMANCE

getuid: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

getegid: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

geteuid: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

getgid: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

ioctl – control device

SYNOPSIS

#include <sys/ioctl.h>

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

DESCRIPTION

Ioctl performs a variety of functions on character special files (devices). The write-ups of various devices in Section (7) discuss how *ioctl* applies to them. The type of *arg* is dependent on the specific *ioctl* call, as described in Section (7).

Request is made up of several fields. They encode the size and direction of the argument (referenced by *arg*), as well as the desired command. An enumeration of the request fields are:

IOC_IN — Argument is read by the driver. (That is, the argument is copied from the application to the driver.)

IOC_OUT — Argument is written by the driver. (That is, the argument is copied from the driver to the application.)

IOCSIZE_MASK — Number of bytes in the passed argument. A nonzero size indicates that *arg* is a pointer to the passed argument. A zero size indicates that *arg* is the passed argument (if the driver wants to use it), and is not treated as a pointer.

IOCCMD_MASK — The request command itself.

When both IOC_IN and IOC_OUT are zero, it can be assumed that *request* is not encoded for size and direction, for compatibility purposes. Requests which do not require any data to be passed and requests which use *arg* as a value (as opposed to a pointer), have the IOC_IN bit set to one and the IOCSIZE_MASK field set to zero.

The following macros are used to create the request argument. X and y are concatenated ((x<<8) | y) to form IOCCMD and shifted into the proper location according to **IOCCMD_MASK.** T is the type (e.g. struct hpib_cmd) of the actual argument that the request references, and its size is taken and shifted into the appropriate place according to **IOCSIZE_MASK.**

_IOR(x,y,t) — Sets IOC_OUT and initializes the values at IOCCMD_MASK and IOCSIZE_MASK accordingly.

_IOW(x,y,t) — Sets IOC_IN and initializes the values at IOCCMD_MASK and IOCSIZE_MASK accordin gly.

_IOWR(x,y,t) — Sets both IOC_IN and IOC_OUT and initializes the values at IOCCMD_MASK and IOCSIZE_MASK.

Note: any data structure referenced by *arg* may *not* contain any pointers.

RETURNS

If an error has occurred, a value of −1 is returned and **errno** is set to indicate the error.

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

[EBADF] — *Fildes* is not a valid open file descriptor.

[ENOTTY] — The request is not appropriate to the selected device.

[EINVAL] — *Request* or *arg* is not valid.

[EINTR] — A signal was caught during the *ioctl* system call.

[EPERM] Typically this error indicates that an ioctl request was attempted that is forbidden in some way to the calling process.

WARNINGS

Check all references to *signal*(5) for appropriateness on systems that support *sigvector*(2). *Sigvector*(2) can affect the behavior described on this page.

AUTHOR

Ioctl was developed by AT&T Bell Laboratories and the Hewlett-Packard Company.

SEE ALSO

ioctl(5), termio(7).

STANDARDS CONFORMANCE

ioctl: SVID2, XPG2

NAME

kill, raise – send a signal to a process or a group of processes

SYNOPSIS

```
#include <signal.h>

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

int raise (sig)
int sig;
```

DESCRIPTION

Kill sends a signal to a process or a group of processes. The process or group of processes to which the signal is to be sent is specified by *pid*. The signal to be sent is specified by *sig* and is either one from the list given in *signal*(2), or **0**.

Raise sends signal *sig* to the executing program. The signal to be sent is specified by *sig* and is either one from the list given in *signal*(2), or **0**.

If *sig* is **0** (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

The real or effective user ID of the sending process must match the real or saved user ID of the receiving process, unless the effective user ID of the sending process is super-user. As a single special case, the continue signal SIGCONT can be sent to any process that is a member of the same session as the sending process.

The value KILL_ALL_OTHERS is defined in the file <**sys/signal.h**> and is guaranteed not to be the ID of any process in the system or the negation of the ID of any process in the system.

If *pid* is greater than zero and not equal to KILL_ALL_OTHERS, *sig* is sent to the process whose process ID is equal to *pid*. *Pid* can equal **1** unless *sig* is SIGKILL or SIGSTOP.

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

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

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

If *pid* is KILL_ALL_OTHERS, *kill* behaves much as when *pid* is equal to **–1**, except that *sig* is not sent to the calling process.

If *pid* is negative but not **–1** or KILL_ALL_OTHERS, *sig* is sent to all processes (excluding special system processes) whose process group ID is equal to the absolute value of *pid*, and whose real and/or effective user ID meets the constraints described above for matching user IDs.

ERRORS

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

[EINVAL]	*Sig* is not a valid signal number or zero.
[EINVAL]	*Sig* is SIGKILL or SIGSTOP and *pid* is **1** (proc1).
[EPERM]	The user ID of the sending process is not super-user and its real or effective user ID does not match the real or saved user ID of the receiving process.
[EPERM]	The sending and receiving processes are not in the same session.

[ESRCH] No process or process group can be found corresponding to that specified by *pid*.

Raise will fail and no signal will be sent if the following is true:

[EINVAL] *Sig* is not a valid signal number or zero.

RETURN VALUE

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

AUTHOR

Kill was developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

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

STANDARDS CONFORMANCE

kill: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

raise: ANSI C

NAME

link – link to a file

SYNOPSIS

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

DESCRIPTION

Path1 points to a path name naming an existing file. *Path2* points to a path name naming the new directory entry to be created. *Link* creates a new link (directory entry) for the existing file.

ERRORS

Link will fail and no link will be created if one or more of the following are true:

[ENOTDIR] A component of either path prefix is not a directory.

[ENOENT] A component of either path prefix does not exist.

[ENOSPC] The directory to contain the file cannot be extended.

[EACCES] A component of either path prefix denies search permission.

[ENOENT] The file named by *path1* does not exist.

[EEXIST] The link named by *path2* exists.

[EPERM] The file named by *path1* is a directory and the effective user ID is not super-user.

[EXDEV] The link named by *path2* and the file named by *path1* are on different logical devices (file systems).

[ENOENT] *Path2* points to a null path name.

[EACCES] The requested link requires writing in a directory that does not permit writing.

[EROFS] The requested link requires writing in a directory on a read-only file system.

[EFAULT] *Path* points outside the allocated address space of the process. The reliable detection of this error will be implementation dependent.

[ENOENT] *Path1* or *path2* is null.

[EMLINK] The maximum number of links to a file would be exceeded.

[ENAMETOOLONG] Either path specified exceeds PATH_MAX bytes, or a component of either path specified exceeds NAME_MAX while POSIX_NO_TRUNC is in effect.

[ELOOP] Too many symbolic links were encountered in translating either path name.

RETURN VALUE

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

SEE ALSO

cp(1), link(1M), symlink(2), symlink(4), unlink(2).

STANDARDS CONFORMANCE

link: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

lockf – provide semaphores and record locking on files

SYNOPSIS

#include <unistd.h>

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

DESCRIPTION

Lockf will allow regions of a file to be used as semaphores (advisory locks) or accessible only by the locking process (enforcement mode record locks). Other processes that attempt to access the locked resource will either return an error or sleep until the resource becomes unlocked. All the locks for a process are removed when the process closes the file or terminates.

Fildes is an open file descriptor. The file descriptor must have been opened with write-only permission (O_WRONLY) or read-write permission (O_RDWR) in order to establish a lock with this function call (see *open*(2)).

If the calling process is a member of a group that has the PRIV_LOCKRDONLY privilege (see *setprivgrp*(2)), it can also use *lockf* to lock files opened with read-only permission (O_RDONLY).

Function is a control value that specifies the action to be taken. The permissible values for *function* are defined in **<unistd.h>** as follows:

```
#define F_ULOCK   0   /* unlock a region */
#define F_LOCK    1   /* lock a region */
#define F_TLOCK   2   /* test and lock a region */
#define F_TEST    3   /* test region for lock */
```

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 region. *Lockf* returns zero if the region is accessible and **–1** if it is not; in this case **errno** will be set to EACCES. **F_LOCK** and **F_TLOCK** both lock a region of a file if the region is available. **F_ULOCK** removes locks from a region of the file.

Size is the number of contiguous bytes to be locked or unlocked. The resource to be locked starts at the current offset in the file, and extends forward for a positive *size*, and backward for a negative *size* (the preceding bytes up to but not including the current offset). If *size* is zero, the region from the current offset through the end of the largest possible file is locked (that is, from the current offset through the present or any future end-of-file). An area need not be allocated to the file in order to be locked, as such locks may exist past the end of the file.

The regions locked with **F_LOCK** or **F_TLOCK** may, in whole or part, contain or be contained by a previously locked region for the same process. When this occurs or if adjacent regions occur, the regions are combined into a single region. 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 region 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, and the **F_TLOCK** will return an EACCES error if the region is already locked by another process.

F_ULOCK requests may, in whole or part, release one or more locked regions controlled by the process. When regions are not fully released, the remaining regions are still locked by the process. Releasing the center section of a locked region requires an additional element in the table of active locks. If this table is full, an EDEADLK error is returned, and the requested region is not released.

Regular files with the file mode of **S_ENFMT** not having the group execute bit set will have an enforcement policy enabled. With enforcement enabled, reads and writes that would access a locked region will sleep until the entire region is available if **O_NDELAY** is cleared, but will return −1 with **errno** set if **O_NDELAY** is set. File access by other system functions, such as *exec*(2), are not subject to the enforcement policy. Locks on directories, pipes, and special files are advisory only; no enforcement policy will be used.

A potential for deadlock occurs if a process controlling a locked resource is put to sleep by accessing the locked resource of another process. Thus, calls to *fcntl*(2), *lockf*(2), *read*(2), or *write*(2) scan for a deadlock prior to sleeping on a locked resource. Deadlock is not checked for the *wait*(2) and *pause*(2) system calls, so potential for deadlock is not eliminated. A *creat*(2) call or an *open*(2) call with the **O_CREATE** and **O_TRUNC** flags set on a regular file will return EAGAIN error if another process has locked part of the file and the file is currently in enforcement mode.

NETWORKING FEATURES

NFS

The advisory record-locking capabilities of *lockf*(2) are implemented throughout the network by the "network lock daemon"; see *lockd*(1M). If the file server crashes and is rebooted, the lock daemon attempts to recover all locks associated with the crashed server. If a lock cannot be reclaimed, the process that held the lock is issued a SIGLOST signal.

Only advisory record locking is implemented for NFS files.

RETURN VALUE

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

ERRORS

Lockf fails if any of the following occur:

[EACCES] *Function* is **F_TLOCK** or **F_TEST** and the region is already locked by another process.

[EBADF] *Fildes* is not a valid, open file descriptor.

[EDEADLK] A deadlock would occur or the number of entries in the system lock table would exceed a system-dependent maximum. HP−UX guarantees this value to be at least 50.

[EINTR] A signal was caught during the *lockf* system call.

[EINVAL] *Function* is not one of the functions specified above.

[EINVAL] *Size* plus current offset produces a negative offset into the file.

[EINVAL] The resulting upper bound of the region to be locked would be greater than 2^30, or the current offset is greater than 2^30.

[ENOLCK] *Function* is **F_TLOCK** or **F_LOCK** and the file is a NFS file with access bits set for enforcement mode.

[ENOLCK] The file is a NFS file and a system error occurred on the remote node.

WARNINGS

Deadlock conditions may arise when either the *wait*(2) or *pause*(2) system calls are used in conjunction with enforced locking; see those pages for details.

File and record locking using file descriptors obtained through *dup*(2) or *link*(2) may not work as expected. For example, unlocking regions that were locked using either file descriptor may also unlock regions that were locked using the other file descriptor.

Unexpected results may occur in processes that use buffers in the user address space. The process may later read/write data which is or was locked. The standard I/O package, *stdio*(3S), is the most common source of unexpected buffering.

In a hostile environment, locking may be misused by holding key public resources locked. This is particularly true with public read files that have enforcement enabled.

It is not recommended that the PRIV_LOCKRDONLY capability be used, as it is provided only for backward compatibility. This feature may be modified or dropped from the future releases of HP-UX.

APPLICATION USAGE

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. For example:

```
if (lockf(fd, F_TLOCK, siz) == -1)
    if ((errno == EAGAIN) || (errno == EACCES))
        /*
         * section locked by another process
         * check for either EAGAIN or EACCES
         * due to different implementations
         */
    else if ...
        /*
         * check for other errors
         */
```

SEE ALSO

chmod(2), close(2), creat(2), fcntl(2), open(2), pause(2), read(2), stat(2), wait(2), write(2), unistd(5).
lockd(1M), statd(1M), in *NFS Services Reference Pages.*

FUTURE DIRECTIONS

The error condition that currently sets **errno** to EACCES will instead set **errno** to EAGAIN. (See also APPLICATION USAGE above.)

STANDARDS CONFORMANCE

lockf: SVID2, XPG2

NAME

lseek – move read/write file pointer; seek

SYNOPSIS

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

off_t lseek (fildes, offset, whence)
int fildes;
off_t offset;
int whence;
```

DESCRIPTION

Lseek sets the file pointer associated with the file descriptor as follows:

> If *whence* is **SEEK_SET**, the pointer is set to *offset* bytes.
> If *whence* is **SEEK_CUR**, the pointer is set to its current location plus *offset*.
> If *whence* is **SEEK_END**, the pointer is set to the size of the file plus *offset*.

These symbolic constants are defined in **<unistd.h>**.

RETURN VALUE

When *lseek* completes successfully, it returns a non-negative integer, which is the resulting file offset as measured in bytes from the beginning of the file. Otherwise, a value of −**1** is returned and **errno** is set to indicate the error.

ERRORS

Lseek fails and the file offset remains unchanged if one or more of the following is true:

[EBADF] *Fildes* is not an open file descriptor.

[ESPIPE] *Fildes* is associated with a pipe or FIFO.

[EINVAL and SIGSYS signal]
Whence is not one of the supported values.

[EINVAL] The resulting file offset would be negative.

WARNINGS

Some devices are incapable of seeking. The value of the file offset associated with such a device is undefined.

Using *lseek* with a *whence* of **SEEK_END** on device special files is not supported and the results are not defined.

SEE ALSO

creat(2), dup(2), fcntl(2), open(2), unistd(5).

STANDARDS CONFORMANCE

lseek: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

mkdir – make a directory file

SYNOPSIS

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

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

DESCRIPTION

Mkdir creates a new directory file named by *path*. The file permission bits of the new directory are initialized from *mode*, and are modified by the process's file mode creation mask. For each bit set in the process's file mode creation mask, the corresponding bit in the new directory's mode is cleared (see *umask*(2)).

The directory's owner ID is set to the process's effective-user-ID. If the set-group-ID bit of the parent directory is set, the directory's group ID is set to group ID of the parent directory. Otherwise, the directory's group ID is set to the process's effective-group-ID. The set-group-ID bit of the new directory is set to the same value as the set-group-ID bit of the parent directory.

Symbolic constants defining the access permission bits are found in the **<sys/stat.h>** header and are used to construct the argument *mode*. The value of the argument *mode* is the bitwise inclusive OR of the values of the desired permissions.

S_IRUSR	Read by owner.
S_IWUSR	Write by owner.
S_IXUSR	Execute (search) by owner.
S_IRGRP	Read by group.
S_IWGRP	Write by group.
S_IXGRP	Execute (search) by group.
S_IROTH	Read by others (that is, anybody else).
S_IWOTH	Write by others.
S_IXOTH	Execute (search) by others.

Access Control Lists (ACLs)

On systems implementing access control lists, the directory is created with three base ACL entries, corresponding to the file access permission bits (see *acl*(5)).

RETURN VALUE

Upon successful completion, *mkdir* returns a value of **0**; a return value of **–1** indicates an error, and an error code is stored in **errno**.

ERRORS

Mkdir fails and no directory is created if any of the following is true:

[EACCES]	A component of the path prefix denies search permission.
[EACCES]	The parent directory of the new directory denies write permission.
[EEXIST]	The named file already exists.
[EFAULT]	*Path* points outside the process's allocated address space. The reliable detection of this error is implementation dependent.
[EIO]	An I/O error occurred while writing to the file system.
[ELOOP]	Too many symbolic links are encountered in translating the path name.
[EMLINK]	The maximum number of links to the parent directory, {LINK_MAX}, would be exceeded.

[ENAMETOOLONG]
The length of the specified path name exceeds PATH_MAX bytes, or the length of a component of the path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.

[ENOENT] A component of the path prefix does not exist.

[ENOSPC] Not enough space on the file system.

[ENOTDIR] A component of the path prefix is not a directory.

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

AUTHOR

Mkdir was developed by the University of California, Berkeley.

SEE ALSO

chmod(2), setacl(2), stat(2), umask(2), acl(5).

STANDARDS CONFORMANCE

mkdir: SVID2, XPG3, POSIX.1, FIPS 151-1

NAME

mknod – make a directory, or a special or regular file

SYNOPSIS

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

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

int mkrnod(path, mode, dev, cnodeid)
char *path;
int mode;
dev_t dev;
cnode_t cnodeid;
```

DESCRIPTION

Mknod creates a new file named by the path name pointed to by *path*. The mode of the new file is specified by the *mode* argument. *Mkrnod* is the same as *mknod* but is used to make device files that can be accessed from a different cnode identified by the additional parameter *cnodeid*. A *cnodeid* value of **0** creates a "generic" device file that can be accessed by any cnode.

Symbolic constants defining the file type and file access permission bits are found in the **<sys/stat.h>** header file and are used to construct the *mode* argument. The value of the *mode* argument should be the bitwise inclusive OR of the values of the desired file type, miscellaneous mode bits, and access permissions. If the **S_IFMT** portion of *mode* has a value of 0, *mknod* creates a regular file. The mode value 0044000 (**S_CDF | S_IFDIR**) is used with *mkrnod* to indicate a hidden directory (see *cdf*(4)).

S_IFMT	File type mask
S_IFNWK	Network special file
S_IFIFO	FIFO special file
S_IFCHR	Character special file
S_IFDIR	Directory node
S_IFBLK	Block special file
S_IFREG	Regular file
S_ISUID	Set user ID on execution
S_ISGID	Set group ID on execution
S_ENFMT	Record locking enforced
S_ISVTX	Save text image after execution
S_IRWXU	Permission mask for owner
S_IRUSR	Read by owner
S_IWUSR	Write by owner
S_IXUSR	Execute (search) by owner
S_IRWXG	Permission mask for group
S_IRGRP	Read by group
S_IWGRP	Write by group
S_IXGRP	Execute (search) by group
S_IRWXO	Permission mask for others
S_IROTH	Read by others
S_IWOTH	Write by others
S_IXOTH	Execute (search) by others

The owner ID of the file is set to the effective-user-ID of the process. If the set-group-ID bit of the parent directory is set, the directory's group ID is set to the group ID of the parent directory.

Otherwise, the directory's group ID is set to the effective-group-ID of the process.

The file access permission bits of *mode* are modified by the process's file mode creation mask: for each bit set in the process's file mode creation mask, the corresponding bit in the file's mode is cleared (see *umask*(2)).

On systems implementing access control lists (ACLs), the directory is created with three base ACL entries, corresponding to the file access permission bits (see *acl*(5)).

Dev is meaningful only if *mode* indicates a block or character special file, and is ignored otherwise. It is an implementation- and configuration-dependent specification of a character or block I/O device. *Dev* is created by using the *makedev* macro defined in **<sys/sysmacros.h>**. The *makedev* macro takes as arguments the major and minor device numbers, whose value and interpretation are implementation dependent. The result of *makedev* is an object of type **dev_t**.

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

WARNINGS

Proper discretion should be used when using *mkrnod* to create generic device files. A generic device file accessed from different cnodes applies to different physical devices. Thus the file's ownership and permissions may not be appropriate in the context of all the cnodes.

RETURN VALUE

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

ERRORS

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

[EACCES] *Path* is in a directory that denies write permission, *mode* is for a FIFO special file, and the caller is not a super-user.

[EACCES] A component of the path prefix denies search permission.

[EEXIST] The named file exists.

[EFAULT] *Path* points outside the process's allocated address space. The reliable detection of this error is implementation dependent.

[ELOOP] Too many symbolic links are encountered in translating the path name.

[ENAMETOOLONG]
The length of the specified path name exceeds PATH_MAX bytes, or the length of a component of the path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.

[ENOENT] *Path* is null.

[ENOENT] A component of the path prefix does not exist.

[ENOSPC] Not enough space on the file system.

[ENOTDIR] A component of the path prefix is not a directory.

[EPERM] The effective-user-ID of the process does not match that of the super-user, and the file type is not FIFO special.

[EROFS] The directory in which the file is to be created is located on a read-only file system.

AUTHOR

Mknod was developed by AT&T and HP.

SEE ALSO

mkdir(2), mkdir(1), mknod(1M), chmod(2), exec(2), setacl(2), umask(2), cdf(4), fs(4), mknod(4), acl(5).

STANDARDS CONFORMANCE

mknod: SVID2, XPG2

NAME

mount – mount a file system

SYNOPSIS

int mount (spec, dir, rwflag)
char *spec, *dir;
int rwflag;

DESCRIPTION

Mount requests that a removable file system contained on the block special device file identified by *spec* be mounted on the directory identified by *dir*. *Spec* and *dir* are pointers to path names.

Upon successful completion, references to the file *dir* will refer to the root directory on the mounted file system.

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

Mount may be invoked only by the super-user.

RETURN VALUE

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

ERRORS

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

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

[ENOENT] The named file does not exist (for example, *path* is null or a component of *path* does not exist).

[ENOTDIR] A component of a path prefix is not a directory.

[ENOTBLK] *Spec* is not a block special device.

[ENXIO] The device associated with *spec* does not exist.

[ENOTDIR] *Dir* is not a directory.

[EFAULT] *Spec* or *dir* points outside the allocated address space of the process. The reliable detection of this error will be implementation dependent.

[EBUSY] *Dir* is currently mounted on, is someone's current working directory, or is otherwise busy.

[EBUSY] The device associated with *spec* is currently mounted.

[EBUSY] There are no more mount table entries.

[ENOENT] *Spec* or *dir* is null.

[EACCES] A component of the path prefix denies search permission.

[ENAMETOOLONG] The length of a specified path name exceeds PATH_MAX bytes, or the length of a component of the path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.

[ELOOP] Too many symbolic links were encountered in translating either path name.

WARNINGS

If *mount* is called from the program level (i.e. not called from *mount*(1M)), the table of mounted devices contained in **/etc/mnttab** is not updated.

DEPENDENCIES

HP Clustered Environment

When *mount* is called from a diskless node (cluster client), *spec* is interpreted as a device attached to the cluster server. This behavior is subject to change in future releases, and use in applications is not recommended.

SEE ALSO

mount(1M), umount(2).

STANDARDS CONFORMANCE

mount: SVID2, XPG2

NAME

msgctl – message control operations

SYNOPSIS

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

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

DESCRIPTION

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

IPC_STAT Place the current value of each member of the data structure associated with *msqid* into the structure pointed to by *buf*. The contents of this structure are defined in the *glossary*.

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

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

This *cmd* can only be executed by a process that has an effective user ID equal to either that of super user or to the value of either **msg_perm.uid** or **msg_perm.cuid** in the data structure associated with *msqid*. Only super user can raise the value of **msg_qbytes**.

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

ERRORS

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

[EINVAL] *Msqid* is not a valid message queue identifier.

[EINVAL] *Cmd* is not a valid command.

[EACCES] *Cmd* is equal to **IPC_STAT** and {READ} operation permission is denied to the calling process (see the *glossary).*

[EPERM] *Cmd* is equal to **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal to that of super-user and it is not equal to the value of either **msg_perm.uid** or **msg_perm.cuid** in the data structure associated with *msqid*.

[EPERM] *Cmd* is equal to **IPC_SET,** an attempt is being made to increase to the value of **msg_qbytes,** and the effective user ID of the calling process is not equal to that of super user.

[EFAULT] *Buf* points to an illegal address. The reliable detection of this error will be implementation dependent.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and

errno is set to indicate the error.

SEE ALSO

ipcrm(1), ipcs(1), msgget(2), msgop(2), stdipc(3C).

STANDARDS CONFORMANCE

msgctl: SVID2, XPG2, XPG3

NAME

msgget − get message queue

SYNOPSIS

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

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

DESCRIPTION

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

A message queue identifier and associated message queue and data structure are created for *key* if one of the following is true:

> *Key* is equal to **IPC_PRIVATE**. This call creates a new identifier, subject to available resources. The identifier will never be returned by another call to *msgget* until it has been released by a call to *msgctl*. The identifier should be used among the calling process and its descendents; however, it is not a requirement. The resource can be accessed by any process having the proper permissions.
>
> *Key* does not already have a message queue identifier associated with it, and (*msgflg* & **IPC_CREAT**) is "true".

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

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

ERRORS

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

[EACCES]	A message queue identifier exists for *key*, but operation permission as specified by the low-order 9 bits of *msgflg* would not be granted.
[ENOENT]	A message queue identifier does not exist for *key* and (*msgflg* & **IPC_CREAT**) is "false".
[ENOSPC]	A message queue identifier is to be created but the system-imposed limit on the maximum number of allowed message queue identifiers system wide would be exceeded.
[EEXIST]	A message queue identifier exists for *key* but ((*msgflg* & IPC_CREAT) **&&** (*msgflg* & **IPC_EXCL**)) is "true".

RETURN VALUE

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

SEE ALSO

ipcrm(1), ipcs(1), msgctl(2), msgop(2), stdipc(3C).

STANDARDS CONFORMANCE

msgget: SVID2, XPG2, XPG3

NAME

msgsnd, msgrcv – message operations

SYNOPSIS

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

int msgsnd (msqid, msgp, msgsz, msgflg)
int msqid;
void *msgp;
int msgsz, msgflg;

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

DESCRIPTION

Msgsnd is used to send a message to the queue associated with the message queue identifier specified by *msqid*.

Msgp points to a user-defined buffer that must contain first a field of type **long** that will specify the type of the message, followed by a data portion that will hold the data bytes of the message. The structure below is an example of what this user-defined buffer might look like:

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

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

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

The number of bytes already on the queue is equal to **msg_qbytes** (see *message queue identifier* in the Glossary).

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

These actions are as follows:

If (*msgflg* & **IPC_NOWAIT**) is "true", the message is not sent and the calling process returns immediately.

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

The condition responsible for the suspension no longer exists, in which case the message is sent.

Msqid is removed from the system (see *msgctl*(2)). When this occurs, **errno** is set equal to EIDRM, and a value of **−1** is returned.

The calling process receives a signal to be caught. In this case the message is not sent and the calling process resumes execution in the manner prescribed in *signal*(5).

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid*:

Msg_qnum is incremented by 1.

Msg_lspid is set equal to the process ID of the calling process.

Msg_stime is set equal to the current time.

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

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

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

Msgtyp specifies the type of message requested as follows:

If *msgtyp* is equal to 0, the first message on the queue is received.

If *msgtyp* is greater than 0, the first message of type *msgtyp* is received.

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

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

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

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

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

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

The calling process receives a signal that is to be caught. In this case a message is not received and the calling process resumes execution in the manner prescribed in *signal*(5)).

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid*.

Msg_qnum is decremented by 1.

Msg_lrpid is set equal to the process ID of the calling process.

Msg_rtime is set equal to the current time.

ERRORS

Msgsnd fails and no message is sent if one or more of the following is true:

[EINVAL]	*Msqid* is not a valid message queue identifier.
[EACCES]	Operation permission is denied to the calling process.
[EINVAL]	**Mtype** is less than 1.
[EAGAIN]	The message cannot be sent for one of the reasons cited above and (*msgflg* & **IPC_NOWAIT**) is "true".
[EINVAL]	*Msgsz* is less than zero or greater than the system-imposed limit.

[EFAULT] *Msgp* points to an illegal address. The reliable detection of this error is implementation dependent.

[EIDRM] The message queue identifier *msqid* has been removed from the system.

[EINTR] The function *msgsnd* was interrupted by a signal.

Msgrcv fails and no message is received if one or more of the following is true:

[EINVAL] *Msqid* is not a valid message queue identifier.

[EACCES] Operation permission is denied to the calling process.

[EINVAL] *Msgsz* is less than 0.

[E2BIG] **Mtext** is greater than *msgsz* and (*msgflg* & **MSG_NOERROR**) is "false".

[ENOMSG] The queue does not contain a message of the desired type and (*msgflg* & **IPC_NOWAIT**) is "true".

[EFAULT] *Msgp* points to an illegal address. The reliable detection of this error is implementation dependent.

[EIDRM] The message queue identifier *msqid* has been removed from the system.

[EINTR] The function *msgrcv* was interrupted by a signal.

RETURN VALUES

Upon successful completion, the return value is as follows:

Msgsnd returns a value of 0.

Msgrcv returns a value equal to the number of bytes actually placed into *mtext*.

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

WARNINGS

Check all references to *signal*(5) for appropriateness on systems that support *sigvector*(2). *Sigvector*(2) can affect the behavior described on this page.

SEE ALSO

ipcs(1), msgctl(2), msgget(2), signal(5), stdipc(3C).

STANDARDS CONFORMANCE

msgrcv: SVID2, XPG2, XPG3

msgsnd: SVID2, XPG2, XPG3

NAME
nice – change priority of a process

SYNOPSIS
int nice (incr)
int incr;

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

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

RETURN VALUE
Upon successful completion, *nice* returns the new nice value minus 20. Otherwise, a value of −1 is returned and **errno** is set to indicate the error.

Note that *nice* assumes a user process priority value of 20. If the super-user of your system has changed the user process priority value to something less than 20, certain increments can cause *nice* to return −1, which is indistinguishable from an error return.

ERRORS
[EPERM] *Nice* will fail and not change the nice value if *incr* is negative or greater than 40 and the effective user ID of the calling process is not super-user.

SEE ALSO
nice(1), exec(2).

STANDARDS CONFORMANCE
nice: SVID2, XPG2, XPG3

NAME

open – open file for reading or writing

SYNOPSIS

```
#include &lt;sys/types.h&gt;
#include &lt;sys/stat.h&gt;
#include &lt;fcntl.h&gt;

int open (path, oflag [ , mode ] )
char *path;
int oflag;
mode_t mode;
```

DESCRIPTION

Path points to a path name naming a file; it must not exceed PATH_MAX bytes in length. *Open* opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*. *Oflag* values are constructed by OR-ing flags from the list below.

Exactly one of the flags **O_RDONLY**, **O_WRONLY**, or **O_RDWR** must be used in composing the value of *oflag*. If none or more than one is used, the behavior is undefined. Several other flags listed below can be changed by using *fcntl* while the file is open. See *fcntl*(2) and *fcntl*(5) for details.

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NDELAY This flag might affect subsequent reads and writes. See *read*(2) and *write*(2).

When opening a FIFO with **O_RDONLY** or **O_WRONLY** set:

If **O_NDELAY** is set:

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

If **O_NDELAY** is clear:

An *open* for reading-only does not return until a process opens the file for writing. An *open* for writing-only does not return until a process opens the file for reading.

When opening a file associated with a communication line:

If **O_NDELAY** is set:

The *open* returns without waiting for carrier.

If **O_NDELAY** is clear:

The *open* does not return until carrier is present.

O_NONBLOCK Same effect as **O_NDELAY** for *open*(2), but slightly different effect in *read*(2) and *write*(2). Only one of **O_NONBLOCK** and **O_NDELAY** may be specified.

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

O_CREAT If the file exists, this flag has no effect, except as noted under **O_EXCL** below. Otherwise, the owner ID of the file is set to the effective user ID of the process, the group ID of the file is set to the effective group ID of the process if the set-group-ID bit of the parent directory is not set, or to the group ID of the parent directory if the set-group-ID bit of the parent directory is set. The file access permission bits of the file mode are set to the value of *mode* modified as follows (see *creat*(2)):

For each bit set in the file mode creation mask of the process, the corresponding bit in the new file's mode is cleared (see *umask*(2)).

The "save text image after execution" bit of the mode is cleared. See *chmod*(2).

On systems with access control lists, three base ACL entries are created corresponding to the file access permissions (see *acl*(5)).

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

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

O_NOCTTY If set, and *path* identifies a terminal device, *open* does not cause the terminal to become the controlling terminal for the process.

O_SYNC If a file is opened with **O_SYNC** or if that flag is set with the F_SETFL option of *fcntl*, file system writes for the file are done through the cache to the disk as soon as possible, and the process blocks until this is completed. This flag is ignored by all I/O calls except *write*, and is ignored for files other than ordinary files and block special devices on those systems that permit I/O to block special devices.

The name **O_SYNCIO** is a synonym for **O_SYNC**, and is defined for backward compatibility in <**fcntl.h**>.

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

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

EXAMPLES

The following call to *open* opens file *inputfile* for reading only and returns a file descriptor for *inputfile*. For an example of reading from file *inputfile*, see the *read*(2) manual page.

```
int myfd;

        myfd = open ("inputfile", O_RDONLY);
```

The following call to *open* opens file *outputfile* for writing and returns a file descriptor for *outputfile*. For an example of preallocating disk space for *outputfile*, see the *prealloc*(2) manual page. For an example of writing to *outputfile*, see the *write*(2) manual page.

```
int outfd;
        outfd = open ("outputfile", O_WRONLY);
```

RETURN VALUE

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

ERRORS

Open fails and the file is not opened if one of the following conditions is true. **Errno** is set accordingly:

[EACCES] *Oflag* permission is denied for the named file.

[EACCES] A component of the path prefix denies search permission.

[EACCES] The file does not exist and the directory in which the file is to be created does not permit writing.

[EAGAIN] One or more segments of a pre–existing file have been locked with *lockf* or *fcntl* by some other process, and **O_TRUNC** is set.

[EAGAIN] The file exists, enforcement mode file/record locking is set, and there are outstanding record locks on the file (see *chmod*(2)).

[EEXIST] **O_CREAT** and **O_EXCL** are set, and the named file exists.

[EFAULT] *Path* points outside the allocated address space of the process.

[EINTR] A signal was caught during the *open* system call, and the system call was not restarted (see *signal*(5) and *sigvector*(2)).

[EINVAL] *Oflag* specifies both **O_WRONLY** and **O_RDWR**.

[EINVAL] *Oflag* specifies both **O_NONBLOCK** and **O_NDELAY**.

[EISDIR] The named file is a directory and *oflag* is write or read/write.

[ELOOP] Too many symbolic links are encountered in translating the path name.

[EMFILE] The maximum number of file descriptors allowed are currently open.

[ENAMETOOLONG]
The length of the specified path name exceeds PATH_MAX bytes, or the length of a component of the path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.

[ENFILE] The system file table is full.

[ENOENT] The named file does not exist (for example, *path* is null or a component of *path* does not exist, or the file itself does not exist and **O_CREAT** is not set).

[ENOTDIR] A component of the path prefix is not a directory.

[ENXIO] **O_NDELAY** is set, the named file is a FIFO, **O_WRONLY** is set, and no process has the file open for reading.

[ENXIO] The named file is a character special or block special file, and the device associated with this special file does not exist.

[EROFS] The named file resides on a read-only file system and *oflag* is write or read/write.

[ETXTBSY] The file is open for execution and *oflag* is write or read/write. Normal executable files are only open for a short time when they start execution. Other executable file types may be kept open for a long time, or indefinitely under some circumstances.

DEPENDENCIES

HP Clustered Environment:

Attempting to open a device file with a *st_rcnode* value that does not match the cnode ID of the machine on which the calling process is running (or "0") will fail with an EOPNOTSUPP error.

AUTHOR

Open was developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

chmod(2), close(2), creat(2), dup(2), fcntl(2), lockf(2), lseek(2), read(2), select(2), setacl(2), umask(2), write(2), acl(5), fcntl(5), signal(5).

STANDARDS CONFORMANCE

open: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

pathconf, fpathconf – get configurable pathname variables

SYNOPSIS

#include <unistd.h>

long pathconf (*path*, *name*)
char **path*;
int *name*;

long fpathconf (*fildes*, *name*)
int *fildes*, *name*;

DESCRIPTION

The *pathconf* and *fpathconf* functions provide a method for applications to determine the value of a configurable limit or option associated with a file or directory (see *limits*(5) and **<unistd.h>**).

For *pathconf*, the *path* argument points to the path name of a file or directory.

For *fpathconf*, the *fildes* argument is an open file descriptor.

For both functions, the *name* argument represents the variable to be queried regarding the file or directory to which the other argument refers.

The following table lists the configuration variables available from *pathconf* and *fpathconf*, and lists for each variable the associated value of the *name* argument:

Variable	Value of *name*	Notes
LINK_MAX	**_PC_LINK_MAX**	1
MAX_CANON	**_PC_MAX_CANON**	2
MAX_INPUT	**_PC_MAX_INPUT**	2
NAME_MAX	**_PC_NAME_MAX**	3, 4
PATH_MAX	**_PC_PATH_MAX**	4, 5
PIPE_BUF	**_PC_PIPE_BUF**	6
_POSIX_CHOWN_RESTRICTED	**_PC_CHOWN_RESTRICTED**	7, 8
_POSIX_NO_TRUNC	**_PC_NO_TRUNC**	3, 4
_POSIX_VDISABLE	**_PC_V_DISABLE**	2

The variables in the table are defined as constants in **<limits.h>** or **<unistd.h>** if they do not vary from one pathname to another. The associated values of the *name* argument are defined in **<unistd.h>**.

RETURN VALUE

The following Notes further qualify the table above.

1. If *path* or *fildes* refers to a directory, the value returned applies to the directory itself.
2. If the variable is constant, the value returned is identical to the variable's definition in **<limits.h>** or **<unistd.h>** regardless of the type of *fildes* or *path*. The behavior is undefined if *path* or *fildes* does not refer to a terminal file.
3. If *path* or *fildes* refers to a directory, the value returned applies to the filenames within the directory.
4. If *path* or *fildes* does not refer to a directory, *pathconf* or *fpathconf* returns −1 and sets **errno** to EINVAL.
5. If *path* or *fildes* refers to a directory, the value returned is the maximum length of a relative path name when the specified directory is the working directory.
6. If *path* refers to a FIFO, or *fildes* refers to a pipe or FIFO, the value returned applies to the pipe or FIFO itself. If *path* or *fildes* refers to a directory, the value returned applies to any

FIFOs that exist or can be created within the directory. If PIPE_BUF is a constant, the value returned is identical to the definition of PIPE_BUF in **<limits.h>** regardless of the type of *fildes* or *path*. The behavior is undefined for a file other than a directory, FIFO, or pipe.

7. If *path* or *fildes* refers to a directory, the value returned applies to files of any type, other than directories, that exist or can be created within the directory.

8. _POSIX_CHOWN_RESTRICTED is defined if the privilege group PRIV_GLOBAL has been granted the CHOWN privilege. (See *getprivgrp*(2) and *chown*(2).) In all other cases, _POSIX_CHOWN_RESTRICTED is undefined and *pathconf* or *fpathconf* returns −1 without changing **errno**. To determine if *chown* can be performed on a file, it is simplest to attempt the *chown* operation and check the return value for failure or success.

If the variable corresponding to *name* is not defined for *path* or *fildes*, the *pathconf* and *fpathconf* functions succeed and return a value of −1, without changing the value of **errno**.

Upon any other successful completion, these functions return the value of the named variable with respect to the specified file or directory, as described above.

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

ERRORS

The *pathconf* and *fpathconf* functions fail if one of the following is true:

[EACCES] A component of the path prefix denies search permission.

[EBADF] The *fildes* argument is not a valid open file descriptor.

[EFAULT] *Path* points outside the allocated address space of the process.

[EINVAL] The value of *name* is not valid, or the implementation does not support an association of the variable *name* with the specified file.

[ELOOP] Too many symbolic links were encountered in translating *path*.

[ENAMETOOLONG] The length of the specified path name exceeds PATH_MAX bytes, or the length of a component of the path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.

[ENOENT] The file named by *path* does not exist (for example, *path* is null or a component of *path* does not exist).

[ENOTDIR] A component of the path prefix is not a directory.

EXAMPLES

The following example sets *val* to the value of MAX_CANON for the device file being used as the standard input. If the standard input is a terminal, this value is the maximum number of input characters that can be entered on a single input line before typing the newline character:

```
if (isatty(0))
    val = fpathconf(0, _PC_MAX_CANON);
```

The following code segment shows two calls to *pathconf*, one to determine whether a file name longer than NAME_MAX bytes will be truncated to NAME_MAX bytes in the **/tmp** directory, and if so, another call to determine the actual value of NAME_MAX so that an error can be printed if a user-supplied file name, stored in *filebuf*, will be truncated in this directory:

```
extern int errno;
char *filebuf;

errno = 0;  /* reset errno */
```

```
if ( pathconf("/tmp" _PC_NO_TRUNC) == -1 )
   /* _POSIX_NO_TRUNC is not in effect for this directory */
   if (strlen(filebuf) > pathconf("/tmp", PC_NAME_MAX)) {
      fprintf(stderr, "Filename %s too long.\n", filebuf);
      /* take error action */
   }
else
   if (errno) {
      perror("pathconf");
      /* take error action */
   }
/* otherwise, _POSIX_NO_TRUNC is in effect for this directory */
if ((fd = open(filebuf, O_CREAT, mode)) < 0)
   perror(filebuf);
```

DEPENDENCIES

NFS/RFA

ERRORS

[EOPNOTSUPP] *Path* or *fildes* refers to a file for which a value for *name* cannot be determined. In particular, _PC_LINK_MAX, _PC_NAME_MAX, _PC_PATH_MAX, _PC_CHOWN_RESTRICTED, and _PC_NO_TRUNC, cannot be determined for an NFS or RFA file.

AUTHOR

Pathconf and *fpathconf* were developed by HP.

SEE ALSO

errno(2), chown(2), limits(5), unistd(5), termio(7).

STANDARDS CONFORMANCE

pathconf: XPG3, POSIX.1, FIPS 151-1
fpathconf: XPG3, POSIX.1, FIPS 151-1

NAME

pause – suspend process until signal

SYNOPSIS

pause ()

DESCRIPTION

Pause suspends the calling process until it receives a signal. The signal must be one that is not currently set to be ignored or blocked (masked) by the calling process.

If the signal causes termination of the calling process, *pause* will not return.

If the signal is *caught* by the calling process and control is returned from the signal-catching function (see *signal*(5)), the calling process resumes execution from the point of suspension; with a return value of −1 from *pause* and **errno** set to EINTR.

WARNING

Check all references to *signal*(5) for appropriateness on systems that support *sigvector*(2). *Sigvector*(2) can affect the behavior described on this page.

SEE ALSO

alarm(2), kill(2), sigvector(2), wait(2), signal(5).

STANDARDS CONFORMANCE

pause: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

pipe – create an interprocess channel

SYNOPSIS

int pipe (fildes)
int fildes[2];

DESCRIPTION

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

A read-only file descriptor *fildes*[0] accesses the data written to *fildes*[1] on a first-in-first-out (FIFO) basis. For details of the I/O behavior of pipes see *read*(2) and *write*(2).

EXAMPLES

The following example uses *pipe* to implement the command string "ls | sort":

```
#include <sys/types.h>
pid_t pid;
int pipefd[2];

/* Assumes file descriptor 0 and 1 are open */
pipe (pipefd);

if ((pid = fork()) == (pid_t)0) {
        close(1);       /* close stdout */
        dup (pipefd[1]);
        execlp ("ls", "ls", (char *)0);
}
else if (pid > (pid_t)0) {
        close(0);       /* close stdin */
        dup (pipefd[0]);
        execlp ("sort", "sort", (char *)0);
}
```

RETURN VALUE

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

ERRORS

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

[EMFILE]	NFILE - 1 or more file descriptors are currently open.
[ENFILE]	The system file table is full.
[ENOSPC]	The file system lacks sufficient space to create the pipe.

SEE ALSO

sh(1), read(2), write(2), popen(3S).

STANDARDS CONFORMANCE

pipe: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

plock – lock process, text, or data in memory

SYNOPSIS

#include <sys/lock.h>

int plock (op)
int op;

DESCRIPTION

Plock allows the calling process to lock the text segment of the process (text lock), its data segment (data lock), or both its text and data segment (process lock) into memory. Locked segments are immune to all routine swapping. *Plock* also allows these segments to be unlocked. To use this call, the calling process must be a member of a privilege group allowing *plock* (see *setprivgrp* on *getprivgrp*(2)) or the effective user ID of the calling process must be super-user. *Op* specifies the following:

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

EXAMPLES

The following call to *plock* locks the calling process in memory:

plock (PROCLOCK);

RETURN VALUE

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

ERRORS

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

[EPERM]	The effective user ID of the calling process is not super-user and the user does not have PRIV_MLOCK.
[EINVAL]	*Op* is equal to PROCLOCK and a process lock, a text lock, or a data lock already exists on the calling process.
[EINVAL]	*Op* is equal to TXTLOCK and a text lock, or a process lock already exists on the calling process.
[EINVAL]	*Op* is equal to DATLOCK and a data lock, or a process lock already exists on the calling process.
[EINVAL]	*Op* is equal to UNLOCK and no type of lock exists on the calling process.
[EINVAL]	*Op* is not equal to either PROCLOCK, TXTLOCK, DATLOCK, or UNLOCK.
[EINVAL]	*Plock* not allowed in [vfork, exec] window (see *vfork*(2)).
[ENOMEM]	There is not sufficient lockable memory in the system to satisfy the locking request.

SEE ALSO

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

STANDARDS CONFORMANCE

plock: SVID2, XPG2

NAME

prealloc – preallocate fast disk storage

SYNOPSIS

int prealloc (fildes, size)
int fildes;
unsigned size;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat, pen, dup*, or *fcntl* system call for an ordinary file of zero length. It must be opened writable, since it will be written to by *prealloc*. *Size* is the size in bytes to be preallocated for the file specified by *fildes*. At least *size* bytes will be allocated. space is be allocated in an implementation-dependent fashion for fast sequential reads and writes. The EOF in an extended file will be left at the end of the preallocated area. The current file pointer is left at zero. The file is zero-filled.

Using *prealloc* on a file does **not** give the file an attribute that is inherited when copying or restoring the file using a program such as *cp*(1) or *tar*(1). It simply ensures that disk space has been preallocated for *size* bytes in a manner suited for sequential access. The file can be extended beyond these limits by *write* operations past the original end of file. However, this space will not necessarily be allocated using any special strategy.

EXAMPLES

Assuming a process has opened a file for writing, the following call to *prealloc* preallocates at least 50 000 bytes on disk for the file represented by file descriptor *outfd*:

prealloc (outfd, 50000);

DEPENDENCIES

Since the exact effect and performance benefits obtainable by using this call vary with the implementation of the file system, performance related details are described in the system administrator manuals for each specific machine.

RETURN VALUE

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

ERRORS

Prealloc will fail and no disk space will be allocated if one or more of the following are true:

[EBADF] *Fildes* is not a valid open file descriptor opened for writing.

[ENOTEMPTY] *Fildes* not associated with an ordinary file of zero length.

[ENOSPC] Not enough space left on device to allocate the requested amount; no space was allocated.

[EFBIG] *Size* exceeds the maximum file size or the process's file size limit. See *ulimit*(2). Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and **errno** is set to indicate the error.

AUTHOR

Prealloc was developed by the Hewlett-Packard Company.

SEE ALSO

prealloc(1), creat(2), dup(2), fcntl(2), open(2), read(2), ulimit(2), write(2).

WARNINGS

The allocation of the file space is highly dependent on current disk usage. A successful return does not tell you how fragmented the file actually might be if the disk is nearing its capacity.

NAME

profil – execution time profile

SYNOPSIS

#include <sys/param.h>

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

DESCRIPTION

Buff points to an area of core whose length (in bytes) is given by *bufsiz*. After this call, the user's program counter (pc) is examined each clock tick, *offset* is subtracted from it, and the result is multiplied by *scale*. If the resulting number corresponds to a word inside *buff*, that word is incremented. The number of samples per second for a given implementation is given by HZ as found in **<sys/param.h>**

The scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: 0177777 (octal) gives a 1-1 mapping of pc's to words in *buff*; 077777 (octal) maps each pair of instruction words together. 02(octal) maps all instructions onto the beginning of *buff* (producing a non-interrupting core clock).

Profiling is turned off by giving a *scale* of 0 or 1. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is turned off when an *exec* is executed, but remains on in child and parent both after a *fork*. Profiling will be turned off if an update in *buff* would cause a memory fault.

RETURN VALUE

Not defined.

SEE ALSO

prof(1), monitor(3C).

STANDARDS CONFORMANCE

profil: SVID2, XPG2

NAME

ptrace – process trace

SYNOPSIS

#include <sys/ptrace.h>

int ptrace(request, pid, addr, data, addr2);
int request, pid, addr, data, addr2;

REMARKS

Much of the functionality of this capability is highly dependent on the underlying hardware. An application that uses this system call should not be expected to be portable across architectures or implementations.

DESCRIPTION

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

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

PT_SETTRC This request must be issued by a child process if it is to be traced by its parent. It turns on the child's trace flag that stipulates that the child should be left in a stopped state upon receipt of a signal rather than the state specified by *func*; see *signal*(2). The *pid*, *addr*, *data*, and *addr2* arguments are ignored, and a return value is not defined for this request. Peculiar results ensue if the parent does not expect to trace the child.

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

PT_RIUSER, PT_RDUSER
With these requests, the word at location *addr* in the address space of the traced process is returned to the tracing process. If instruction (I) and data (D) space are separated, request **PT_RIUSER** returns a word from I space, and request **PT_RDUSER** returns a word from D space. If I and D space are not separated, either request **PT_RIUSER** or request **PT_RDUSER** may be used with equal results. The *data* and *addr2* arguments are ignored. These two requests fail if *addr* is not the start address of a word, in which case a value of **–1** is returned to the tracing process and its **errno** is set to EIO.

PT_RUAREA With this request, the word at location *addr* in the USER area of the traced process in the system's address space (see <**sys/user.h**>) is returned to the tracing process. Addresses in this area are system dependent, but start at zero. The limit can be derived from <**sys/user.h**>. The *data* and *addr2* arguments are ignored. This request fails if *addr* is not the start address of a word or is outside the area, in which case a value of **–1** is returned to the tracing process and its **errno** is set to EIO.

PT_WIUSER, PT_WDUSER
With these requests, the value given by the *data* argument is written into the address space of the traced process at location *addr*. Request **PT_WIUSER**

writes a word into I space, and request **PT_WDUSER** writes a word in D space. Upon successful completion, the value written into the address space of the traced process is returned to the tracing process. The *addr2* argument is ignored. These two requests fail if *addr* is not the start address of a word, or if *addr* is a location in a pure procedure space and either another process is executing in that space or the tracing process does not have write access for the executable file corresponding to that space. Upon failure a value of **−1** is returned to the tracing process and its **errno** is set to EIO.

PT_WUAREA With this request, a few entries in the traced process' USER area can be written. *Data* gives the value that is to be written and *addr* is the location of the entry. The *addr2* argument is ignored. The few entries that can be written are dependent on the architecture of the system, but include the user data registers, auxiliary data registers, and status register (the set of registers, or bits in registers, which the user's program could modify).

PT_CONTIN This request causes the traced process to resume execution. If the *data* argument is 0, all pending signals including the one that caused the traced process to stop are canceled before it resumes execution. If the *data* argument is a valid signal number, the traced process resumes execution as if it had incurred that signal, and any other pending signals are canceled. The *addr* argument must be equal to 1 for this request. The *addr2* argument is ignored. Upon successful completion, the value of *data* is returned to the tracing process. This request fails if *data* is not 0 or a valid signal number, in which case a value of **−1** is returned to the tracing process and its **errno** is set to EIO.

PT_EXIT This request causes the traced process to terminate with the same consequences as *exit*(2). The *addr*, *data*, and *addr2* arguments are ignored.

PT_SINGLE This request causes a flag to be set so that an interrupt occurs upon the completion of one machine instruction, and then executes the same steps as listed above for request **PT_CONTIN**. If the processor does not provide a trace bit, this request returns an error. This effectively allows single stepping of the traced process.

Whether or not the trace bit remains set after this interrupt is a function of the hardware.

PT_ATTACH This request stops the process identified by *pid* and allows the calling process to trace it. Process *pid* does not have to be a child of the calling process, but the effective user ID of the calling process must match the real and saved *uid* of process *pid* (unless the effective user ID of the tracing process is superuser). The calling process can use the *wait*(2) system call to wait for process *pid* to stop. The *addr*, *data*, and *addr2* arguments are ignored.

PT_DETACH This request detaches the traced process *pid* and allows it to continue its execution in the manner of **PT_CONTIN.**

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

ERRORS

In general, *ptrace* fails if one or more of the following is true:

[EIO] *Request* is an illegal number.

[EPERM] The specified process cannot be attached for tracing.

[ESRCH] *Pid* identifies a process to be traced that does not exist or has not executed a *ptrace* with request **PT_SETTRC**.

DEPENDENCIES

Series 300

The following additional requests are available:

PT_RFPREGS With this request, the child's floating point accelerator register set is returned to the parent process in *addr*. *Addr* must be the address of a buffer of at least 136 bytes. The first 128 bytes contains the 16 double precision floating point registers and the next 8 bytes contains the status and control registers. The data argument is ignored. This request fails if the child process is not using the floating point accelerator, in which case a value of **−1** is returned to the parent process and the parent's **errno** is set to EIO. This request also fails if *addr* is a bad address, in which case a value of **−1** is returned to the parent process and the parent's **errno** is set to EFAULT.

PT_WFPREGS With this request, the child's floating point accelerator register set is written from the buffer pointed to by *addr*. *Addr* must be the address of a buffer of at least 136 bytes. The first 128 bytes contains the new values for the 16 double precision floating point registers and the next 8 bytes contains the new values for the status and control registers. The data argument is ignored. This request fails if the child process is not using the floating point accelerator, in which case a value of **−1** is returned to the parent process and the parent's **errno** is set to EIO. This request also fails if *addr* is a bad address, in which case a value of **−1** is returned to the parent process and the parent's **errno** is set to EFAULT.

Series 800

The request **PT_WUAREA** is not supported. Therefore, it returns **−1**, sets **errno** to EIO and does not affect the USER area of the traced process.

If the *addr* argument to a **PT_CONTIN** or **PT_SINGLE** request is not 1, the Instruction Address Offset Queue (program counter) is loaded with the values *addr* and *addr*+4 before execution resumes. Otherwise, execution resumes from the point where it was interrupted.

If the *addr* argument to a **PT_DETACH** request is not 1, the Instruction Address Offset Queue is loaded with the values *addr* and *addr2*.

Additional requests are available:

PT_RUREGS With this request, the word at location *addr* in the **save_state** structure at the base of the per-process kernel stack is returned to the tracing process. *Addr* must be word-aligned and less than STACKSIZE*NBPG (see <**sys/param.h**> and <**machine/param.h**>). The **save_state** structure contains the registers and other information about the process. The *data* and *addr2* arguments are ignored.

PT_WUREGS The **save_state** structure at the base of the per-process kernel stack is written, as it is read with request **PT_RUREGS**. Only a few locations can be written in this way: the general registers, most floating point registers, a few control registers, and certain bits of the interruption processor status word. The *addr2* argument is ignored.

PT_RDTEXT, PT_RDDATA
These requests are identical to **PT_RIUSER** and **PT_RDUSER** except that the *data* argument specifies the number of bytes to read and the *addr2*

argument specifies where to store that data in the tracing process.

PT_WRTEXT, PT_WRDATA
These requests are identical to **PT_WIUSER** and **PT_WDUSER** except that the *data* argument specifies the number of bytes to write and the *addr2* argument specifies where to read that data in the tracing process.

SEE ALSO

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

STANDARDS CONFORMANCE

ptrace: SVID2, XPG2

NAME

read, readv – read input

SYNOPSIS

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

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

int readv (fildes, iov, iovcnt)
int fildes;
struct iovec *iov;
int iovcnt;
```

DESCRIPTION

Read attempts to read *nbyte* bytes from the file associated with the file descriptor into the buffer pointed to by *buf*. *Readv* performs the same action but scatters the input data into the *iovcnt* buffers specified by the elements of the *iovec* array: iov[0], iov[1], ..., iov[*iovcnt* - 1].

For *readv*, the **iovec** structure is defined as:

```
struct iovec {
     caddr_t iov_base;
     int iov_len;
};
```

Each **iovec** entry specifies the base address and length of an area in memory where data should be placed. *Readv* always fills one area completely before proceeding to the next area. The **iovec** array can be at most MAXIOV long.

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

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

When attempting to read from a regular file with enforcement-mode file and record locking set (see *chmod*(2)), and the segment of the file to be read is blocked by a write lock owned by another process, the behavior is determined by the O_NDELAY and O_NONBLOCK file status flags:

> If O_NDELAY or O_NONBLOCK is set, the *read* function returns **–1** and **errno** is set to EAGAIN.
>
> If O_NDELAY and O_NONBLOCK are clear, the *read* function does not return until the blocking write lock is removed.

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

> If O_NONBLOCK is set, the read returns **–1** and **errno** is set to EAGAIN.
>
> If O_NDELAY is set, the read returns a **0**.
>
> If O_NDELAY and O_NONBLOCK are clear, the read blocks until data is written to the file or the file is no longer open for writing.

When attempting to read a file associated with a tty that has no data currently available:

If O_NONBLOCK is set, the read returns −**1** and **errno** is set to EAGAIN.

If O_NDELAY is set, the read returns a **0**.

If O_NDELAY and O_NONBLOCK are clear, the read blocks until data becomes available.

RETURN VALUE

Upon successful completion, *read* returns the number of bytes actually read and placed in the buffer; this number may be less than *nbyte* if

- the file is associated with a communication line (see *ioctl*(2) and *termio*(7)), or
- the number of bytes left in the file is less than *nbyte* bytes.

When an end-of-file is reached, a value of **0** is returned Otherwise, a −**1** is returned and **errno** is set to indicate the error.

ERRORS

Read fails if one of the following conditions is true:

[EBADF] *Fildes* is not a valid file descriptor open for reading.

[EINTR] A signal was caught during the *read* system call.

[EAGAIN] Enforcement-mode file and record locking is set, O_NDELAY or O_NONBLOCK is set, and there is a blocking write lock.

[EDEADLK] A resource deadlock would occur as a result of this operation (see *lockf*(2) and *fcntl*(2)).

[EFAULT] *Buf* points outside the allocated address space. The reliable detection of this error is implementation dependent.

[EIO] The process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the SIGTTIN signal or the process group of the process is orphaned.

[EISDIR] An attempt was made to read a directory on an NFS file system using the *read* system call.

[ENOLCK] The system record lock table is full, preventing the read from sleeping until the blocking write lock is removed.

In addition, *readv* can return one of the following errors:

[EFAULT]
Iov_base or *iov* points outside of the allocated address space. The reliable detection of this error is implementation dependent.

[EINVAL]
Iovcnt is less then or equal to **0**, or greater than *MAXIOV*.

[EINVAL]
The sum of *iov_len* values in the *iov* array exceeded UINT_MAX (see <**limits.h**>).

EXAMPLES

Assuming a process opened a file for reading, the following call to *read*(2) reads BUFSIZ bytes from the file into the buffer pointed to by *mybuf*:

```
#include <stdio.h>   /* include this for BUFSIZ definition */

char mybuf[BUFSIZ];
int nbytes, fildes;

nbytes = read (fildes, mybuf, BUFSIZ);
```

WARNINGS

Record locking might not be enforced by the system, depending on the setting of the file's mode bits (see *lockf*(2)).

The character-special devices, and raw disks in particular, apply constraints on how *read* can be used. See the specific Section (7) entries for details on particular devices.

Check all references to *signal*(5) for appropriateness on systems that support *sigvector*(2). *Sigvector*(2) can affect the behavior described on this page.

In general, avoid using *read* to get the contents of a directory; use the *readdir* library routine, see *directory*(3C).

DEPENDENCIES

NFS

When obtaining the contents of a directory on an NFS file system, the *readdir* library routine must be used; see *directory*(3C). *Read* returns with an error if used to read a directory using NFS.

AUTHOR

Read was developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

creat(2), dup(2), fcntl(2), ioctl(2), lockf(2), open(2), pipe(2), select(2), ustat(2), tty(7), directory(3C).

STANDARDS CONFORMANCE

read: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

readlink – read value of a symbolic link

SYNOPSIS

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

DESCRIPTION

Readlink obtains the path name pointed to by the symbolic link, *path*. This path name is placed in the buffer *buf*, which has size *bufsiz*. The path name is not null terminated when returned.

RETURN VALUE

If the call succeeds, it returns the count of characters placed in the buffer. If an error occurs, it returns a **−1** and places the error code in the global variable **errno**.

ERRORS

Readlink will fail and the file mode will be unchanged if:

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG]
A component of *path* exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect, or *path* exceeds PATH_MAX bytes.

[ENOENT] The named file does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[ELOOP] Too many symbolic links were encountered in translating the path name.

[EINVAL] The named file is not a symbolic link.

[EFAULT] *Buf* points outside the process' allocated address space. Reliable detection of this error is implemenation dependent.

AUTHOR

Readlink was developed by the University of California, Berkeley.

SEE ALSO

stat(2), lstat(2), symlink(2), symlink(4).

NAME

reboot – boot the system

SYNOPSIS

#include <sys/reboot.h>

int reboot (RB_AUTOBOOT [| RB_NOSYNC]);
int reboot (RB_HALT [| RB_NOSYNC]);
int reboot (howto, device_file [, filename [, server_linkaddress]]);
int howto;
char *device_file, *filename;
char *server_linkaddress;

DESCRIPTION

Reboot causes the system to reboot. *Howto* is a mask of reboot options (see **<sys/reboot.h>**), specified as follows:

RB_AUTOBOOT	A file system sync is performed (unless RB_NOSYNC is set) and the processor is rebooted from the default device and file.
RB_HALT	The processor is simply halted. A sync of the file system is performed unless the RB_NOSYNC flag is set. RB_HALT should be used with caution.
RB_NOSYNC	A sync of the file system is not performed.
RB_NEWDEVICE	The *device_file* argument is used as the file name of the device from which to reboot.
RB_NEWFILE	The *filename* argument is used as the name of the file being rebooted.
RB_NEWSERVER	The additional optional parameter, *server_linkaddress*, specifies the ETHERNET link address of a new boot server. The *server_linkaddress* is a 12-character hexadecimal number that has the same format as the machine ID field of **/etc/clusterconf**. The **0x** prefix is optional. This allows a standalone system or HP cluster server to reboot and join an HP cluster as a diskless client, or for an existing diskless client to join a different HP cluster.

Device_file specifies the "boot device," the device from which the reboot occurs. *Device_file* must be a block or character special file name and is used only if the RB_NEWDEVICE option is set.

If the RB_NEWFILE option is set, *filename* specifies the "boot file", the name of the file being rebooted. This file will be loaded into memory by the bootstrap and control passed to it.

If the RB_NEWSERVER option is set, *reboot*(2) does not verify that *server_linkaddress* is a valid ETHERNET address, nor that the specified server is valid or provides the required service.

If the boot device is not a LAN device, the *server_linkaddress* information is ignored. The boot device is considered a LAN device if the previous boot was from a LAN device or if a LAN device is specified via the RB_NEWDEVICE option.

Unless the RB_NOSYNC flag has been specified, *reboot*(2) unmounts all mounted file systems and marks them clean so that it will not be necessary to run *fsck*(1M) on these files systems when the system reboots.

Only the super-user can reboot a machine.

RETURN VALUE

If successful, this call never returns. Otherwise, a **−1** is returned and an error code is returned in the global variable **errno**.

ERRORS

Reboot fails if any of the following is true:

[EFAULT]	*Device_file* points outside the allocated address space of the process.
[ENAMETOOLONG]	the path name specified by *device_file* exceeds PATH_MAX bytes, or the length of a component of the path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.
[EINVAL]	*Device_file* is not a block or a character device.
[ENET]	The device specified by *device_file* is remote.
[ENOENT]	The file specified by *device_file* does not exist.
[ENOTDIR]	A component of the path prefix specified by *device_file* is not a directory.
[ENXIO]	The device named by *device_file* does not exist.
[EPERM]	The effective user ID of the caller is not super-user.

DEPENDENCIES

Series 300

Filename must be one of the files listed by the boot ROM at power-up.

The default device, file, and server for RB_AUTOBOOT are those from which the system was previously booted.

If the RB_NEWDEVICE option is used and *device_file* specifies a LAN device, the RB_NEWSERVER option and *server_linkaddress* parameter must also be used.

If an invalid *server_linkaddress* is specified with the RB_NEWSERVER option, or if the requested server does not respond, the Series 300 boot ROM displays the message BOOTING A SYSTEM and retries infinitely, or until the requested server responds or the system is rebooted manually.

Series 800

The RB_NEWDEVICE, RB_NEWFILE, and RB_NEWSERVER options and the *device_file*, *filename* and *server_linkaddress* parameters are ignored, and, therefore, none of the errors associated with them are returned.

The default file and device for RB_AUTOBOOT are **/hp-ux** on the current root device.

AUTHOR

Reboot was developed by HP and the University of California, Berkeley.

SEE ALSO

reboot(1M), clusterconf(4).

NAME

rename – change the name of a file

SYNOPSIS

#include <stdio.h>

rename(source, target)
const char *source, *target;

DESCRIPTION

Rename causes the file named *source* to be renamed *target*. If *target* exists, it is first removed. Both *source* and *target* must be of the same type (that is, either directories or non-directories), and must reside on the same file system.

If *target* can be created or if it existed before the call, *rename* guarantees that an instance of *target* will exist, even if the system crashes in the midst of the operation.

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

RETURN VALUE

If the operation succeeds, **0** is returned. If not, *rename* returns **–1** and the global variable **errno** indicates the reason for the failure.

ERRORS

Rename will fail and neither file will be affected if any of the following is true:

[EACCES]	A component of either path prefix denies search permission.
[EACCES]	The requested link requires writing to a directory without write permission.
[EBUSY]	*Target* is an existing directory that is the mount point for a mounted file system.
[EFAULT]	*Source* or *target* points outside the allocated address space of the process. The reliable detection of this error is implementation dependent.
[EINVAL]	*Source* is a parent directory of *target*, or an attempt is made to rename "." or "..".
[EISDIR]	*Target* is a directory, but *source* is not.
[ELOOP]	Too many symbolic links were encountered in translating either path name.
[ENAMETOOLONG]	A component of either path name exceeds **NAME_MAX** bytes while _POSIX_NO_TRUNC is in effect, or the entire length of either path name exceeds **PATH_MAX** bytes.
[ENOENT]	A component of the *source* path does not exist, or a path prefix of *target* does not exist.
[ENOSPC]	The destination directory cannot be extended, because of a lack of space on the file system containing the directory.
[ENOTDIR]	A component of either path prefix is not a directory.
[ENOTDIR]	*Source* is a directory, but *target* is not.
[ENOTEMPTY]	*Target* is a directory and is not empty.
[EROFS]	The requested link requires writing in a directory on a read-only file system.

[EXDEV] The paths named by *source* and *target* are on different logical devices (file systems).

AUTHOR

Rename was developed by the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

open(2).

STANDARDS CONFORMANCE

rename: XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

rmdir – remove a directory file

SYNOPSIS

rmdir(path)
char *path;

DESCRIPTION

Rmdir removes a directory file whose name is given by *path*. The directory must be empty (except for files "." and "..") before it can be removed.

RETURN VALUE

A **0** is returned if the directory removal succeeds; otherwise, a **−1** is returned and an error code is stored in the global location **errno**.

ERRORS

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

[EACCES] A component of the path prefix denies search permission.

[EACCES] Write permission is denied on the directory containing the link to be removed.

[EBUSY] The directory to be removed is the mount point for a mounted file system.

[EEXIST] The named directory is not empty. It contains files other than "." and "..".

[EFAULT] *Path* points outside the process's allocated address space. The reliable detection of this error is implementation dependent.

[ENAMETOOLONG]
The length of the specified path name exceeds PATH_MAX bytes, or the length of a component of the path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.

[ENOENT] The named file does not exist.

[ENOTDIR] A component of the path is not a directory.

[EROFS] The directory entry to be removed resides on a read-only file system.

[EINVAL] The path is ".".

[ELOOP] Too many symbolic links were encountered in translating the path name.

AUTHOR

Rmdir was developed by the University of California, Berkeley.

SEE ALSO

mkdir(2), unlink(2).

STANDARDS CONFORMANCE

rmdir: SVID2, XPG3, POSIX.1, FIPS 151-1

NAME

rtprio – change or read realtime priority

SYNOPSIS

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

rtprio (pid, prio)
int pid, prio;
```

DESCRIPTION

Rtprio is used to set or read the realtime priority of a process. If *pid* is zero, it names the calling process; otherwise it gives the pid of the process. When setting the realtime priority of another process, the real or effective user ID of the calling process must match the real or saved user ID of the process to be modified, or the effective user ID of the calling process must be that of super-user. The calling process must also be a member of a privilege group allowing *rtprio* (see *getprivgrp*(2)) or the effective user ID of the calling process must be super-user. Simply reading realtime priorities requires no special privilege.

Real time scheduling policies differ from the normal timesharing policies in that the realtime priority is used to absolutely order all realtime processes; this priority is not degraded over time. All realtime processes are of higher priority than normal user and system processes, although some system processes may run at realtime priorities. If there are several eligible processes at the same priority level, they will be run in a round robin fashion as long as no process with higher priority intervenes. A realtime process will receive cpu service until it either voluntarily gives up the cpu or is preempted by a process of equal or higher priority. Interrupts may also preempt a realtime process.

Valid realtime priorities run from zero to 127. Zero is the highest (most important) priority. This realtime priority is inherited across *fork*s and *exec*s.

Prio specifies the following:

0–127 Set process to this realtime priority.

RTPRIO_NOCHG
: Do not change realtime priority. This is used for reading the process realtime priority.

RTPRIO_RTOFF Set this process to no longer have a realtime priority. It will resume a normal timesharing priority. Any process, regardless of privilege, is allowed to turn off its own realtime priority using a *pid* of zero.

EXAMPLES

The following call to *rtprio* sets the calling process to a real-time priority of 90:

rtprio (0, 90);

RETURN VALUE

If no error occurs, *rtprio* will return the *pid*'s former (before the call) realtime priority. If the process was not a realtime process, RTPRIO_RTOFF will be returned. If an error does occur, -1 is returned and **errno** is set to one of the values described in the ERRORS section.

ERRORS

[EINVAL] *Prio* is not RTPRIO_NOCHG, RTPRIO_RTOFF, or in the range of 0 to 127.

[EPERM] The calling process is not the super–user and neither its real or effective user–id match the real or saved user–id of the process indicated by *pid.*

[EPERM] The group access list of the calling process does not contain a group having PRIV_RTPRIO capability and *prio* is not RTPRIO_NOCHG, or RTPRIO_RTOFF with a *pid* of zero.

[ESRCH] No process can be found corresponding to that specified by *pid*.

DEPENDENCIES

Series 800:

Because processes executing at realtime priorities get scheduling preference over a system process executing at a lower priority, unexpected system behavior can occur after a power failure. For example, when *init*(1M) receives the powerfail signal SIGPWR, it normally reloads programmable hardware such as terminal multiplexers. If a higher-priority real-time process is eligible to run after the power failure, running of *init* is delayed. This condition temporarily prevents terminal input to any process, including realtime shells of higher priority than the eligible realtime process. To avoid this situation, a realtime process should catch SIGPWR and suspend itself until *init* has finished its powerfail processing.

AUTHOR

Rtprio was developed by HP.

SEE ALSO

rtprio(1), getprivgrp(2), nice(2), plock(2).

WARNINGS

Normally, compute bound programs should not be run at realtime priorities, because all time sharing work on the cpu would come to a complete halt.

NAME

select – synchronous I/O multiplexing

SYNOPSIS

#include <time.h>

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

DESCRIPTION

Select examines the file descriptors specified by the bit masks *readfds*, *writefds* and *exceptfds*. The bits from 0 through *nfds*-1 are examined. File descriptor *f* is represented by the bit 1<<*f* in the masks. More formally, a file descriptor is represented by:

```
fds[(f / BITS_PER_INT)] & (1 << (f % BITS_PER_INT))
```

When *select* completes successfully it returns the three bit masks modified as follows: For each file descriptor less than *nfds*, the corresponding bit in each mask is set if the bit was set upon entry and the file descriptor is ready for reading, writing or has an exceptional condition pending.

If *timeout* is a non-zero pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a zero pointer, the select waits until an event causes one of the masks to be returned with a valid (non-zero) value. To poll, the *timeout* argument should be non-zero, pointing to a zero valued timeval structure. Specific implementations may place limitations on the maximum timeout interval supported. The constant MAX_ALARM defined in **<sys/param.h>** specifies the implementation-specific maximum (in seconds). Whenever *timeout* specifies a value greater than this maximum, it is silently rounded down to this maximum. On all implementations, MAX_ALARM is guaranteed to be at least 31 days (in seconds). Note that the use of a timeout does not affect any pending timers set up by *alarm*(2) or *setitimer*(2).

Any or all of *readfds*, *writefds*, and *exceptfds* may be given as 0 if no descriptors are of interest. If all the masks are given as 0 and *timeout* is not a zero pointer, *select* blocks for the time specified, or until interrupted by a signal. If all the masks are given as 0 and *timeout* is a zero pointer, *select* blocks until interrupted by a signal.

Ordinary files always select true whenever selecting on reads, writes, and/or exceptions.

EXAMPLES

The following call to *select* checks if any of 4 terminals are ready for reading. *Select* will time out after 5 seconds if no terminals are ready for reading. Note that the code for opening the terminals or reading from the terminals is not shown in this example. Also, note that this example must be modified if the calling process has more than 32 file descriptors open. Following this first example is an example of select with more than 32 file descriptors.

```
#define MASK(f)        (1 << (f))
#define NTTYS 4

int tty[NTTYS];
int ttymask[NTTYS];
int readmask = 0;
int readfds;
int nfound, i;
struct timeval timeout;
```

```
        /* First open each terminal for reading and put the
         * file descriptors into array tty[NTTYS]. The code
         * for opening the terminals is not shown here.
         */

        for (i=0; i < NTTYS; i++) {
                ttymask[i] = MASK(tty[i]);
                readmask |= ttymask[i];
        }

        timeout.tv_sec  = 5;
        timeout.tv_usec = 0;
        readfds = readmask;

        /* select on NTTYS+3 file descriptors if stdin, stdout
         * and stderr are also open
         */
        if ((nfound = select (NTTYS+3, &readfds, 0, 0, &timeout)) == -1)
                perror ("select failed");
        else if (nfound == 0)
                printf ("select timed out \n");
        else for (i=0; i < NTTYS; i++)
                if (ttymask[i] & readfds)
                        /* Read from tty[i]. The code for reading
                         * is not shown here.
                         */
                else printf ("tty[%d] is not ready for reading \n",i);
```

The following example is the same as the previous example, except that it will work for more than 32 open files. Definitions for howmany, fd_set, and NFDBITS are in <sys/types.h>.

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

#define MASK(f)        (1 << (f))
#define NTTYS NOFILE - 3
#define NWORDS  howmany(FD_SETSIZE, NFDBITS)

int tty[NTTYS];
int ttymask[NTTYS];
struct fd_set readmask, readfds;
int nfound, i, j, k;
struct timeval timeout;

        /* First open each terminal for reading and put the
         * file descriptors into array tty[NTTYS]. The code
         * for opening the terminals is not shown here.
         */

        for (k=0; k < NWORDS; k++)
                readmask.fds_bits[k] = 0;
```

```
for (i=0, k=0; i < NTTYS && k < NWORDS; k++)
        for (j=0; j < NFDBITS && i < NTTYS; j++, i++) {
                ttymask[i] = MASK(tty[i]);
                readmask.fds_bits[k] |= ttymask[i];
        }

timeout.tv_sec  = 5;
timeout.tv_usec = 0;
for (k=0; k < NWORDS; k++)
        readfds.fds_bits[k] = readmask.fds_bits[k];

/* select on NTTYS+3 file descriptors if stdin, stdout
 * and stderr are also open
 */
if ((nfound = select (NTTYS+3, &readfds, 0, 0, &timeout)) == -1)
        perror ("select failed");
else if (nfound == 0)
        printf ("select timed out \n");
else for (i=0, k=0; i < NTTYS && k < NWORDS; k++)
        for (j=0; j < NFDBITS && i < NTTYS; j++, i++)
                if (ttymask[i] & readfds.fds_bits[k])
                        /* Read from tty[i]. The code for reading
                         * is not shown here.
                         */
                else printf ("tty[%d] is not ready for reading \n",i);
```

RETURN VALUE

Select returns the number of descriptors contained in the bit masks, or −1 if an error occurred. If the time limit expires then *select* returns 0 and all the masks are cleared.

ERRORS

An error return from *select* indicates:

[EBADF]	One or more of the bit masks specified an invalid descriptor.
[EINTR]	A signal was delivered before any of the selected for events occurred or before the time limit expired.
[EFAULT]	One or more of the pointers was invalid. The reliable detection of this error will be implementation dependent.
[EINVAL]	Invalid timeval passed for timeout.
[EINVAL]	The value of *nfds* is less than zero.

WARNINGS

Check all references to *signal*(5) for appropriateness on systems that support *sigvector*(2). *Sigvector*(2) can affect the behavior described on this page.

The file descriptor masks are always modified on return, even if the call returns as the result of a timeout.

DEPENDENCIES

Series 300

Select(2) supports the following devices and file types:

pipes
fifo special files (named pipes)
All serial interfaces
All ITEs and HP-HIL input devices

pty(7) special files
HP 98643 LAN interface card driver

File types not supporting *select*(2) always return true.

Series 800

Select(2) supports the following devices and file types:

pipes
fifo special files (named pipes)
all serial devices
All ITEs and HP-HIL input devices
hpib(7) special files
gpio(7) special files
lan(7) special files
pty(7) special files

The convention for device files that do not support select(2) is to always return true for those conditions the user is selecting on.

Consult the individual device manual pages to determine the extent to which any particular driver supports select(2).

HP Clustered Environment

In a clustered environment, *select* is not supported for distributed fifos, i.e., fifos that are open simultaneously on multiple machines. In this case an error of EINVAL is returned.

AUTHOR

Select was developed by HP and the University of California, Berkeley.

SEE ALSO

fcntl(2), read(2), write(2).

NAME

semctl – semaphore control operations

SYNOPSIS

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

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

DESCRIPTION

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

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

GETVAL Return the value of semval (see the *glossary).*

SETVAL Set the value of semval to *arg.val*. When this cmd is successfully executed, the semadj value corresponding to the specified semaphore in all processes is cleared.

GETPID Return the value of sempid.

GETNCNT Return the value of semncnt.

GETZCNT Return the value of semzcnt.

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

GETALL Place semvals into array pointed to by *arg.array*.

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

The following *cmd*s are also available:

IPC_STAT Place the current value of each member of the data structure associated with *semid* into the structure pointed to by *arg.buf*. The contents of this structure are defined in the *glossary.*

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

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

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

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

EXAMPLES

The following call to *semctl* initializes the set of 4 semaphores to the values 0, 1, 0 and 1 respectively. This example assumes the process has a valid semid representing a set of 4 semaphores as shown on the *semget*(2) manual page. For an example of performing "P" and "V" operations on the semaphores below, refer to the *semop*(2) manual page.

```
ushort semarray[4];

        semarray[0] = 0;
        semarray[1] = 1;
        semarray[2] = 0;
        semarray[3] = 1;

        semctl (mysemid, 0, SETALL, semarray);
```

ERRORS

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

[EINVAL]	*Semid* is not a valid semaphore identifier.
[EINVAL]	*Semnum* is less than zero or greater than or equal **sem_nsems**.
[EINVAL]	*Cmd* is not a valid command.
[EACCES]	Operation permission is denied to the calling process (see the *glossary).*
[ERANGE]	*Cmd* is **SETVAL** or **SETALL** and the value to which semval is to be set is greater than the system imposed maximum.
[EPERM]	*Cmd* is equal to **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal to that of super-user and it is not equal to the value of either **sem_perm.uid** or **sem_perm.cuid** in the data structure associated with *semid*.
[EFAULT]	*Arg.buf* or *arg.array* points to an illegal address. The reliable detection of this error will be implementation dependent.

RETURN VALUE

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

GETVAL	The value of semval.
GETNCNT	The value of semncnt.
GETZCNT	The value of semzcnt.
GETPID	The value of sempid.
All others	A value of 0.

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

SEE ALSO

ipcrm(1), ipcs(1), semget(2), semop(2), stdipc(3C).

STANDARDS CONFORMANCE

semctl: SVID2, XPG2, XPG3

NAME

semget – get set of semaphores

SYNOPSIS

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

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

DESCRIPTION

Semget returns the semaphore identifier associated with *key*.

A semaphore identifier and associated data structure and set containing *nsems* semaphores are created for *key* if one of the following is true:

Key is equal to **IPC_PRIVATE**. This call creates a new identifier, subject to available resources. The identifier will never be returned by another call to *semget* until it has been released by a call to *semctl*. The identifier should be used among the calling process and its descendents; however, it is not a requirement. The resource can be accessed by any process having the proper permissions.

Key does not already have a semaphore identifier associated with it, and (*semflg* & **IPC_CREAT**) is "true".

Specific behavior may be requested by or'ing the following masks into *semflg*.

IPC_CREAT: Create a semaphore identifier, if one does not already exist for *key*.

IPC_EXCL: If **IPC_CREAT** is specified and *key* already has a semaphore identifier associated with it, return an error.

The low-order 9 bits of *semflg* are the semaphore operation permissions which are defined in the *glossary*.

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

In the operation-permission structure, **sem_perm.cuid** and **sem_perm.uid** are set equal to the effective-user-ID of the calling process, while **sem_perm.cgid** and **sem_perm.gid** are set to the effective-group-ID of the calling process.

The low-order 9 bits of **sem_perm.mode** are set equal to the low-order 9 bits of *semflg*.

Sem_nsems is set equal to the value of *nsems*.

Sem_otime is set equal to 0 and **sem_ctime** is set equal to the current time.

EXAMPLES

The following call to *semget* returns a semid associated with the key returned by ftok("myfile", 'A'). If a semid associated with the key does not exist, a new semid, set of 4 semaphores and associated data structure will be created. If a semid for the key already exists, the semid is simply returned.

```
int semid;
        mysemid = semget (ftok("myfile",'A'), 4, IPC_CREAT | 0600);
```

ERRORS

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

[EINVAL] *Nsems* is either less than or equal to zero or greater than the system-imposed limit.

[EACCES] A semaphore identifier exists for *key*, but operation permission as specified by the low-order 9 bits of *semflg* would not be granted.

[EINVAL] A semaphore identifier exists for *key*, but the number of semaphores in the set associated with it is less than *nsems* and *nsems* is not equal to zero.

[ENOENT] A semaphore identifier does not exist for *key* and (*semflg* & **IPC_CREAT**) is "false".

[ENOSPC] A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphore identifiers system wide would be exceeded.

[ENOSPC] A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphores system wide would be exceeded.

[EEXIST] A semaphore identifier exists for *key* but ((*semflg* & IPC_CREAT) **&&** (*semflg* & **IPC_EXCL**)) is "true".

RETURN VALUE

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

SEE ALSO

ipcrm(1), ipcs(1), semctl(2), semop(2), stdipc(3C).

STANDARDS CONFORMANCE

semget: SVID2, XPG2, XPG3

NAME

semop – semaphore operations

SYNOPSIS

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

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

DESCRIPTION

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

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

Each semaphore operation specified by *sem_op* is performed on the corresponding semaphore specified by *semid* and *sem_num*. Semaphore array operations are atomic, in that none of the semaphore operations will be performed until blocking conditions on all of the semaphores in the array have been removed.

Sem_op specifies one of three semaphore operations as follows:

If *sem_op* is a negative integer, one of the following will occur:

If semval (see *semaphore identifier* in the Glossary) is greater than or equal to the absolute value of *sem_op*, the absolute value of *sem_op* is subtracted from semval. Also, if (*sem_flg* & **SEM_UNDO**) is "true", the absolute value of *sem_op* is added to the calling process's semadj value (see the Glossary and *exit*(2)) for the specified semaphore.

If semval is less than the absolute value of *sem_op* and (*sem_flg* & **IPC_NOWAIT**) is "true", *semop* will return immediately.

If semval is less than the absolute value of *sem_op* and (*sem_flg* & **IPC_NOWAIT**) is "false", *semop* will increment the semncnt associated with the specified semaphore and suspend execution of the calling process until one of the following conditions occur:

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

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

The calling process receives a signal that is to be caught. When this occurs, the value of semncnt associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in *signal*(5).

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

If *sem_op* is zero, one of the following will occur:

> If semval is zero, *semop* will proceed to the next semaphore operation specified by *sops*, or return immediately if this is the last operation.
>
> If semval is not equal to zero and (*sem_flg* & **IPC_NOWAIT**) is "true", *semop* will return immediately.
>
> If semval is not equal to zero and (*sem_flg* & **IPC_NOWAIT**) is "false", *semop* will increment the semzcnt associated with the specified semaphore and suspend execution of the calling process until one of the following occurs:
>
> > Semval becomes zero, at which time the value of semzcnt associated with the specified semaphore is decremented.
> >
> > The semid for which the calling process is awaiting action is removed from the system. When this occurs, **errno** is set equal to EIDRM, and a value of −1 is returned.
> >
> > The calling process receives a signal that is to be caught. When this occurs, the value of semzcnt associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in *signal*(5).

EXAMPLES

The following call to *semop* atomically performs a "P" or "get" operation on the second semaphore in the semaphore set and a "V" or "release" operation on the third semaphore in the set. This example assumes the process has a valid semid which represents a set of 4 semaphores as shown on the *semget*(2) manual page. It also assumes that the semvals of the semaphores in the set have been initialized as shown on the *semctl*(2) manual page.

```
struct sembuf sops[4];

sops[0].sem_num = 1;
sops[0].sem_op  = -1;/* P (get) */
sops[0].sem_flg = 0;
sops[1].sem_num = 2;
sops[1].sem_op  =  1;/* V (release) */
sops[1].sem_flg = 0;

semop (mysemid, sops, 2);
```

ERRORS

Semop will fail if one or more of the following are true for any of the semaphore operations specified by *sops*:

[EINVAL] *Semid* is not a valid semaphore identifier.

[EFBIG] *Sem_num* is less than zero or greater than or equal to the number of semaphores in the set associated with *semid*.

[E2BIG] *Nsops* is greater than the system-imposed maximum.

[EACCES] Operation permission is denied to the calling process (see the Glossary).

[EAGAIN] The operation would result in suspension of the calling process but (*sem_flg* & **IPC_NOWAIT**) is "true".

[ENOSPC] The limit on the number of individual processes requesting an **SEM_UNDO** would be exceeded.

[EINVAL] The number of individual semaphores for which the calling process requests a **SEM_UNDO** would exceed the limit.

[ERANGE] An operation would cause a semval to overflow the system-imposed limit.

[ERANGE] An operation would cause a semadj value to overflow the system-imposed limit.

[EFAULT] *Sops* points to an illegal address. The reliable detection of this error will be implementation dependent.

Upon successful completion, the value of sempid for each semaphore specified in the array pointed to by *sops* is set equal to the process ID of the calling process. The value of **sem_otime** in the data structure associated with the semaphore identifier will be set to the current time.

RETURN VALUE

If *semop* returns due to the receipt of a signal, a value of **−1** is returned to the calling process and **errno** is set to EINTR. If it returns due to the removal of a *semid* from the system, a value of **−1** is returned and **errno** is set to EIDRM.

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

WARNINGS

Check all references to *signal*(5) for appropriateness on systems that support *sigvector*(2). *Sigvector*(2) can affect the behavior described on this page.

SEE ALSO

ipcs(1), exec(2), exit(2), fork(2), semctl(2), semget(2), stdipc(3C), signal(5).

STANDARDS CONFORMANCE

semop: SVID2, XPG2, XPG3

NAME

setacl, fsetacl – set access control list (ACL) information

SYNOPSIS

#include <unistd.h>
#include <sys/acl.h>

```
int setacl (path, nentries, acl)
char    *path;
int     nentries;
struct  acl_entry acl[];

int fsetacl (fildes, nentries, acl)
int     fildes;
int     nentries;
struct  acl_entry acl[];
```

Remarks:

To ensure continued conformance with emerging industry standards, features described in this manual entry are likely to change in a future release.

DESCRIPTION

Setacl sets an existing file's access control list (ACL) or deletes optional entries from it. *Path* points to a path name of a file.

Similarly, *fsetacl* sets an existing file's access control list for an open file known by the file descriptor *fildes*.

The effective user ID of the process must match the owner of the file or be the superuser to set a file's ACL.

A successful call to *setacl* deletes all of a file's previous optional ACL entries (see explanation below), if any. The *nentries* parameter indicates how many valid entries are defined in the *acl* parameter. If it is zero or greater, the new ACL is applied to the file. If any of the file's base entries (see below) is not mentioned in the new ACL, it is retained but its access mode is set to zero (no access). Hence, routine calls of *setacl* completely define the file's ACL.

As a special case, if *nentries* is negative (that is, a value of ACL_DELOPT (defined in **<sys/acl.h>**)), the *acl* parameter is ignored, all of the file's optional entries, if any, are deleted, and its base entries are left unaltered.

Some of the miscellaneous mode bits in the file's mode might be turned off as a consequence of calling *setacl*. See *chmod*(2).

Access Control Lists

An ACL consists of a series of entries. Entries can be categorized in four levels of specificity:

(*u.g*, mode)	applies to user *u* in group *g*
(*u*.%, mode)	applies to user *u* in any group
(%.*g*, mode)	applies to any user in group *g*
(%.%, mode)	applies to any user in any group

Entries in the ACL must be unique; no two entries can have the same user ID (*uid*) and group ID (*gid*) (see below). Entries can appear in any order. The system orders them as needed for access checking.

The **<sys/acl.h>** header file defines ACL_NSUSER as the non-specific *uid* value and ACL_NSGROUP as the non-specific *gid* value represented by "%" above. If *uid* in an entry is ACL_NSUSER, it is a %.*g* entry. If *gid* in an entry is ACL_NSGROUP, it is a *u*.% entry. If both *uid* and *gid* are non-specific, the file's entry is %.%.

The <**unistd.h**> header file defines meanings of mode bits in ACL entries (R_OK, W_OK, and X_OK). Irrelevant bits in mode values must be zero.

Every file's ACL has three base entries which cannot be added or deleted, but only modified. The base ACL entries are mapped directly from the file's permission bits.

> (<file's owner> . ACL_NSGROUP, <file's owner mode bits>)
> (ACL_NSUSER . <file's group>, <file's group mode bits>)
> (ACL_NSUSER . ACL_NSGROUP, <file's other mode bits>)

In addition, up to 13 optional ACL entries can be set to restrict or grant access to a file.

Altering a base ACL entry's modes with *setacl* changes the file's corresponding permission bits. The permission bits can be altered also with *chmod*(2) and read with *stat*(2).

The number of entries allowed per file (see NACLENTRIES in <**sys/acl.h**>) is small for space and performance reasons. User groups should be created as needed for access control purposes. Since ordinary users cannot create groups, their ability to control file access with ACLs might be somewhat limited.

RETURN VALUE

Upon successful completion, *setacl* and *fsetacl* return a value of zero. If an error occurs, a value of **−1** is returned and the file's ACL is not modified. The global variable **errno** is set to indicate the error.

ERRORS

Setacl and *fsetacl* fail if any of the following is true:

[ENOTDIR] A component of the *path* prefix is not a directory.

[ENOENT] The named file does not exist (for example, *path* is null or a component of *path* does not exist).

[EBADF] *Fildes* is not a valid file descriptor.

[EACCES] A component of the *path* prefix denies search permission.

[EPERM] The effective user ID does not match the owner of the file and the effective user ID is not superuser.

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

[EFAULT] *Path* or *acl* points outside the allocated address space of the process, or *acl* is not as large as indicated by *nentries*.

[EINVAL] There is a redundant entry in the ACL, or *acl* contains an invalid *uid*, *gid*, or *mode* value.

[E2BIG] An attempt was made to set an ACL with more than NACLENTRIES entries.

[EOPNOTSUPP] *Setacl* is not supported on remote files by some networking services.

[ENOSPC] Not enough space on the file system.

[ENFILE] System file table is full.

[ENAMETOOLONG]
The length of *path* exceeds PATH_MAX bytes, or the length of a component of *path* exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.

[ELOOP] Too many symbolic links were encountered in translating the *path* name.

EXAMPLES

The following code fragment defines and sets an ACL on file "../shared" which allows the file's owner to read, write, and execute/search the file, and user 103, group 204 to read the file.

```
#include <unistd.h>
#include <sys/stat.h>
#include <sys/acl.h>

char *filename = "../shared";
struct acl_entry acl [2];
struct stat statbuf;

if (stat (filename, & statbuf) < 0)
        error (...);

acl [0] . uid  = statbuf . st_uid;  /* file owner */
acl [0] . gid  = ACL_NSGROUP;
acl [0] . mode = R_OK | W_OK | X_OK;

acl [1] . uid  = 103;
acl [1] . gid  = 204;
acl [1] . mode = R_OK;

if (setacl (filename, 2, acl))
        error (...);
```

The following call deletes all optional ACL entries from "file1":

```
setacl ("file1", ACL_DELOPT, (struct acl_entry *) 0);
```

DEPENDENCIES

RFA and NFS

Setacl and *fsetacl* are not supported on remote files.

AUTHOR

Setacl and *fsetacl* were developed by HP.

SEE ALSO

access(2), chmod(2), getaccess(2), getacl(2), stat(2), unistd(5).

NAME
setaudid – set the audit ID (aid) for the current process

SYNOPSIS
#include <sys/audit.h>

int setaudid (audid)
aid_t audid;

DESCRIPTION
Setaudid sets the audit ID (*aid*) for the current process. This call is restricted to the superuser.

RETURN VALUE
Upon successful completion, *setaudid* returns a value of **0**; otherwise, a **–1** is returned.

ERRORS
Setaudid fails if one of the following is true:

[EPERM] The caller is not a superuser.

[EINVAL] The audit ID (audid) is invalid.

AUTHOR
Setaudid was developed by HP.

SEE ALSO
getaudid(2).

NAME

setaudproc – controls process level auditing for the current process and its decendents

SYNOPSIS

#include <sys/audit.h>

int setaudproc (aflag)
int aflag;

DESCRIPTION

Setaudproc controls process level auditing for the current process and its decendents. It accomplishes this by setting or clearing the **u_audproc** flag in the **u** area of the calling process. When this flag is set, the system audits the process; when it is cleared, the process is not audited. This call is restricted to superusers.

One of the following *aflags* must be used:

AUD_PROC	Audit the calling process and its decendents.
AUD_CLEAR	Do not audit the calling process and its decendents.

The **u_audproc** flag is inherited by the descendents of a process. consequently, the effect of a call to *setaudproc* is not limited to the current process, but will propagate to all its decendents as well. For example, if *setaudproc* is called with the **AUD_PROC** flag, all subsequent audited system calls in the current process *and its decendents* will be audited until *setaudproc* is called with the **AUD_CLEAR** flag.

Further, *setaudproc* performs its action regardless of whether the user executing the process has been selected to be audited or not. For example, if *setaudproc* is called with the **AUD_PROC** (or the **AUD_CLEAR**) flag, all subsequent audited system calls will be audited (or not audited), regardless of whether the user executing the process has been selected for auditing or not.

Due to these features, *setaudproc* should not be used in most self-auditing applications. *Audswitch*(2) should be used when the objective is to suspend auditing within a process without affecting its decendents or overriding the user selection aspect of the auditing system.

RETURN VALUE

Upon successful completion, a value of **0** is returned; otherwise, **–1** is returned.

AUTHOR

Setaudproc was developed by HP.

SEE ALSO

getaudproc(2), audswitch(2), audusr(1M), audevent(1M), audit(5).

NAME

setevent – set current events and system calls which are to be audited

SYNOPSIS

#include <sys/audit.h>

int setevent (a_syscall, a_event)
struct aud_type *a_syscall;
struct aud_event_tbl *a_event;

DESCRIPTION

Setevent sets the events and system calls to be audited. The event and system call settings in the tables pointed to by *a_syscall* and *a_event* become the current settings. This call is restricted to the superuser.

RETURN VALUE

Upon successful completion, *setevent* returns a value of **0**; otherwise, a **−1** is returned.

ERRORS

Setevent fails if the following is true:

[EPERM] The caller is not a superuser.

AUTHOR

Setevent was developed by HP.

SEE ALSO

getevent(2), audevent(1M).

NAME

setgroups – set group access list

SYNOPSIS

#include <sys/param.h>
#include <sys/types.h>
setgroups(ngroups, gidset)
int ngroups;
gid_t *gidset;

DESCRIPTION

Setgroups sets the group access list of the current user process according to the array *gidset*. The parameter *ngroups* indicates the number of entries in the array and must be no more than NGROUPS, as defined in *<sys/param.h>*.

Only the superuser may set new groups by adding to the group access list of the current user process; any user may delete groups from it.

RETURN VALUE

A 0 value is returned on success, −1 on error, with an error code stored in *errno*.

ERRORS

The *setgroups* call will fail if:

[EPERM] The caller is not the superuser and has attempted to set new groups.

[EFAULT] The address specified for *gidset* is outside the process address space. The reliable detection of this error will be implementation dependent.

[EINVAL] *Ngroups* is greater than NGROUPS or not positive.

[EINVAL] An entry in *gidset* is not a valid group ID.

AUTHOR

Setgroups was developed by the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

getgroups(2), initgroups(3C)

NAME

sethostname – set name of host cpu

SYNOPSIS

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

DESCRIPTION

This call sets the name of the host processor to *name,* which has a length of *namelen* characters. This is normally executed by **/etc/rc** when the system is bootstrapped. Host names are limited to MAXHOSTNAMELEN characters; MAXHOSTNAMELEN is defined in <**sys/param.h**>.

ERRORS

Sethostname fails and returns an error if:

[EPERM] It is not executed by the superuser.

[EFAULT] *Name* points to an illegal address. The reliable detection of this error is implementation dependent.

AUTHOR

Sethostname was developed by the University of California, Berkeley.

SEE ALSO

hostname(1), uname(1), gethostname(2), uname(2).

NAME

setpgid, setpgrp2 – set process group ID for job control

SYNOPSIS

#include <sys/types.h>

int setpgid(pid,pgid)
pid_t *pid, pgid;*

int setpgrp2(pid,pgid)
pid_t *pid, pgid;*

DESCRIPTION

Setpgid or *setpgrp2* causes the process specified by *pid* to join an existing process group or create a new process group within the session of the calling process. The process group ID of the process whose process ID is *pid* is set to *pgid*. If *pid* is zero, the process ID of the calling process is used. If *pgid* is zero, the process ID of the indicated process is used. The process group ID of a session leader does not change.

Setpgrp2 is provided for backward compatibility only.

ERRORS

Setpgid or *setpgrp2* fails and no change occurs if any of the following are true:

[EACCES]	The value of *pid* matches the process ID of a child process of the calling process and the child process has successfully executed one of the *exec*(2) functions.
[EINVAL]	The value of *pgid* is less than zero or is outside the range of valid process group ID values.
[EPERM]	The process indicated by *pid* is a session leader.
[EPERM]	The value of *pid* is valid but matches the process ID of a child process of the calling process, and the child process is not in the same session as the calling process.
[EPERM]	The value of *pgid* does not match the process ID of the process indicated by *pid* and there is no process with a process group ID that matches the value of *pgid* in the same session as the calling process.
[ESRCH]	The value of *pid* does not match the process ID of the calling process or of a child process of the calling process.

RETURN VALUE

Upon successful completion, *setpgid* or *setpgrp2* returns zero. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

AUTHOR

Setpgid and *setpgrp2* were developed by HP and the University of California, Berkeley.

SEE ALSO

bsdproc(2), exec(2), exit(2), fork(2), getpid(2), kill(2), setsid(2), signal(2), termio(7).

STANDARDS CONFORMANCE

setpgid: XPG3, POSIX.1, FIPS 151-1

setpgrp2: not applicable

NAME

setresuid, setresgid – set real, effective, and saved user and group IDs

SYNOPSIS

int setresuid (ruid, euid, suid)
int ruid, euid, suid;

int setresgid (rgid, egid, sgid)
int rgid, egid, sgid;

DESCRIPTION

Setresuid sets the real, effective and/or saved user ID of the calling process.

If the current real, effective or saved user ID is equal to the super-user's user ID, *setresuid* sets the real, effective and saved user IDs to *ruid*, *euid* and *suid*, respectively. Otherwise, *setresuid* will only set the real, effective and saved user IDs if *ruid*, *euid* and *suid* each match at least one of the current real, effective or saved user IDs.

If *ruid*, *euid* or *suid* is **-1**, *setresuid* will leave the current real, effective or saved user ID unchanged.

Setresgid sets the real, effective and/or saved group ID of the calling process.

If the current real, effective or saved user ID is equal to the super-user's user ID, *setresgid* sets the real, effective and saved group IDs to *rgid*, *egid* and *sgid*, respectively. Otherwise, *setresgid* will only set the real, effective and saved group IDs if *rgid*, *egid* and *sgid* each match at least one of the current real, effective or saved group IDs.

If *rgid*, *egid* or *sgid* is **-1**, *setresgid* will leave the current real, effective or saved group ID unchanged.

ERRORS

Setresuid and *setresgid* will fail and return **-1** if:

[EINVAL] *Ruid*, *euid* or *suid* (*rgid*, *egid* or *sgid*) is not a valid user (group) ID.

[EPERM] None of the conditions above are met.

RETURN VALUE

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

AUTHOR

Setresuid and *setresgid* were developed by HP.

SEE ALSO

exec(2), getuid(2), setuid(2).

NAME

setsid, setpgrp – create session and set process group ID

SYNOPSIS

#include <sys/types.h>

pid_t setsid()

pid_t setpgrp()

DESCRIPTION

If the calling process is not a process group leader, *setsid* or *setpgrp* creates a new session. The calling process becomes the session leader of this new session, becomes the process group leader of a new process group, and has no controlling terminal. The process group ID of the calling process is set equal to the process ID of the calling process. The calling process is the only process in the new process group, and the only process in the new session.

Setpgrp is provided for backward compatibility only.

ERRORS

No change occurs if any of the following conditions occur. In addition, *setsid* fails with the following errors:

[EPERM]	The calling process is already a process group leader.
[EPERM]	The process group ID of a process other than the calling process matches the process ID of the calling process.

RETURN VALUE

Setpgrp returns the value of the process group ID of the calling process.

Upon successful completion, *setsid* returns the value of the new process group ID of the calling process. Otherwise, a value of −1 is returned and errno is set to indicate the error.

AUTHOR

Setpgrp and *setsid* were developed by HP and AT&T.

SEE ALSO

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

STANDARDS CONFORMANCE

setpgrp: SVID2, XPG2

setsid: XPG3, POSIX.1, FIPS 151-1

NAME

setuid, setgid – set user and group IDs

SYNOPSIS

```
#include <sys/types.h>
int setuid (uid)
uid_t uid;

int setgid (gid)
gid_t gid;
```

DESCRIPTION

Setuid sets the real-user-ID (*ruid*), effective-user-ID (*euid*), and/or saved-user-ID (*suid*) of the calling process. The super-user's *euid* is zero. The following conditions govern *setuid*'s behavior:

> If the *euid* is zero, *setuid* sets the *ruid*, *euid*, and *suid* to *uid*.
>
> If the *euid* is not zero, but the argument *uid* is equal to the *ruid* or the *suid*, *setuid* sets the *euid* to *uid*; the *ruid* and *suid* remain unchanged. (If a set-user-ID program is not running as super-user, it can change its *euid* to match its *ruid* and reset itself to the previous *euid* value.)
>
> If the *euid* is not zero, but the argument *uid* is equal to the *euid*, and the calling process is a member of a group that has the PRIV_SETRUGID privilege (see *privgrp*(4)), *setuid* sets the *ruid* to *uid*; the *euid* and *suid* remain unchanged.

Setgid sets the real-group-ID (*rgid*), effective-group-ID (*egid*), and/or saved-group-ID (*sgid*) of the calling process. The following conditions govern *setgid*'s behavior:

> If the *euid* is zero, *setgid* sets the *rgid* and *egid* to *gid*.
>
> If the *euid* is not zero, but the *rgid* or *sgid* is equal to *gid*, and the calling process is a member of a group that has the PRIV_SETRUGID privilege (see *privgrp*(4)), *setgid* sets the *egid* to *gid*; the *rgid* and *sgid* remain unchanged.
>
> If the *euid* is not zero, but the *gid* is equal to the *egid*, *setgid* sets the *rgid* to *gid*; the *egid* and *sgid* remain unchanged.

RETURN VALUE

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

ERRORS

Setuid and *setgid* fail and return **–1** if either of the following is true:

[EPERM]	None of the conditions above are met.
[EINVAL]	*Uid* (*gid*) is not a valid user (group) ID.

WARNINGS

It is recommended that the PRIV_SETRUGID capability be avoided, as it is provided for backward compatibility. This feature may be modified or dropped from future HP-UX releases. When changing the real user ID and real group ID, use of *setresuid*(2) and *setresgid*(2) are recommended instead.

AUTHOR

Setuid was developed by AT&T, the University of California, Berkeley and HP.

Setgid was developed by AT&T.

SEE ALSO

exec(2), getprivgrp(2), getuid(2), setresuid(2) privgrp(4).

STANDARDS CONFORMANCE

setuid: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

setgid: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

shmctl – shared memory control operations

SYNOPSIS

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

int shmctl (shmid, cmd, buf)
int shmid, cmd;
struct shmid_ds *buf;

DESCRIPTION

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

IPC_STAT Place the current value of each member of the data structure associated with *shmid* into the structure pointed to by *buf*. The contents of this structure are defined in the *glossary*.

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

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

IPC_RMID
Remove the shared memory identifier specified by *shmid* from the system and destroy the shared memory segment and data structure associated with it. If the segment is attached to one or more processes, then the segment key is changed to IPC_PRIVATE and the segment is marked removed. The segment will disappear when the last attached process detaches it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of super-user or to the value of either **shm_perm.uid** or **shm_perm.cuid** in the data structure associated with *shmid*.

SHM_LOCK
Lock the shared memory segment specified by *shmid* in memory. This *cmd* can only be executed by a process that either has an effective user ID equal to super-user or has an effective user ID equal to the value of either **shm_perm.uid** or **shm_perm.cuid** in the data structure associated with *shmid* and has PRIV_MLOCK privilege (see *setprivgrp* on *getprivgrp*(2)).

SHM_UNLOCK
Unlock the shared memory segment specified by *shmid*. This *cmd* can only be executed by a process that either has an effective user ID equal to super-user or has an effective user ID equal to the value of either **shm_perm.uid** or **shm_perm.cuid** in the data structure associated with *shmid* and has PRIV_MLOCK privilege (see *setprivgrp* on *getprivgrp*(2)).

EXAMPLES

The following call to *shmctl* locks in memory the shared memory segment represented by *myshmid*. This example assumes the process has a valid shmid, which can be obtained by calling *shmget*(2).

shmctl (myshmid, SHM_LOCK, 0);

The following call to *shmctl* removes the shared memory segment represented by *myshmid*. This example assumes the process has a valid shmid, which can be obtained by calling *shmget*(2).

shmctl (myshmid, IPC_RMID, 0);

ERRORS

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

[EINVAL] *Shmid* is not a valid shared memory identifier.

[EINVAL] *Cmd* is not a valid command.

[EACCES] *Cmd* is equal to **IPC_STAT** and operation permission is denied to the calling process (see *glossary*).

[EPERM] *Cmd* is equal to **IPC_RMID, IPC_SET, SHM_LOCK,** or **SHM_UNLOCK** and the effective user ID of the calling process is not equal to that of super-user and it is not equal to the value of either **shm_perm.uid** or **shm_perm.cuid** in the data structure associated with *shmid*.

[EPERM] *Cmd* is equal to **SHM_LOCK** or **SHM_UNLOCK** and the effective user ID of the calling process is not equal to that of super-user and the calling process does not have PRIV_MLOCK privilege (see *setprivgrp* on *getprivgrp*(2)).

[EINVAL] *Cmd* is equal to **SHM_UNLOCK** and the shared-memory segment specified by *shmid* is not locked in memory.

[EFAULT] *Buf* points to an illegal address. The reliable detection of this error will be implementation dependent.

[ENOMEM] *Cmd* is equal to SHM_LOCK and there is not sufficient lockable memory to fill the request.

RETURN VALUE

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

DEPENDENCIES

Series 300

[EACCES] *Shmid* is the id of a shared memory segment currently being used by the system to implement other features (see *graphics*(7) and *iomap*(7)).

AUTHOR

Shmctl was developed by AT&T and HP.

SEE ALSO

ipcrm(1), ipcs(1), shmget(2), shmop(2), stdipc(3C).

STANDARDS CONFORMANCE

shmctl: SVID2, XPG2, XPG3

NAME

shmget – get shared memory segment

SYNOPSIS

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

int shmget (key, size, shmflg)
key_t key;
int size, shmflg;

DESCRIPTION

Shmget returns the shared memory identifier associated with *key*.

A shared memory identifier and associated data structure and shared memory segment of size *size* bytes (see *glossary*) are created for *key* if one of the following is true:

> *Key* is equal to **IPC_PRIVATE**. This call creates a new identifier, subject to available resources. The identifier will never be returned by another call to *shmget* until it has been released by a call to *shmctl*. The identifier should be used among the calling process and its descendents; however, it is not a requirement. The resource can be accessed by any process having the proper permissions.
>
> *Key* does not already have a shared memory identifier associated with it, and (*shmflg* & **IPC_CREAT**) is "true".

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

> **Shm_perm.cuid**, **shm_perm.uid**, **shm_perm.cgid**, and **shm_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.
>
> The low-order 9 bits of **shm_perm.mode** are set equal to the low-order 9 bits of *shmflg*. **Shm_segsz** is set equal to the value of *size*.
>
> **Shm_lpid**, **shm_nattch**, **shm_atime**, and **shm_dtime** are set equal to 0.
>
> **Shm_ctime** is set equal to the current time.

EXAMPLES

The following call to *shmget* returns a unique shmid for the newly created shared memory segment of 4096 bytes:

```
int myshmid;

myshmid = shmget (IPC_PRIVATE, 4096, 0600);
```

ERRORS

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

[EINVAL] *Size* is less than the system-imposed minimum or greater than the system-imposed maximum.

[EACCES] A shared memory identifier exists for *key* but operation permission (see *glossary*) as specified by the low-order 9 bits of *shmflg* would not be granted.

[EINVAL] A shared memory identifier exists for *key* but the size of the segment associated with it is less than *size* and *size* is not equal to zero.

[ENOENT] A shared memory identifier does not exist for *key* and (*shmflg* & **IPC_CREAT**) is "false".

[ENOSPC] A shared memory identifier is to be created but the system-imposed limit on the maximum number of allowed shared memory identifiers system wide would be exceeded.

[ENOMEM] A shared memory identifier and associated shared memory segment are to be created but the amount of available physical memory is not sufficient to fill the request.

[EEXIST] A shared memory identifier exists for *key* but ((*shmflg* & **IPC_CREAT**) **&&** (*shmflg* & **IPC_EXCL**)) is "true".

RETURN VALUE

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

SEE ALSO

ipcrm(1), ipcs(1), shmctl(2), shmop(2), stdipc(3C).

STANDARDS CONFORMANCE

shmget: SVID2, XPG2, XPG3

NAME

shmat, shmdt – shared memory operations

SYNOPSIS

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

char *shmat (shmid, shmaddr, shmflg)
int shmid;
char *shmaddr
int shmflg;

int shmdt (shmaddr)
char *shmaddr;

DESCRIPTION

Shmat attaches the shared memory segment associated with the shared memory identifier specified by **shmid** to the data segment of the calling process.

On Series 800 systems, if the shared memory segment is not already attached, *shmaddr* must be specified as zero and the segment is attached at a location selected by the operating system. That location is identical in all processes accessing that shared memory object.

If the shared memory segment is already attached, a non-zero value of *shmaddr* is accepted, provided the specified address is identical to the current attach address of the segment.

On Series 300 systems, *shmaddr* can be specified as a non-zero value as a machine-dependent extension (see DEPENDENCIES below). However, those systems do not necessarily guarantee that a given shared memory object appears at the same address in all processes that access it, unless the user specifies an address.

The segment is attached for reading if (*shmflg* & **SHM_RDONLY**) is "true" otherwise it is attached for reading and writing It is not possible to attach a segment for write only.

Shmdt detaches from the calling process's data segment the shared memory segment located at the address specified by *shmaddr*.

RETURN VALUE

Upon successful completion, the return value is as follows:

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

Shmdt returns a value of **0**.

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

ERRORS

Shmat fails and does not attach the shared memory segment if one or more of the following is true:

[EINVAL] *Shmid* is not a valid shared memory identifier.

[EACCES] Operation permission is denied to the calling process.

[ENOMEM] The available data space is not large enough to accommodate the shared memory segment.

[EINVAL] *Shmaddr* is not zero and the machine does not permit non-zero values or *shmaddr* is not equal to the current attach location for the shared memory segment.

[EMFILE] The number of shared memory segments attached to the calling process exceed the system-imposed limit.

Shmdt fails and returns **−1** if the following is true:

[EINVAL] *Shmdt* fails and does not detach the shared memory segment if *shmaddr* is not the data segment start address of a shared memory segment.

EXAMPLES

The following call to *shmat* attaches the shared memory segment to the process. This example assumes the process has a valid *shmid*, which can be obtained by calling *shmget*(2).

```
char *shmptr, *shmat();
shmptr = shmat(myshmid, (char *)0, 0);
```

The following call to *shmdt* then detaches the shared memory segment.

```
shmdt (shmptr);
```

DEPENDENCIES

Series 300

Shmaddr can be non-zero, and if it is, the segment is attached at the address specified by one of the following criteria:

If *shmaddr* is equal to zero, the segment is attached at the first available address as selected by the system. The selected value varies for each process accessing that shared memory object.

If *shmaddr* is not equal to zero and (*shmflg* & **SHM_RND**) is "true", the segment is attached at the address given by (*shmaddr* - (*shmaddr* % **SHMLBA**)). The character % is the C language modulus operator.

If *shmaddr* is not equal to zero and (*shmflg* & **SHM_RND**) is "false", the segment is attached at the address given by *shmaddr*.

This form of *shmat* fails and does not attach the shared memory segment if one or more of the following is true:

[EACCES] *Shmid* is the ID of a shared memory segment currently being used by the system to implement other features (see *graphics*(7) and *iomap*(7)).

[EINVAL] *Shmaddr* is not equal to zero, and the value of (*shmaddr* - (*shmaddr* % **SHMLBA**)) is an illegal address.

[EINVAL] *Shmaddr* is not equal to zero, (*shmflg* & **SHM_RND**) is "false", and the value of *shmaddr* is an illegal address.

[ENOMEM] The calling process is locked (see *plock*(2)) and there is not sufficient lockable memory to support the process-related data structure overhead.

Series 800

Shmat will fail and return **−1** if the following is true:

[EINVAL] The calling process is already attached to *shmid*.

SEE ALSO

ipcs(1), exec(2), exit(2), fork(2), shmctl(2), shmget(2), stdipc(3C).

STANDARDS CONFORMANCE

shmat: SVID2, XPG2, XPG3

shmdt: SVID2, XPG2, XPG3

NAME

sigaction – examine and change signal action

SYNOPSIS

#include <signal.h>

int sigaction (sig, act, oact)
int sig ;
struct sigaction *act, *oact ;

DESCRIPTION

Sigaction allows the calling process to examine and specify the action to be taken on delivery of a specific signal. The argument *sig* specifies the signal; acceptable values are defined in **<signal.h>.** More details on the semantics of specific signals can be found on the *signal*(5) manual page.

The *sigaction* structure and type *sigset_t* are defined in **<signal.h>.**

Act and *oact* are pointers to *sigaction* structures that include the following elements:

```
void       (*sa_handler)();
sigset_t   sa_mask;
int        sa_flags;
```

Unless it is a null pointer, the argument *act* points to a structure specifying the action to be taken when delivering the specified signal. If the argument *oact* is not a null pointer, the action previously associated with the signal is stored in the location pointed to by *oact.* If the argument *act* is a null pointer, signal handling is unchanged; thus *sigaction* can be used to inquire about the current handling of a given signal.

The *sa_handler* member of the *sigaction* structure is assigned one of three values: **SIG_DFL, SIG_IGN,** or a *function address.* The actions prescribed by these values are as follows:

SIG_DFL Execute default action for signal.

Upon receipt of the signal *sig,* the default action (specified on *signal*(5)) is performed. The default action for most signals is to terminate the process.

A pending signal is discarded (whether or not it is blocked) if *sigaction* is set to **SIG_DFL** for a pending signal whose default action is to ignore the signal (as in the case of **SIGCHLD**).

SIG_IGN Ignore the signal.

Setting a signal action to **SIG_IGN** causes a pending signal to be discarded, whether or not it is blocked.

The **SIGKILL** and **SIGSTOP** signals cannot be ignored.

function address Catch the signal.

Upon receipt of the signal *sig,* the receiving process executes the signal-catching function pointed to by *sa_handler.* The signal-catching function is entered as a C language function call. Details on the arguments passed to this function can be found on the *signal*(5) manual page.

The signals **SIGKILL** and **SIGSTOP** cannot be caught.

When a signal is caught by a signal-catching function installed by *sigaction,* a new mask is calculated and installed for the duration of the signal-catching function, or until a call is made to *sigprocmask*(2) or *sigsuspend*(2). This mask is formed by taking the union of the current signal mask, the signal to be delivered, and unless the **SA_RESETHAND** flag is set (see below), the signal mask specified in the *sa_mask* field of the *sigaction* structure associated with the signal being delivered. If and when the signal-catching function returns normally, the original signal mask is restored.

Once an action is installed for a specific signal, it remains installed until another action is explicitly requested, or until one of the *exec*(2) functions is called.

If the previous action for *sig* was established by *signal*(2), the values of the fields returned in the structure pointed to by *oact* are unspecified; in particular, *oact->sa_handler* is not necessarily the same value passed to *signal*(2). However, if a pointer to the same structure or a copy thereof is passed to a subsequent call to *sigaction*(2) via the *act* argument, handling of the signal is reinstated as if the original call to *signal*(2) were repeated.

The set of signals specified by the *sa_mask* field of the *sigaction* structure pointed to by the *act* argument cannot block the **SIGKILL** or **SIGSTOP** signal. This is enforced by the system without causing an error to be indicated.

The *sa_flags* field in the *sigaction* structure can be used to modify the behavior of the specified signal. The following flag bits, defined in the **<signal.h>** header, can be set in *sa_flags:*

SA_NOCLDSTOP Do not generate **SIGCHLD** when untraced children stop (see *ptrace*(2)).

SA_ONSTACK Use the space reserved by *sigspace*(2) for signal processing.

SA_RESETHAND Use the semantics of *signal*(2). The signal mask specified by the *sa_mask* field is not used when setting up the effective signal mask for the signal handler. If the signal is not one of those marked "not reset when caught" (see *signal*(5)), the default action for the signal is reinstated when the signal is caught, prior to entering the signal-catching function. The "not reset when caught" distinction is insignificant when *sigaction* is called and **SA_RESETHAND** is not set.

RETURN VALUE

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

ERRORS

Sigaction fails and no new signal-catching function is installed if one of the following is true:

[EINVAL] The value of the *sig* argument is not a valid signal number, or an attempt is made to supply an action other than **SIG_DFL** for the **SIGKILL** or **SIGSTOP** signal.

[EFAULT] *Act* or *oact* points to an invalid address. The reliable detection of this error is implementation dependent.

AUTHOR

Sigaction was derived from the IEEE Standard POSIX 1003.1-1988.

SEE ALSO

ptrace(2), sigprocmask(2), sigpending(2), sigspace(2), sigsuspend(2), sigsetops(3C), signal(5).

STANDARDS CONFORMANCE

sigaction: XPG3, POSIX.1, FIPS 151-1

NAME

sigblock – block signals

SYNOPSIS

#include <signal.h>

long sigblock(mask);
long mask;

DESCRIPTION

Sigblock causes the signals specified in *mask* to be added to the set of signals currently being blocked from delivery. Signal *i* is blocked if the *i*-th bit in *mask* is **1**, as specified with the macro **sigmask**(*i*).

It is not possible to block signals that cannot be ignored, as documented in *signal*(5); this restriction is silently imposed by the system.

Sigsetmask(2) can be used to set the mask absolutely.

RETURN VALUE

The previous set of masked signals is returned.

EXAMPLES

The following call to *sigblock* adds the SIGUSR1 and SIGUSR2 signals to the mask of signals currently blocked for the process:

```
long oldmask;

oldmask = sigblock (sigmask (SIGUSR1) | sigmask (SIGUSR2));
```

WARNINGS

Sigblock should not be used in conjunction with the facilities described under *sigset*(2V).

AUTHOR

Sigblock was developed by the University of California, Berkeley.

SEE ALSO

kill(2), sigprocmask(2), sigsetmask(2), sigvector(2).

NAME

signal – specify what to do upon receipt of a signal

SYNOPSIS

#include <signal.h>

```
void (*signal (sig, action))()
int sig;
void (*action)();

void action (sig [, code, scp ])
int sig, code;
struct sigcontext * scp;
```

DESCRIPTION

Signal allows the calling process to choose one of three ways to handle the receipt of a specific signal. *Sig* specifies the signal and *action* specifies the choice.

Acceptable values for *sig* are defined in <signal.h>. The specific signals are described in full on the *signal*(5) manual page.

The value of the *action* argument specifies what to do upon the receipt of signal *sig*, and should be one of the following:

SIG_DFL
: Execute the default action, which varies depending on the signal. The default action for most signals is to terminate the process (see *signal*(5)).

  A pending signal is discarded (whether or not it is blocked) if *action* is set to SIG_DFL but the default action of the pending signal is to ignore the signal (as in the case of SIGCLD).

SIG_IGN
: Ignore the signal.
  When *signal* is called with *action* set to SIG_IGN and an instance of the signal *sig* is pending, the pending signal is discarded, whether or not it is blocked.

  The SIGKILL and SIGSTOP signals cannot be ignored.

address
: Catch the signal.
  Upon receipt of the signal *sig*, reset the value of *action* for the caught signal to SIG_DFL (except signals marked with "not reset when caught"; see *signal*(5)), call the signal-catching function to which *address* points, and resume executing the receiving process at the point it was interrupted.

  The signal-catching function is called with the following three parameters:

  sig — The signal number.

  code — A word of information usually provided by the hardware.

  scp — A pointer to the machine-dependent structure *sigcontext* defined in **<signal.h>**.

Depending on the value of *sig*, *code* can be zero and/or *scp* can be NULL. The meanings of *code* and *scp* and the conditions determining when they are other than zero or NULL are implementation dependent (see DEPENDENCIES below). It is possible for *code* to always be zero, and *scp* to always be NULL.

The pointer *scp* is valid only during the context of the signal-catching function.

The signals SIGKILL and SIGSTOP cannot be caught.

RETURN VALUE

Upon successful completion, *signal* returns the previous value of *action* for the specified signal *sig*. Otherwise, a value of **SIG_ERR** is returned and **errno** is set to indicate the error.

ERRORS

Signal fails if the following is true:

[EINVAL] *Sig* is an illegal signal number, or is equal to SIGKILL or SIGSTOP.

EXAMPLES

The following call to *signal* sets up a signal-catching function for the SIGINT signal:

void myhandler();

(void) signal(SIGINT, myhandler);

WARNINGS

Signal should not be used in conjunction with the facilities described under *bsdproc*(2), *sigaction*(2), *sigset*(2V), or *sigvector*(2).

The *signal* function does not detect an invalid value for the *action* argument, and if it does not equal SIG_DFL or SIG_IGN, or point to a valid function address, subsequent receipt of the signal *sig* causes undefined results.

DEPENDENCIES

Series 300

The *code* word is always zero for all signals except SIGILL and SIGFPE. For SIGILL, *code* has the following values:

- 0 illegal instruction;
- 6 check instruction;
- 7 TRAPV;
- 8 privilege violation.

Refer to the MC6800xx processor documentation for more detailed information about the meaning of the SIGILL errors.

For SIGFPE, *code* has the following values:

- 0 software floating point exception;
- 5 integer divide-by-zero.
- 0x8xxxxxx
  any value with the high-order bit set indicates an exception while using the HP98248 floating point accelerator. The value of (*code* &~ 0x8000000) is the value of the HP 98248 status register. Refer to the HP 98248 documentation for more detailed information.
- other
  any other value indicates an exception while using the MC68881 or MC68882 floating point coprocessor. The value of *code* is the value of the MC68881 or MC68882 status register. Refer to the MC68881 documentation for more detailed information.

Series 800

The structure pointer *scp* is always defined.

The *code* word is always zero for all signals except SIGILL and SIGFPE. For SIGILL, *code* has the following values:

- 8 illegal instruction trap;
- 9 break instruction trap;
- 10 privileged operation trap;
- 11 privileged register trap.

For SIGFPE, *code* has the following values:

12 overflow trap;
13 conditional trap;
14 assist exception trap;
22 assist emulation trap.

AUTHOR

Signal was developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

kill(1), init(1M), exit(2), kill(2), lseek(2), pause(2), sigaction(2), sigvector(2), wait(2), abort(3C), setjmp(3C), signal(5).

STANDARDS CONFORMANCE

signal: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

sigpause – atomically release blocked signals and wait for interrupt

SYNOPSIS

#include <signal.h>

long sigpause(mask)
long mask;

DESCRIPTION

Sigpause blocks signals according to the value of *mask* in the same manner as *sigsetmask*(2), then atomically waits for an unmasked signal to arrive. On return *sigpause* restores the current signal mask to the value that existed before the *sigpause* call. When no signals are to be blocked, a value of 0L is used for *mask.*

In normal usage, a signal is blocked using *sigblock*(2). To begin a critical section variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses, awaiting work by using *sigpause* with the mask returned by *sigblock*.

RETURN VALUE

Sigpause will terminate when it is interrupted by a signal. When *sigpause* terminates, it will return −1 and set **errno** to **EINTR**.

EXAMPLES

The following call to *sigpause* waits until the calling process receives a signal:

sigpause (0L);

The following example blocks the SIGIO signal until *sigpause* is called. When a signal is received at the *sigpause* statement, the signal mask is restored to its value before *sigpause* was called:

```
long savemask;
savemask = sigblock (sigmask (SIGIO));
/* critical section */
sigpause (savemask);
```

WARNINGS

Check all references to *signal*(5) for appropriateness on systems that support *sigvector*(2). *Sigvector*(2) can affect the behavior described on this page.

Sigpause should not be used in conjunction with the facilities described under *sigset*(2V).

AUTHOR

Sigpause was developed by the University of California, Berkeley.

SEE ALSO

sigblock(2), sigsetmask(2), sigsuspend(2), sigvector(2).

NAME

sigpending – examine pending signals

SYNOPSIS

#include <signal.h>

int sigpending (set)
sigset_t *set ;

DESCRIPTION

Sigpending stores sets of signals that are blocked from delivery and are pending to the calling process, at the location pointed to by *set.*

RETURN VALUE

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

ERRORS

Sigpending fails if the following is true:

[EFAULT] *Set* points to an invalid address. The reliable detection of this error is implementation dependent.

AUTHOR

Sigpending was derived from the IEEE Standard POSIX 1003.1-1988.

SEE ALSO

sigaction(2), sigsuspend(2), sigprocmask(2), sigsetops(3C), signal(5).

STANDARDS CONFORMANCE

sigpending: XPG3, POSIX.1, FIPS 151-1

NAME

sigprocmask – examine and change blocked signals

SYNOPSIS

#include <signal.h>

int sigprocmask (how, set, oset)
int how ;
sigset_t *set, *oset ;

DESCRIPTION

Sigprocmask allows the calling process to examine and/or change its signal mask.

Unless it is a null pointer, the argument *set* points to a set of signals to be used to change the currently blocked set.

The argument *how* indicates how the set is changed, and consists of one of the following values (see <signal.h>):

SIG_BLOCK The resulting set is the union of the current set and the signal set pointed to by *set.*

SIG_UNBLOCK The resulting set is the intersection of the current set and the complement of the signal set pointed to by *set.*

SIG_SETMASK The resulting set is the signal set pointed to by *set.*

If the argument *oset* is not a null pointer, the previous signal mask is stored in the location pointed to by *oset.* If *set* is a null pointer, the value of the argument *how* is insignificant and the process's signal mask is unchanged; thus the call can be used to inquire about currently blocked signals.

If any pending unblocked signals remain after the call to *sigprocmask,* at least one of those signals is delivered before the call to *sigprocmask* returns.

It is impossible to block the **SIGKILL** or **SIGSTOP** signal. This is enforced by the system without causing an error to be indicated.

The process's signal mask is not changed if *sigprocmask* fails for any reason.

RETURN VALUE

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

ERRORS

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

[EINVAL] The value of the *how* argument is not equal to one of the defined values.

[EFAULT] *Set* or *oset* points to an invalid address. The reliable detection of this error is implementation dependent.

AUTHOR

Sigprocmask was derived from the IEEE Standard POSIX 1003.1-1988.

SEE ALSO

sigaction(2), sigsuspend(2), sigpending(2), sigsetops(3C), signal(5).

STANDARDS CONFORMANCE

sigprocmask: XPG3, POSIX.1, FIPS 151-1

NAME

sigset, sighold, sigrelse, sigignore, sigpause – signal management

SYNOPSIS

```
#include <signal.h>

void (* sigset ( sig, func ))()
int sig;
int (* func )();

int sighold (sig)
int sig;

int sigrelse (sig)
int sig;

int sigignore (sig)
int sig;

int sigpause (sig)
int sig;
```

DESCRIPTION

The system defines a set of signals that can be delivered to a process. The set of signals is defined in *signal*(5), along with the meaning and side effects of each signal. An alternate mechanism for handling these signals is defined here. The facilities described here should not be used in conjunction with the other facilities described under *signal*(2), *sigvector*(2), *sigblock*(2), *sigsetmask*(2), *sigpause*(2) and *sigspace*(2).

Sigset allows the calling process to choose one of four ways to handle the receipt of a specific signal. *Sig* specifies the signal and *func* specifies the choice.

Sig can be any one of the signals described under *signal*(5) except SIGKILL or SIGSTOP.

Func is assigned one of four values: SIG_DFL, SIG_IGN, SIG_HOLD or a *function address*. The actions prescribed by SIG_DFL and SIG_IGN are described under *signal*(5). The action prescribed by SIG_HOLD and *function address* are described below:

SIG_HOLD Hold signal.
The signal *sig* is held upon receipt. Any pending signal of this signal type remains held. Only one signal of each type is held.
Note: the signals SIGKILL, SIGCONT and SIGSTOP cannot be held.

function address Catch signal.
Func must be a pointer to a function, the signal-catching handler, that is called when signal *sig* occurs. *Sigset* specifies that the process calls this function upon receipt of signal *sig*. Any pending signal of this type is released. This handler address is retained across calls to the other signal management functions listed here. Upon receipt of the signal *sig*, the receiving process executes the signal-catching function pointed to by *func* as described under *signal*(5) with the following differences:
Before calling the signal-catching handler, the system signal action of *sig* is set to SIG_HOLD. During a normal return from the signal-catching handler, the system signal action is restored to *func* and any held signal of this type is released. If a non-local goto (*longjmp*(3C)) is taken, *sigrelse* must be called to restore the system signal action to *func* and release any held signal of this type.

Sighold(2) holds the signal *sig*. *Sigrelse*(2) restores the system signal action of *sig* to that specified previously by *sigset*. *Sighold* and *sigrelse* are used to establish critical regions of code. *Sighold* is analogous to raising the priority level and deferring or holding a signal until the priority is lowered by *sigrelse*.

Sigignore sets the action for signal *sig* to SIG_IGN. (See *signal*(5)).

Sigpause suspends the calling process until it receives an unblocked signal. If the signal *sig* is held, it is released before the process pauses. *Sigpause* is useful for testing variables that are changed when a signal occurs. For example, *sighold* should be used to block the signal first, then test the variables. If they have not changed, call *sigpause* to wait for the signal.

These functions can be linked into a program by giving the **−lV3** option to *ld*(1).

RETURN VALUE

Upon successful completion, *sigset* returns the previous value of the system signal action for the specified signal *sig*. Otherwise, a value of SIG_ERR is returned and **errno** is set to indicate the error. SIG_ERR is defined in <**signal.h**>.

For the other functions, a **0** value indicates that the call succeeded. A **−1** return value indicates an error occurred and **errno** is set to indicate the reason.

ERRORS

The *sigset* function fails and the system signal action for *sig* is not changed if the following occurs:

[EFAULT] The *func* argument points to memory that is not a valid part of the process address space. The reliable detection of this error is implementation dependent.

The *sigset*, *sighold*, *sigrelse*, *sigignore*, and *sigpause* functions fail and the system signal action for *sig* is not changed if one of the following occurs:

[EINVAL] *Sig* is not a valid signal number.

[EINVAL] An attempt is made to ignore, hold, or supply a handler for a signal that cannot be ignored, held, or caught; see *signal*(5).

The *sigpause* function returns when the following occurs:

[EINTR] A signal was caught.

WARNINGS

These signal facilities should not be used in conjunction with *bsdproc*(2), *signal*(2), *sigvector*(2), *sigblock*(2), *sigsetmask*(2), *sigpause*(2) and *sigspace*(2).

SEE ALSO

kill(1), kill(2), signal(2), pause(2), wait(2), abort(3C), setjmp(3C), signal(5).

NAME

sigsetmask – set current signal mask

SYNOPSIS

#include <signal.h>

long sigsetmask(mask);
long mask;

DESCRIPTION

Sigsetmask sets the current signal mask (those signals that are blocked from delivery). Signal *i* is blocked if the *i*-th bit in *mask*, as specified with the macro **sigmask(*i*)**, is a **1**.

It is not possible to mask signals that cannot be ignored, as documented in *signal*(5); this restriction is silently imposed by the system.

Sigblock(2) can be used to add elements to the set of blocked signals.

RETURN VALUE

The previous set of masked signals is returned.

EXAMPLES

The following call to *sigsetmask* causes only the SIGUSR1 and SIGUSR2 signals to be blocked:

```
long oldmask;

oldmask = sigsetmask (sigmask (SIGUSR1) | sigmask (SIGUSR2));
```

WARNINGS

Sigsetmask should not be used in conjunction with the facilities described under *sigset*(2V).

AUTHOR

Sigsetmask was developed by the University of California, Berkeley.

SEE ALSO

kill(2), sigblock(2), sigpause(2), sigprocmask(2), sigvector(2).

NAME

sigspace – assure sufficient signal stack space

SYNOPSIS

#include <sys/types.h>

size_t sigspace(stacksize)
size_t stacksize;

DESCRIPTION

Sigspace requests additional stack space that is guaranteed to be available for processing signals received by the calling process.

If the value of *stacksize* is positive, it specifies the size of a space, in bytes, which the system guarantees to be available when processing a signal. If the value of *stacksize* is zero, any guarantee of space is removed. If the value is negative, the guarantee is left unchanged; this can be used to interrogate the current guaranteed value.

When a signal's action indicates that its handler should use the guaranteed space (specified with a *sigaction*(2), *sigvector*(2) or *sigvec* (on *bsdproc*(2)) call), the system checks to see if the process is currently using that space. If the process is not currently using that space, the system arranges for that space to be available for the duration of the signal handler's execution. If that space has already been made available (due to a previous signal) no change is made. The normal stack discipline is resumed when the signal handler first using the guaranteed space is exited.

The guaranteed space is inherited by child processes resulting from a successful *fork*(2) system call, but the guarantee of space is removed after any *exec*(2) system call.

The guaranteed space cannot be increased in size automatically, as is done for the normal stack. If the stack overflows the guaranteed space, the resulting behavior of the process is undefined.

Guaranteeing space for a stack can interfere with other memory allocation routines, in an implementation-dependent manner.

During normal execution of the program the system checks for possible overflow of the stack. Guaranteeing space might cause the space available for normal execution to be reduced.

Leaving the context of a service routine abnormally, such as by *longjmp* on *setjmp*(3C), removes the guarantee that the ordinary execution of the program will not extend into the guaranteed space. It might also cause the program to lose forever its ability to automatically increase the stack size, causing the program to be limited to the guaranteed space.

RETURN VALUE

Upon successful completion, the size of the former guaranteed space is returned. Otherwise, a value of −1 is returned and **errno** is set to indicate the error.

ERRORS

Sigspace fails and the guaranteed amount of space remains unchanged if the following occurs:

[ENOMEM] The requested space cannot be guaranteed either because of hardware limitations or because some software-imposed limit would be exceeded.

WARNINGS

The guaranteed space is allocated using *malloc*(3C). This use might interfere with other heap management mechanisms.

Methods for calculating the required size are not well developed.

Sigspace should not be used in conjunction with the facilities described under *sigset*(2V).

Sigspace should not be used in conjunction with *sigstack*(2).

DEPENDENCIES

Series 300

The kernel overhead taken in the reserved space is 608 bytes on Series 300. This overhead must be included in the requested amount. These values are subject to change in future releases.

AUTHOR

Sigspace was developed by HP.

SEE ALSO

sigaction(2), sigstack(2), sigvector(2), malloc(3C), setjmp(3C).

NAME

sigstack – set and/or get signal stack context

SYNOPSIS

#include <signal.h>

int sigstack (ss, oss)
struct sigstack *ss, *oss;

DESCRIPTION

Sigstack allows the calling process to indicate to the system an area of its address space to be used for processing signals received by the process.

The correct use of *sigstack*(2) is hardware dependent, and therefore is not portable between different implementations of HP-UX (see DEPENDENCIES below). *Sigspace*(2) is portable between different implementations of HP-UX and it should be used when the application does not need to know where the signal stack is located. *Sigstack* is provided for compatability with other systems that provide this functionality. Users should note that there is no guarantee that functionality similar to this is even possible on some architectures.

If the value of the *ss* argument is not a null pointer, it is assumed to point to a **struct sigstack** structure, which includes the following members:

int ss_onstack;	non-zero when signal stack is in use
void *ss_sp;	signal stack pointer

The value of the **ss_onstack** member indicates whether the process wants the system to use a signal stack when delivering signals; the value of the **ss_sp** member indicates the desired location (see DEPENDENCIES) of the signal stack area in the process's virtual address space.

If the *ss* argument is a null pointer, the current signal stack context is not changed.

If the *oss* argument is not a null pointer, it should point to a variable of type **struct sigstack**; the current signal stack context is returned in that variable. The value stored in the **ss_onstack** member tells whether the process is currently using a signal stack, and if so, the value stored in the **ss_sp** member is the current stack pointer for the stack in use.

If the *oss* argument is a null pointer, the current signal stack context is not returned.

When a signal's action indicates its handler should execute on the signal stack (specified by calling *sigaction*(2), *sigvector*(2), or *sigvec* (on *bsdproc*(2))), the system checks to see if the process is currently executing on that stack. If the process is not currently executing on the signal stack, the system arranges a switch to the signal stack for the duration of the signal handler's execution.

The signal stack context is inherited by child processes resulting from a successful *fork*(2) system call, but the context is removed after an *exec*(2) system call.

RETURN VALUE

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

ERRORS

Sigstack fails and the signal stack context remains unchanged if the following is true:

[EFAULT] Either of *ss* or *oss* is not a null pointer and points outside the allocated address space of the process. The reliable detection of this error is implementation dependent.

WARNINGS

Sigstack(2) should not be used in conjunction with *sigspace*(2).

User-defined signal stacks do not grow automatically, as does the normal process stack. If a signal stack overflows, the resulting behavior of the process is undefined.

Methods for calculating the required stack size are not well developed.

Leaving the context of a service routine abnormally, such as by *longjmp* (on *setjmp*(3C)), might remove the guarantee that the ordinary execution of the program does not extend into the guaranteed space. It might also cause the program to lose forever its ability to automatically increase the stack size, causing the program to be limited to the guaranteed space.

DEPENDENCIES

Series 300

Stack addresses grow from high addresses to low addresses; therefore the signal stack address provided to *sigstack*(2) should point to the end of the space to be used for the signal stack. This address should be aligned to a four-byte boundary.

Series 800

Stack addresses grow from low addresses to high addresses; therefore the signal stack address provided to *sigstack*(2) should point to the beginning of the space to be used for the signal stack. This address should be aligned to an eight-byte boundary.

AUTHOR

Sigstack was developed by HP and the University of California, Berkeley.

SEE ALSO

sigspace(2), setjmp(3C).

NAME

sigsuspend – wait for a signal

SYNOPSIS

#include <signal.h>

int sigsuspend (sigmask)
sigset_t *sigmask ;

DESCRIPTION

Sigsuspend replaces the process's current signal mask with the set of signals pointed to by *sigmask,* and then suspends the process until delivery of a signal that either executes a signal handler or terminates the process.

If the signal terminates the process, *sigsuspend* never returns. If the signal executes a signal handler, *sigsuspend* returns after the signal handler returns, and restores the signal mask to the set that existed prior to the *sigsuspend* call.

It is impossible to block the **SIGKILL** or **SIGSTOP** signal. This is enforced by the system without causing an error to be indicated.

RETURN VALUE

Since *sigsuspend* suspends a process indefinitely, there is no successful completion return value. If a return occurs, a value of **−1** is returned and **errno** is set to indicate the error.

ERRORS

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

[EINTR] *Sigsuspend* was interrupted by receipt of a signal.

[EFAULT] *Sigmask* points to an invalid address. The reliable detection of this error is implementation dependent.

AUTHOR

Sigsuspend was derived from the IEEE Standard POSIX 1003.1-1988.

SEE ALSO

sigaction(2), sigpending(2), sigprocmask(2), sigsetops(3C), signal(5).

STANDARDS CONFORMANCE

sigsuspend: XPG3, POSIX.1, FIPS 151-1

NAME

sigvector – software signal facilities

SYNOPSIS

#include <signal.h>

sigvector(sig, vec, ovec**)**
int sig;
struct sigvec *vec, *ovec;

DESCRIPTION

The system defines a set of signals that can be delivered to a process. The set of signals is defined in *signal*(5), along with the meaning and side effects of each signal. This manual page, along with those for *sigblock*(2), *sigsetmask*(2), *sigpause*(2), and *sigspace*(2), defines an alternate mechanism for handling these signals that assures the delivery of signals and integrity of signal handling procedures. The facilities described here should not be used in the same program as *signal*(2).

With the *sigvector* interface, signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process can specify a handler function to be invoked when a signal is delivered, or specify that a signal should be blocked or ignored. A process can also specify that a default action should be taken by the system when a signal occurs. It is possible to ensure a minimum amount of stack space for processing signals using the *sigspace*(2) call.

All signals have the same priority. Signal routines execute with the signal that causes their invocation to be blocked, although other signals can yet occur. A global signal mask defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent (normally **0**). It can be changed with a *sigblock*(2), *sigsetmask*(2), or *sigpause*(2) call, or when a signal is delivered to the process.

A signal mask is represented as a **long**, with one bit representing each signal being blocked. The following macro defined in **<signal.h>** is used to convert a signal number to its corresponding bit in the mask:

```
#define    sigmask(signo)    (1L << (signo-1))
```

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently blocked by the process, it is delivered to the process. When a signal is delivered, the current state of the process is saved, a new signal mask is calculated (as described below), and the signal handler is invoked. The call to the handler is arranged so that if the signal handling routine returns normally, the process resumes execution in the same context as before the signal's delivery. If the process wishes to resume in a different context, it must arrange to restore the previous context itself.

When a signal is delivered to a process, a new signal mask is installed for the duration of the process' signal handler (or until a *sigblock*(2) or *sigsetmask*(2) call is made). This mask is formed by taking the current signal mask, computing the bitwise inclusive OR with the value of *vec.sv_mask* (see below) from the most recent call to *sigvector* for the signal to be delivered, and, unless the SV_RESETHAND flag is set (see below), setting the bit corresponding to the signal being delivered. When the user's signal handler returns normally, the original mask is restored.

Sigvector assigns a handler for the signal specified by *sig*. *Vec* and *ovec* are pointers to *sigvec* structures that include the following elements:

```
void    (*sv_handler)();
long    sv_mask;
long    sv_flags;
```

If *vec* is non-zero, it specifies a handler routine (*sv_handler*), a mask (*sv_mask*) that the system should use when delivering the specified signal, and a set of flags (*sv_flags*) that modify the delivery of the signal. If *ovec* is non-zero, the previous handling information for the signal is returned to the user. If *vec* is zero, signal handling is unchanged: thus, the call can be used to enquire about the current handling of a given signal. If *vec* and *ovec* point to the same structure, the value of *vec* is read prior to being overwritten.

The *sv_flags* field can be used to modify the receipt of signals. The following flag bits are defined:

SV_ONSTACK	Use the *sigspace* allocated space
SV_BSDSIG	Use the Berkeley signal semantics
SV_RESETHAND	Use the semantics of *signal*(2)

If SV_ONSTACK is set, the system uses, or permits the use of, the space reserved for signal processing in the *sigspace*(2) system call.

If SV_BSDSIG is set, the signal is given the Berkeley semantics. The following signal is affected by this flag:

SIGCLD — In addition to being sent when a child process dies, the signal is also sent when any child's status changes from running to stopped. This would normally be used by a program such as *csh*(1) when maintaining process groups under Berkeley job control.

If SV_RESETHAND is set, the signal handler will be installed with the same semantics as a handler installed with *signal*(2). This affects the signal mask set up during the signal handler (see above) and whether the handler is reset after a signal is caught (see below).

If SV_RESETHAND is not set, once a signal handler is installed, it remains installed until another *sigvector* call is made or an *exec*(2) system call is performed. If SV_RESETHAND is set and the signal is not one of those marked "not reset when caught" under *signal*(5), the default action is reinstated when the signal is caught, prior to entering the signal-catching function. The "not reset when caught" distinction is not significant when *sigvector* is called and SV_RESETHAND is not set.

The default action for a signal can be reinstated by setting *sv_handler* to SIG_DFL; this default usually results in termination of the process. If *sv_handler* is SIG_IGN the signal is usually subsequently ignored, and pending instances of the signal are discarded. The exact meaning of SIG_DFL and SIG_IGN for each signal is discussed in *signal*(5).

Certain system calls can be interrupted by a signal; all other system calls complete before the signal is serviced. The *scp* pointer described in *signal*(5) is never null if *sigvector* is supported. *Scp* points to a machine-dependent *sigcontext* structure. All implementations of this structure include the fields:

```
int     sc_syscall;
char    sc_syscall_action;
```

The value SYS_NOTSYSCALL for the *sc_syscall* field indicates that the signal is not interrupting a system call; any other value indicates which system call it is interrupting.

If a signal that is being caught occurs during a system call that can be interrupted, the signal handler is immediately invoked. If the signal handler exits normally, the value of the *sc_syscall_action* field is inspected; if the value is SIG_RETURN, the system call is aborted and the interrupted program continues past the call. The result of the interrupted call is -1 and **errno** is set to EINTR. If the value of the *sc_syscall_action* field is SIG_RESTART, the call is restarted. A call is restarted if, in the case of a *read*(2) or *write*(2) system call, it had transferred no data. If some data had been transferred, the operation is considered to have completed with a partial transfer, and the *sc_syscall* value is SYS_NOTSYSCALL. Other values are undefined and reserved for future use.

Exiting the handler abnormally (such as with *longjmp* on *setjmp*(3C)) aborts the call, leaving the user responsible for the context of further execution. The value of *scp->sc_syscall_action* is ignored when the value of *scp->sc_syscall* is SYS_NOTSYSCALL. *Scp->sc_syscall_action* is always initialized to SIG_RETURN before invocation of a signal handler. When an system call that can be interrupted is interrupted by multiple signals, if any signal handler returns a value of SIG_RETURN in *scp->sc_syscall_action*, all subsequent signal handlers are passed a value of SYS_NOTSYSCALL in *scp->sc_syscall*.

Note that calls to *read*(2), *write*(2) or *ioctl*(2) on fast devices (disks) cannot be interrupted, but I/O to a slow device (teletype) can be interrupted. Other system calls, such as those used for networking, also can be interrupted on some implementations. In these cases additional values can be specified for *scp->sc_syscall*. Programs that look at the values of *scp->sc_syscall* always should compare them to these symbolic constants; the numerical values represented by these constants might vary among implementations. System calls that can be interrupted and their corresponding values for *scp->sc_syscall* are listed below:

Call	sc_syscall value
------	----------------
read (slow devices)	SYS_READ
readv (slow devices)	SYS_READV
write (slow devices)	SYS_WRITE
writev (slow devices)	SYS_WRITEV
open (slow devices)	SYS_OPEN
ioctl (slow requests)	SYS_IOCTL
wait	SYS_WAIT
select	SYS_SELECT
pause	SYS_PAUSE
sigpause	SYS_SIGPAUSE
semop	SYS_SEMOP
msgsnd	SYS_MSGSND
msgrcv	SYS_MSGRCV

These system calls are not defined if the preprocessor macro _XPG2 is defined when <**signal.h**> is included. This is because *Issue 2 of the X/Open Portability Guide* specifies a different meaning for the symbol SYS_OPEN (see *limits*(5)).

After a *fork*(2) or *vfork*(2) system call, the child inherits all signals, the signal mask, and the reserved signal stack space.

Exec(2) resets all caught signals to the default action; ignored signals remain ignored, the signal mask remains unchanged, and the reserved signal stack space is released.

The mask specified in *vec* is not allowed to block signals that cannot be ignored, as defined in *signal*(5). This is enforced silently by the system.

If *sigvector* is called to catch SIGCLD in a process that currently has terminated (zombie) children, a SIGCLD signal is delivered to the calling process immediately, or as soon as SIGCLD is unblocked if it is currently blocked. Thus, in a process that spawns multiple children and catches SIGCLD, it is sometimes advisable to reinstall the handler for SIGCLD after each invocation in case there are multiple zombies present. This is true even though the handling of the signal is not reset by the system, as with *signal*(2), because deaths of multiple processes while SIGCLD is blocked in the handler result in delivery of only a single signal. Note that the function must reinstall itself after it has called *wait*(2) or *wait3*(2). Otherwise the presence of the child that caused the original signal always causes another signal to be delivered.

RETURN VALUE

A **0** value indicates that the call succeeded. A **−1** return value indicates an error occurred and **errno** is set to indicate the reason.

ERRORS

Sigvector fails and no new signal handler is installed if one of the following occurs:

[EFAULT] Either *vec* or *ovec* points to memory that is not a valid part of the process address space. The reliable detection of this error is implementation dependent.

[EINVAL] *Sig* is not a valid signal number.

[EINVAL] An attempt is made to ignore or supply a handler for a signal that cannot be caught or ignored; see *signal*(5).

WARNINGS

Restarting a *select*(2) call can sometimes cause unexpected results. If the *select* call has a timeout specified, the timeout is restarted with the call, ignoring any portion that had elapsed prior to interruption by the signal. Normally this simply extends the timeout and is not a problem. However, if a handler repeatedly catches signals and the timeout specified to select is longer than the time between those signals, restarting the *select* call effectively renders the timeout infinite.

Sigvector should not be used in conjunction with the facilities described under *sigset*(2V).

AUTHOR

Sigvector was developed by HP and the University of California, Berkeley.

SEE ALSO

kill(1), kill(2), ptrace(2), sigblock(2), signal(2), sigpause(2), sigsetmask(2), sigspace(2), setjmp(3C), signal(5), termio(7).

NAME

stat, lstat, fstat – get file status

SYNOPSIS

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

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

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

int fstat (fildes, buf)
int fildes;
struct stat *buf;

DESCRIPTION

Stat obtains information about the named file.

Path points to a path name naming a file. Read, write, or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable.

Similarly, *fstat* obtains information about an open file known by the file descriptor *fildes*, obtained from a successful *open*(2), *creat*(2), *dup*(2), *fcntl*(2), or *pipe*(2) system call.

Lstat is similar to *stat* except when the named file is a symbolic link, in which case *lstat* returns the information about the link, while *stat* returns information about the file to which the link points.

Buf is a pointer to a **stat** structure into which information is placed concerning the file.

The contents of the structure **stat** pointed to by *buf* include the following members. Note that there is no necessary correlation between the placement in this list and the order in the structure.

```
dev_t   st_dev;         /* ID of device containing a */
                        /* directory entry for this file */
ino_t   st_ino;         /* Inode number */
ushort  st_fstype;      /* Type of filesystem this file  */
                        /* is in; see vfsmount(OS) */
ushort  st_mode;        /* File type, attributes, and */
                        /* access control summary */
ushort  st_basemode     /* Permission bits (see chmod(1)) */
ushort  st_nlink;       /* Number of links */
uid_t   st_uid;         /* User ID of file owner */
gid_t   st_gid;         /* Group ID of file group */
dev_t   st_rdev;        /* Device ID; this entry defined */
                        /* only for char or blk spec files */
off_t   st_size;        /* File size (bytes) */
time_t  st_atime;       /* Time of last access */
time_t  st_mtime;       /* Last modification time */
time_t  st_ctime;       /* Last file status change time */
                        /* Measured in secs since */
                        /* 00:00:00 GMT, Jan 1, 1970 */
uint    st_acl:1;       /* Set if the file has optional */
                        /* access control list entries */
```

st_atime Field indicating when file data was last accessed. Changed by the following system calls: *creat*(2), *mknod*(2), *pipe*(2), *read*(2), *readv* (on *read*(2)), and *utime*(2).

st_mtime Field indicating when data was last modified. Changed by the following system calls: *creat*(2), *truncate*(2), *ftruncate* (on *truncate*(2)), *mknod*(2), *pipe*(2), *prealloc*(2), *utime*(2), *write*(2), and *writev* (on *write*(2)). Also changed by *close*(2) when the reference count reaches zero on a named pipe (FIFO special) file that contains data.

st_ctime Field indicating when file status was last changed. Changed by the following system calls: *chmod*(2), *chown*(2), *creat*(2), *fchmod*(2), *fchown*(2), *truncate*(2), *ftruncate* (on *truncate*(2)), *link*(2), *mknod*(2), *pipe*(2), *prealloc*(2), *rename*(2), *setacl*(2), *unlink*(2), *utime*(2), *write*(2), and *writev* (on *write*(2)).

The *touch*(1) command can be used to explicitly control the times of a file.

st_mode The value returned in this field is the bitwise inclusive OR of a value indicating the file's type, attribute bits, and a value summarizing its access permission. See *mknod*(2).

For ordinary users, the least significant nine bits consist of the file's permission bits modified to reflect the access granted or denied to the caller by optional entries in the file's access control list.

For superusers, the least significant nine bits are the file's access permission bits. In addition, the S_IXUSR (execute by owner) mode bit is set if the following conditions are met:

-- the file is a regular file,
-- no permission execute bits are set, and
-- an execute bit is set in one or more of the file's optional access control list entries.

The write bit is not cleared for a file on a read-only file system or a shared-text program file that is being executed. However, *getaccess*(2) clears this bit under these conditions.

NETWORKING FEATURES

RFA

The contents of the structure **stat** pointed to by *buf* also include the following members:

```
uint    st_remote:1;    /* Set if file is remote */
dev_t   st_netdev;      /* ID of device containing */
                        /* network special file */
ino_t   st_netino;      /* Inode number of network special file */
```

st_remote Field indicating whether the file is on a remote node. A zero value indicates that the file is on the local node; non-zero indicates that the file is on a remote node, and accessed through remote file access (RFA). Not all HP-UX systems support RFA; **st_remote** is always zero on non-RFA supported systems.

st_netdev, st_netino All remote file access takes place through a special file in the local file system known as a network special file. Each network special file identifies a particular remote node. When **st_remote** is non-zero, **st_netdev** and **st_netino** identify the appropriate network special file;

otherwise these fields are zero.

RETURN VALUE

Upon successful completion, **0** is returned. Otherwise, **−1** is returned and **errno** is set to indicate the error.

ERRORS

Stat or *lstat* fails if any of the following is true:

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] The named file does not exist (for example, *path* is null or a component of *path* does not exist).

[EACCES] Search permission is denied for a component of the path prefix.

[EFAULT] *Buf* or *path* points to an invalid address. The reliable detection of this error is implementation dependent.

[ELOOP] Too many symbolic links were encountered in translating the path name.

[ENAMETOOLONG]
The length of the specified path name exceeds PATH_MAX bytes, or the length of a component of the path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.

Fstat fails if any of the following is true:

[EBADF] *Fildes* is not a valid open file descriptor.

[EFAULT] *Buf* points to an invalid address. The reliable detection of this error is implementation dependent.

DEPENDENCIES

HP Clustered Environment

The contents of the **stat** structure include the following additional members:

```
cnode_t st_cnode;   /* Cnode ID of machine */
                    /* where the inode lives */
cnode_t st_rcnode   /* Cnode ID where this */
                    /* device file can be used */
dev_t   st_realdev; /* Real device number of device */
                    /* containing the inode for this file */
```

st_dev The ID number for the volume on which the inode exists. This number may or may not be the device number for the device containing the volume. Device numbers are not unique throughout a cluster, but the value of *st_dev* is guaranteed to be unique among all volumes currently mounted in the file system. The device number for the volume can always be found in the field **st_realdev**, which together with **st_cnode** fully specifies the device containing the volume.

CD-ROM

The **st_uid** and **st_gid** fields are set to −1 if they are not specified on the disk for a given file.

RFA and NFS

The *st_basemode* and *st_acl* fields are zero on files accessed remotely.

AUTHOR

Stat and *fstat* were developed by AT&T. *Lstat* was developed by the University of California, Berkeley.

SEE ALSO

touch(1), chmod(2), chown(2), creat(2), link(2), mknod(2), pipe(2), read(2), rename(2), setacl(2), time(2), truncate(2), unlink(2), utime(2), write(2), acl(5), stat(5).

STANDARDS CONFORMANCE

stat: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

fstat: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

statfs, fstatfs − get file system statistics

SYNOPSIS

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

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

int fstatfs(fildes, buf)
int fildes;
struct statfs *buf;
```

DESCRIPTION

Statfs returns information about a mounted file system. *Path* is the path name of any file within the mounted file system.

Buf is a pointer to a **statfs** structure into which information is placed concerning the file system. The contents of the structure pointed to by *buf* include the following members:

```
long    f_bavail;   /* free blocks available to non-superuser */
long    f_bfree;    /* free blocks */
long    f_blocks;   /* total blocks in file system */
long    f_bsize;    /* fundamental file system block size in bytes */
long    f_ffree;    /* free file nodes in file system */
long    f_files;    /* total file nodes in file system */
long    f_type;     /* type of info, zero for now */
fsid_t  f_fsid;     /* file system ID */
```

A *file node* is a structure in the file system hierarchy that describes a file. For mounted HP-UX volumes, *file node* is an HP-UX inode. For other types of mounts, *file node* is defined by the system embodying the file pointed to by *path*.

Fields that are undefined for a particular file system are set to −1.

Fstatfs returns the same information about an open file referred to by file descriptor *fildes*.

RETURN VALUE

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

ERRORS

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

[EACCES]	Search permission is denied for a component of the path prefix.
[EFAULT]	*Buf* or *path* points to an invalid address.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[ELOOP]	Too many symbolic links are encountered in translating the path name.
[ENAMETOOLONG]	A component of *path* exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect, or *path* exceeds PATH_MAX bytes.
[ENOENT]	The named file does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.

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

[EBADF]
Fildes is not a valid open file descriptor.

[EFAULT]
Buf points to an invalid address.

[EIO] An I/O error occurs while reading from or writing to the file system.

AUTHOR

Statfs and *fstatfs* were developed by Sun Microsystems, Inc.

SEE ALSO

df(1M), stat(2), ustat(2).

NAME

stime – set time and date

SYNOPSIS

int stime (tp)
long *tp;

DESCRIPTION

Stime sets the system's idea of the time and date. *Tp* points to the value of time as measured in seconds from 00:00:00 GMT January 1, 1970.

RETURN VALUE

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

ERRORS

[EPERM] *Stime* will fail if the effective user ID of the calling process is not super-user.

DEPENDENCIES

HP Clustered Environment

On systems that are members of a cluster, setting the time sets the time and date on all systems in the cluster.

SEE ALSO

date(1), gettimeofday(2), time(2).

STANDARDS CONFORMANCE

stime: SVID2, XPG2

NAME

stty, gtty – control device

SYNOPSIS

#include <sgtty.h>

stty(fildes,argp)
int fildes;
struct sgttyb *argp;

gtty(fildes,argp)
int fildes;
struct sgttyb *argp;

REMARKS

These system calls are preserved for backward compatibility with Bell Version 6. They provide as close an approximation as possible to the old Version 6 functions. All new code should use the TCSETA/TCGETA *ioctl* calls described in *termio*(7).

DESCRIPTION

For certain status setting and status inquiries about terminal devices, the functions *stty* and *gtty* are equivalent to

```
ioctl(fildes, TIOCSETP, argp)
ioctl(fildes, TIOCGETP, argp)
```

respectively; see *termio*(7).

RETURNS

Zero is returned if the call was successful; -1 if the file descriptor does not refer to the kind of file for which it was intended.

SEE ALSO

stty(1), exec(2), sttyV6(7), tty(7), termio(7).

NAME

swapon – add swap space for interleaved paging/swapping

SYNOPSIS

swapon (special)|(directory, [min, limit, reserve, priority])
char * special, directory;
int min, limit, reserve, priority;

DESCRIPTION

Swapon makes the block device *special* available to the system for allocation for paging and swapping. The names of potentially available devices are known to the system and defined at system configuration time. See the appropriate system administrator's manual for information on how the size of the swap area is calculated.

Swapon can also make the blocks on the file system specified by *directory* available for paging and swapping.

The *min limit reserve* and *priority* parameters default to zero and only have meaning if the first parameter passed to swapon is a *directory.*

min indicates the number of file system blocks to take from the file system at the time that *swapon()* is called.

limit indicates the maximum number of file system blocks the swap system is allowed to take from the file system.

reserve indicates the number of file system blocks that are saved for file system use only.

priority indicates the order in which space is taken from the file systems used for swap. Space is taken from the lower priority systems first.

File systems used for swapping do not have to be configured into the system.

Swapon may be invoked only by the super-user.

ERRORS

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

[ENOTBLK] *Special* is not the name of a block special file.

[ENXIO] The device associated with *special* could not be opened.

[EBUSY] The device associated with *special* is already in use.

[ENODEV] The device associated with *special* does not exist.

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

[ELOOP] Too many symbolic links were encountered in translating the path name.

[ENOENT] The swap space requested is not a block special file or a directory

[ENOSPC] There is is not enough available space on the file system specified to allocate the amount requested in the *min* parameter.

[EINVAL] The system imposed limit on the number of swap file entries has been reached.

[ENAMETOOLONG]
The length of the specified path name exceeds PATH_MAX bytes, or the length of a component of the path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.

WARNINGS

There is no way to stop swapping on a disk so that the pack may be dismounted.

The system will allocate no less than the amount specified in "min", however, to make the most efficient use of space, more than the amount requested might be taken from the file system. The actual amount taken will not exceed the number of file system blocks indicated in "reserve".

Swapping to the file system can be slower than swapping to a device.

AUTHOR

Swapon was developed by the University of California, Berkeley.

SEE ALSO

swapon(1M).

NAME

symlink – make symbolic link to a file

SYNOPSIS

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

DESCRIPTION

Symlink creates a file *name2*, which is a symbolic link to *name1*. Either name may be an arbitrary path name. The files need not be on the same file system.

RETURN VALUE

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

ERRORS

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

[ENOTDIR] A component of the *name2* prefix is not a directory.

[ENAMETOOLONG] A component of either path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect, or the entire length of either path name exceeds PATH_MAX bytes.

[ENOENT] The named file does not exist.

[EACCES] A component of the *name2* path prefix denies search permission.

[ELOOP] Too many symbolic links were encountered in translating the path name.

[EEXIST] *Name2* already exists.

[EIO] An I/O error occurred while making the directory entry for *name2*, allocating the inode for *name2*, or writing out the link contents of *name2*.

[EROFS] The file *name2* resides on a read-only file system.

[ENOSPC] The directory in which the entry for the new symbolic link is being placed cannot be extended because there is no space left on the file system containing the directory.

[ENOSPC] The new symbolic link cannot be created because there there is no space left on the file system that will contain the symbolic link.

[ENOSPC] There are no free inodes on the file system on which the symbolic link is being created.

[EIO] An I/O error occurred while making the directory entry or allocating the inode.

[EFAULT] *Name1* or *name2* points outside the process' allocated address space. The reliable detection of this error is implementation dependent.

AUTHOR

Symlink was developed by the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

symlink(4), readlink(2), link(2), cp(1), unlink(2).

NAME

sync, lsync – update super-block

SYNOPSIS

void sync ( )

void lsync ( )

DESCRIPTION

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

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

The writing, although scheduled, is not necessarily complete upon return from *sync*.

In some HP-UX systems, *sync* may be reduced to a no-op. This is permissible on a system which does not cache buffers, or in a system that in some way ensures that the disks are always in a consistent state.

In the HP Clustered Environment, *sync* causes updates of all file systems in the cluster to be written out, while *lsync* performs only a local *sync*; that is, local buffers are flushed to disk and to remote nodes of the cluster, but remote nodes do not flush their own pages.

AUTHOR

Sync was developed by HP and AT&T Bell Laboratories. *Lsync* was developed by HP.

SEE ALSO

sync(1M).

STANDARDS CONFORMANCE

sync: SVID2, XPG2

NAME

sysconf – get configurable system variables

SYNOPSIS

#include <unistd.h>

long sysconf(*name*)
int *name*;

DESCRIPTION

The *sysconf* function enables applications to determine the current value of a configurable limit or variable.

The *name* argument represents the system variable being queried.

The following table lists the configuration variables whose values can be determined by calling *sysconf*, and for each variable, the associated value of the *name* argument and function return:

Variable	Value of *name*	Value Returned
ARG_MAX	_SC_ARG_MAX	Maximum length of the arguments for *exec*(2) in bytes, including environment data
CHILD_MAX	_SC_CHILD_MAX	Maximum number of simultaneous processes per user ID
CLK_TCK	_SC_CLK_TCK	Number of clock intervals per second
NGROUPS_MAX	_SC_NGROUPS_MAX	Maximum number of simultaneous supplementary group IDs per process
OPEN_MAX	_SC_OPEN_MAX	Maximum number of files that one process can have open at one time
PASS_MAX	_SC_PASS_MAX	Maximum number of significant characters in a password
_POSIX_JOB_CONTROL	_SC_JOB_CONTROL	Non-zero if the system supports POSIX job control; **−1** otherwise
_POSIX_SAVED_IDS	_SC_SAVED_IDS	Non-zero if each process has a saved set-user-ID and a saved set-group-ID; **−1** otherwise

_POSIX_VERSION	**_SC_VERSION**	Version of the POSIX Standard (such as 198808L) to which the system conforms. This value will change with each published revision of the standard, to indicate the year (first four digits) and month (next two digits) that the standard was approved by the IEEE Standards Board. If the system does not conform to any version, **−1** is returned.

The variables in the table are defined as constants in <**limits.h**> (see *limits*(5)). The associated values of the *name* argument are defined in <**unistd.h**>.

RETURN VALUE

If the value of *name* is not valid, *sysconf* returns **−1** and sets **errno** to indicate the error. If the variable corresponding to *name* is not defined, *sysconf* returns −**1**; however, **errno** will not be changed.

Upon any successful completion, *sysconf* returns the value of the named variable, as described above. These values do not change during the lifetime of the calling process.

ERRORS

Sysconf fails if the following condition is true:

[EINVAL] The value of *name* is not valid.

EXAMPLES

The following example determines the number of times the system clock ticks each second:

```
#include <unistd.h>
long hz;
...
hz = sysconf(_SC_CLK_TCK);
```

AUTHOR

Sysconf was developed by HP and POSIX.

SEE ALSO

pathconf(2), unistd(5), limits(5).

STANDARDS CONFORMANCE

sysconf: XPG3, POSIX.1, FIPS 151-1

NAME

time – get time

SYNOPSIS

#include <time.h>

time_t time (tloc)
time_t *tloc;

DESCRIPTION

Time returns the value of time in seconds since the Epoch.

If *tloc* is not a null pointer, the return value is also assigned to the object to which it points.

ERRORS

[EFAULT] *Time* will fail if *tloc* points to an illegal address. The reliable detection of this error will be implementation dependent.

RETURN VALUE

Upon successful completion, *time* returns the value of time. Otherwise, a value of (time_t)−1 is returned and **errno** is set to indicate the error.

SEE ALSO

date(1), gettimeofday(2), stime(2), ctime(3C), strftime(3C).

STANDARDS CONFORMANCE

time: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

times – get process and child process times

SYNOPSIS

#include <sys/times.h>

clock_t times (buffer)
struct tms *buffer;

DESCRIPTION

Times fills the structure pointed to by *buffer* with time-accounting information. The structure defined in **sys/times.h** is as follows:

```
struct tms {
    clock_t tms_utime;    /* user time */
    clock_t tms_stime;    /* system time */
    clock_t tms_cutime;   /* user time, children */
    clock_t tms_cstime;   /* system time, children */
};
```

This information comes from the calling process and each of its terminated child processes for which it has executed a *wait*, *wait3*, or *waitpid*. The times are in units of 1 / CLK_TCK seconds, where CLK_TCK is processor dependent The value of CLK_TCK can be queried using the *sysconf*(2) call.

Tms_utime is the CPU time used while executing instructions in the user space of the calling process.

Tms_stime is the CPU time used by the system on behalf of the calling process.

Tms_cutime is the sum of the *tms_utime*s and *tms_cutime*s of the child processes.

Tms_cstime is the sum of the *tms_stime*s and *tms_cstime*s of the child processes.

ERRORS

[EFAULT] *Times* will fail if *buffer* points to an illegal address. The reliable detection of this error will be implementation dependent.

RETURN VALUE

Upon successful completion, *times* returns the elapsed real time, in units of 1 / CLK_TCK of a second, since an arbitrary point in the past (e.g., system start-up time). This point does not change from one invocation of *times* to another. If *times* fails, a –1 is returned and **errno** is set to indicate the error.

SEE ALSO

time(1), gettimeofday(2), exec(2), fork(2), sysconf(2), time(2), wait(2).

BUGS

Not all CPU time expended by system processes on behalf of a user process is counted in the system CPU time for that process.

STANDARDS CONFORMANCE

times: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

truncate, ftruncate – truncate a file to a specified length

SYNOPSIS

truncate(path, length)
char *path;
unsigned long length;

ftruncate(fd, length)
int fd;
unsigned long length;

DESCRIPTION

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

RETURN VALUES

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

ERRORS

Truncate succeeds unless:

[ENOTDIR] A component of the path prefix of *path* is not a directory.

[EACCES] A component of the path prefix denies search permission.

[EACCES] Write permission is denied on the file.

[EISDIR] The named file is a directory.

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

[ETXTBSY] The file is a pure procedure (shared text) file that is being executed.

[EFAULT] *Path* points outside the process's allocated address space. The reliable detection of this error will be implementation dependent.

[ELOOP] Too many symbolic links were encountered in translating the path name.

[ENAMETOOLONG]
The length of the specified path name exceeds PATH_MAX bytes, or the length of a component of the path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.

Ftruncate succeeds unless:

[EBADF] The *fd* is not a valid descriptor.

[EINVAL] The *fd* references a file that was opened without write permission.

AUTHOR

Truncate was developed by the University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer Science.

SEE ALSO

open(2).

NAME

ulimit – get and set user limits

SYNOPSIS

#include <ulimit.h>

long ulimit (cmd, ...)
int cmd;

DESCRIPTION

This function provides for control over process limits. The *cmd* values available are:

UL_GETFSIZE Get the file size limit of the process. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.

UL_SETFSIZE Set the file size limit of the process to the value of the optional second argument. Any process may decrease this limit, but only a process with an effective user ID of super-user may increase the limit. Note that the limit must be specified in units of 512-byte blocks.

UL_GETMAXBRK Get the maximum possible break value. See *brk*(2). Depending on system resources such as swap space, this maximum may not be attainable at a given time.

ERRORS

Ulimit will fail if one or more of the following conditions is true.

[EINVAL] *cmd* is not in the correct range.

[EPERM] *Ulimit* will fail and the limit will be unchanged if a process with an effective user ID other than super-user attempts to increase its file size limit.

RETURN VALUE

Upon successful completion, a non-negative value is returned. Errors return a -1, with **errno** set appropriately.

SEE ALSO

brk(2), write(2).

STANDARDS CONFORMANCE

ulimit: SVID2, XPG2, XPG3

NAME

umask – set and get file creation mask

SYNOPSIS

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

mode_t umask (cmask)
mode_t cmask;

DESCRIPTION

Umask sets the process's file mode creation mask to *cmask* and returns the previous value of the mask. Only the file access permission bits of the masks are used.

The bits set in *cmask* specify which permission bits to turn off in the mode of the created file, and should be specified using the symbolic values defined in *stat*(5).

EXAMPLES

The following creates a file named **path** in the current directory with permissions S_IRWXU|S_IRGRP|S_IXGRP, so that the file can be written only by its owner, and can be read or executed only by the owner or processes with group permission, even though group write permission and all permissions for others are passed in to *creat*.

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

int fildes;

(void) umask(S_IWGRP|S_IRWXO);
fildes = creat("path", S_IRWXU|S_IRWXG|S_IRWXO);
```

RETURN VALUE

The previous value of the file mode creation mask is returned.

SEE ALSO

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

STANDARDS CONFORMANCE

umask: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

umount – unmount a file system

SYNOPSIS

int umount (name)
char *name;

DESCRIPTION

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

Umount can also request that a file system mounted previously on the directory identified by *name* be unmounted. After unmounting the file system, *name* reverts to its ordinary interpretation.

Umount can be invoked only by the superuser.

NETWORKING FEATURES

NFS

Path must indicate a directory name when unmounting an NFS file system.

RETURN VALUE

If successful, *umount* returns a value of **0**. Otherwise, it returns a value of **−1** and sets **errno** to indicate the error.

ERRORS

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

[EPERM] The effective user ID of the process is not that of the superuser.

[ENOENT] *Name* does not exist.

[ENOTBLK] *Name* is not a block special device.

[EINVAL] *Name* is not mounted.

[EBUSY] A file on *name* is busy.

[EFAULT] *Name* points outside the allocated address space of the process. Reliable detection of this error is implementation dependent.

[ENXIO] The device associated with *name* does not exist.

[ENOTDIR] A component of *name* is not a directory.

[ENOENT] *Name* is null.

[ENAMETOOLONG] *Name* exceeds PATH_MAX bytes, or a component of *name* exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.

[EACCES] A component of the path prefix of *name* denies search permission.

[ELOOP] Too many symbolic links were encountered in translating the path name.

WARNINGS

If *umount* is called from the program level (that is, not from the *mount*(1M) level), the table of mounted devices contained in **/etc/mnttab** is not updated automatically.

DEPENDENCIES

HP Clustered Environment:

When *umount* is called from a diskless node and *path* refers to a block-special file, *path* is interpreted from the root server. This behavior is subject to change in future releases, and its use in applications is not recommended.

When *umount* is called from a diskless node and *path* refers to a directory on which is mounted a UFS file system (as opposed to an NFS file system; see *vfsmount*(2)), an EINVAL error is returned. This behavior is subject to change in future releases, and its use in applications is not recommended.

SEE ALSO

mount(1M), mount(2), vfsmount(2).

STANDARDS CONFORMANCE

umount: SVID2, XPG2

NAME

uname, setuname − get/set name of current HP-UX system

SYNOPSIS

#include <sys/utsname.h>

int uname (name)
struct utsname *name;

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

DESCRIPTION

Uname stores information identifying the current HP-UX system in the structure pointed to by *name*.

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

```
#define UTSLEN    9
#define SNLEN     15

char          sysname[UTSLEN];
char          nodename[UTSLEN];
char          release[UTSLEN];
char          version[UTSLEN];
char          machine[UTSLEN];
char          idnumber[SNLEN];
```

Uname returns a null-terminated string in each field. The *sysname* field contains "HP-UX". Similarly, the *nodename* field contains the name by which the system is known on a communications network. The *release* field contains the release number of the operating system, such as "1.0" or "3.0.1". The *version* field contains additional information about the operating system. The first character of the *version* field is set to:

Character	Series 800	Series 300
A	single user system	two-user system
B	16-user system	unlimited-users system
C	32-user system	
D	64-user system	
U	unlimited-users system	

(Note that the contents of the version field might change on future releases, as AT&T license agreement restrictions change.) The *machine* field contains a standard name that identifies the hardware on which the UNIX system is running. The *idnumber* is a unique identification number within that class of hardware, possibly a hardware or software serial number. This field returns the null string to indicate the lack of an identification number.

Setuname sets the *nodename* field in the *utsname* structure to *name,* which has a length of *namelen* characters. This is usually executed by **/etc/rc** when the system is bootstrapped. Names are limited to UTSLEN - 1 characters; UTSLEN is defined in **<sys/utsname.h>**.

ERRORS

[EPERM] *Setuname* is not executed by the superuser.

[EFAULT] *Name* points to an illegal address. The reliable detection of this error is implementation dependent.

RETURN VALUE

Upon successful completion, a non-negative value is returned. Otherwise, **−1** is returned and

errno is set to indicate the error.

AUTHOR

Uname was developed by AT&T Bell Laboratories and the Hewlett-Packard Company.

SEE ALSO

hostname(1), uname(1), gethostname(2), sethostname(2).

STANDARDS CONFORMANCE

uname: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

unlink – remove directory entry; delete file

SYNOPSIS

int unlink (path)
char *path;

DESCRIPTION

Unlink removes the directory entry named by the path name pointed to by *path*.

When all links to a file have been removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, the removal is postponed until all references to the file have been closed.

RETURN VALUE

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

ERRORS

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

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] The named file does not exist (for example, *path* is null or a component of *path* does not exist).

[EACCES] Search permission is denied for a component of the path prefix.

[EACCES] Write permission is denied on the directory containing the link to be removed.

[EPERM] The named file is a directory and the effective user ID of the process is not super-user.

[EBUSY] The entry to be unlinked is the mount point for a mounted file system.

[ETXTBSY] The entry to be unlinked is the last link to a pure procedure (shared text) file that is being executed.

[EROFS] The directory entry to be unlinked is part of a read-only file system.

[EFAULT] *Path* points outside the process's allocated address space. The reliable detection of this error will be implementation dependent.

[ENAMETOOLONG]
The length of the specified path name exceeds PATH_MAX bytes, or the length of a component of the path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.

[ELOOP] Too many symbolic links were encountered in translating the path name.

SEE ALSO

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

STANDARDS CONFORMANCE

unlink: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

ustat – get file system statistics

SYNOPSIS

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

int ustat (dev, buf)
dev_t dev;
struct ustat *buf;
```

DESCRIPTION

Ustat returns information about a mounted file system. *Dev* is a device number identifying a device containing a mounted file system. *Buf* is a pointer to a *ustat* structure (defined in **ustat.h**) that includes the following elements:

```
daddr_t f_tfree;     /* Total free blocks */
ino_t   f_tinode;    /* Number of free inodes */
char    f_fname[6];  /* Filsys name */
char    f_fpack[6];  /* Filsys pack name */
int     f_blksize;   /* Block size */
```

The values of the **f_tfree** and **f_blksize** fields are reported in fragment size units.

ERRORS

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

[EINVAL] *Dev* is not the device number of a device containing a mounted file system.

[EFAULT] *Buf* points outside the process's allocated address space. The reliable detection of this error will be implementation dependent.

RETURN VALUE

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

AUTHOR

Ustat was developed by AT&T Bell Laboratories and the Hewlett-Packard Company.

SEE ALSO

touch(1), stat(2), fs(4).

STANDARDS CONFORMANCE

ustat: SVID2, XPG2

NAME

utime – set file access and modification times

SYNOPSIS

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

int utime (path, times)
char *path;
struct utimbuf *times;
```

DESCRIPTION

Utime sets the access and modification times of the file to which the *path* argument refers.

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

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

The following times in the <**utimbuf**> structure, defined in <**unistd.h**>, are measured in seconds since 00:00:00 GMT, Jan. 1, 1970.

```
time_t  actime;      /* access time */
time_t  modtime;     /* modification time */
```

RETURN VALUE

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

ERRORS

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

[ENOENT] The named file does not exist.

[ENOTDIR] A component of the path prefix is not a directory.

[EACCES] Search permission is denied by a component of the path prefix.

[EPERM] The effective user ID is not superuser and not the owner of the file, and *times* is not a null pointer.

[EACCES] The effective user ID is not superuser and not the owner of the file, and *times* is a null pointer and write access is denied.

[EROFS] The file system containing the file is mounted read-only.

[EFAULT] *Times* is not a null pointer, and points outside the process's allocated address space. The reliable detection of this error is implementation dependent.

[EFAULT] *Path* points outside the process's allocated address space. The reliable detection of this error is implementation dependent.

[ENAMETOOLONG]
The length of the specified path name exceeds PATH_MAX bytes, or the length of a component of the path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.

SEE ALSO

touch(1), stat(2), unistd(5).

STANDARDS CONFORMANCE

utime: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

vfork − spawn new process in a virtual memory efficient way

SYNOPSIS

int vfork()

REMARKS

Vfork is provided as a higher performance version of *fork* on those systems which choose to provide it and for which there is a performance advantage.

Vfork differs from *fork* only in that the child process may share code and data with the calling process (parent process). This speeds the cloning activity significantly at a risk to the integrity of the parent process if *vfork* is misused.

The use of *vfork* for any purpose except as a prelude to an immediate *exec* or *exit* is not supported. Any program which relies upon the differences between *fork* and *vfork* is not portable across HP-UX systems.

All implementations of HP-UX must provide the entry *vfork*, but it is permissible for them to treat it identically to *fork*. Some implementations may not choose to distinguish the two because their implementation of fork is as efficient as possible, and others may not wish to carry the added overhead of two similar calls.

DESCRIPTION

Vfork can be used to create new processes without fully copying the address space of the old process. If a forked process is simply going to do an *exec*(2), the data space copied from the parent to the child by *fork*(2) is not used. This is particularly inefficient in a paged environment. *Vfork* is useful in this case. Depending upon the size of the parent's data space, it can give a significant performance improvement over *fork*.

Vfork differs from *fork* in that the child borrows the parent's memory and thread of control until a call to *exec* or an exit (either by a call to *exit*(2) or abnormally.) The parent process is suspended while the child is using its resources.

Vfork returns 0 in the child's context and (later) the pid of the child in the parent's context.

Vfork can normally be used just like *fork*. It does not work, however, to return while running in the child's context from the procedure which called *vfork* since the eventual return from *vfork* would then return to a no longer existent stack frame. Be careful, also, to call *_exit* rather than *exit* if you cannot *exec*, since *exit* will flush and close standard I/O channels, and thereby mess up the parent process's standard I/O data structures. (Even with *fork* it is wrong to call *exit* since buffered data would then be flushed twice.)

The [vfork,exec] window begins at the *vfork* call and ends when the child completes its *exec* call.

RETURN VALUE

Upon successful completion, *vfork* returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of −1 is returned to the parent, no child process is created, and **errno** is set to indicate the error.

ERRORS

Vfork fails and no child process are created if one or more of the following is true:

[EAGAIN] The system-wide limit on the total number of processes under execution would be exceeded.

[EAGAIN] The system-imposed limit on the total number of processes under execution by a single user would be exceeded.

DEPENDENCIES

Series 800

Process times for the parent and child processes within the [vfork,exec] window may be inaccurate.

The parent and child processes share the same stack space within the [vfork,exec] window. If the size of the stack has been changed within this window by the child process (return from or call to a function, for example), it is likely that the parent and child processes will be killed with signal SIGSEGV or SIGBUS.

In the [vfork,exec] window, a call to *signal*(2) that installs a catching function can affect handling of the signal by the parent. The parent is not affected if the handling is being set to SIG_DFL or SIG_IGN, or if either *sigaction*(2) or *sigvector*(2) is used.

AUTHOR

Vfork was developed by the University of California, Berkeley.

SEE ALSO

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

NAME

vfsmount – mount a file system

SYNOPSIS

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

int vfsmount(type, dir, flags, data)
int type;
char *dir;
int flags;
caddr_t data;
```

DESCRIPTION

Vfsmount attaches a file system to a directory. After a successful return, references to directory *dir* will refer to the root directory of the newly mounted file system. *Dir* is a pointer to a null-terminated string containing a path name. *Dir* must exist already, and must be a directory. Its old contents are inaccessible while the file system is mounted. *Vfsmount* differs from *mount*(2) in its ability to mount other than just a local file system.

Type indicates the type of the file system. It must be one of the types described below.

The *flags* argument determines whether the file system can be written on (functionally identical to the *rwflag* argument in *mount*(2) in this regard). It also controls whether programs from the mounted file system are allowed to have set-uid execution. Physically write-protected and magnetic tape file systems must be mounted read-only. Failure to do so will result in a return of −1 by *vfsmount* and a value of EIO in **errno**. The following values for the *flags* argument are defined in **<sys/mount.h>**:

M_RDONLY Mount done as read-only.

M_NOSUID Execution of set-uid programs not permitted.

Data is a pointer to a structure that contains arguments specific to the value contained in *type*. The following values for *types* are defined in **<sys/mount.h>**:

MOUNT_UFS Mount a local HFS file system. *Data* points to a structure of the following format:

```
struct ufs_args {
        char    *fspec;
};
```

Fspec points to the name of the block special file that is to be mounted. This is identical in use and function to the first argument for *mount*(2).

MOUNT_CDFS Mount a local CD-ROM file system. *Data* points to a structure of the following format:

```
struct cdfs_args {
        char    *fspec;
};
```

Fspec points to the name of the block special file that is to be mounted.

NETWORKING FEATURES

NFS

An additional value for the *type* argument is supported.

MOUNT_NFS Mount an NFS file system. *Data* points to a structure of the following format:

```
#include        <nfs/nfs.h>
#include        <netinet/in.h>

struct nfs_args {
        struct sockaddr_in  *addr;
        fhandle_t        *fh;
        int      flags;
        int      wsize;
        int      rsize;
        int      timeo;
        int      retrans;
        char     *hostname;
};
```

Addr points to a local socket address structure (see *inet*(7)), which is used by the system to communicate with the remote file server.

Fh points to a structure containing a *file handle*, an abstract data type that is used by the remote file server in serving an NFS request.

Flags is a bit map that sets options and indicates which of the following fields contain valid information. The following values of the bits are defined in <**nfs/nfs.h**>:

NFSMNT_SOFT Specify whether the mount is a soft mount or a hard mount. If set, the mount is soft and will cause requests to be retried *retrans* number of times. Otherwise, the mount is hard and requests will be tried forever.

NFSMNT_WSIZE
Set the write size.

NFSMNT_RSIZE
Set the read size.

NFSMNT_TIMEO
Set the initial timeout value.

NFSMNT_RETRANS
Set the number of request retries.

NFSMNT_HOSTNAME
Set a hostname.

NFSMNT_INT Set the option to have interruptible I/O to the mounted file system.

NFSMNT_NODEVS
Set the option to deny access to local devices via NFS device files. By default, access to local devices via NFS device files is allowed.

Wsize can be used to advise the system on the maximum number of data bytes to use for a single outgoing protocol (such as UDP) message. This value must be greater than 0. Default **wsize** is **8192**.

Rsize can be used to advise the system on the maximum number of data bytes to use for a single incoming protocol (such as UDP) message. This value must be greater than 0. Default **rsize** is **8192**.

Timeo can be used to advise the system on the time to wait between NFS request retries. This is in units of 0.1 seconds. This value must be greater than 0. Default **timeo** is **7**.

Retrans can be used to advise the system on the number of times the system will resend a request. This value must be 0 or greater. Default **retrans** is **4**.

Hostname is a name for the file server that can be used when any messages are given concerning the server. The string can be of length from 0 to 32 characters.

RETURN VALUE

Upon successful completion, *vfsmount* returns a value of **0**. Otherwise, no file system is mounted, a value of **−1** is returned and **errno** is set to indicate the error.

ERRORS

Vfsmount will fail when one of the following occurs:

[EBUSY] *Dir* is not a directory, or another process currently holds a reference to it.

[EBUSY] No space remains in the mount table.

[EBUSY] The super block for the file system had a bad magic number or an out-of-range block size.

[EBUSY] Not enough memory was available to read the cylinder group information for the file system.

[EFAULT] *Data* or *dir* points outside the allocated address space of the process.

[EIO] An I/O error occurred while reading from or writing to the file system.

[ELOOP] Too many symbolic links were encountered in translating the path name of file system referred to by *data* or *dir*.

[ENAMETOOLONG]
The path name of the file system referred to by *data* or *dir* PATH_MAX bytes, or the length of a component of the path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.

[ENOENT] The file system referred to by *data* or *dir* does not exist.

[ENOENT] The file system referred to by *data* does not exist.

[ENOTBLK] The file system referred to by *data* is not a block device. This is for a local mount.

[ENOTDIR] A component of the path prefix in *dir* is not a directory.

[ENOTDIR] A component of the path prefix of the file system referred to by *data* or *dir* is not a directory.

[ENXIO] The major device number of the file system referred to by *data* is out of range (this indicates no device driver exists for the associated hardware).

[EPERM] The caller is not the super-user.

DEPENDENCIES

NFS: *Vfsmount* fails when one of the following occurs, and returns the error indicated:

[EFAULT] A pointer in the *data* structure points outside the process's allocated address space.

[EINVAL] A value in a field of *data* is out of proper range.

[EREMOTE] An attempt was made to remotely mount a file system that was already mounted from another remote node.

See *getfh*(2), *inet*(7), and *mountd*(1M) for more information.

HP Clustered Environment:
Vfsmount of a local file system (**MOUNT_UFS**) is not supported from a cluster client. Such a call returns an EINVAL error.

WARNINGS

Use of *mount*(1M) is preferred over *vfsmount* because *mount*(1M) supports all mounting options that are available from *vfsmount* directly, plus *mount*(1M) also maintains the **/etc/mnttab** file which lists what file systems are mounted.

AUTHOR

Vfsmount was developed by HP and Sun Microsystems, Inc.

SEE ALSO

mount(2), umount(2), mount(1M).

NAME

wait, waitpid, wait3 – wait for child or traced process to stop or terminate

SYNOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait (stat_loc)
int *stat_loc;

pid_t wait ((int *)0)

pid_t waitpid (pid, stat_loc, options)
pid_t pid;
int *stat_loc;
int options;

pid_t wait3 (stat_loc, options, (int *)0)
int *stat_loc;
int options;
```

DESCRIPTION

Wait suspends the calling process until one of the immediate children terminates or until a process being traced stops, because that traced process has hit a break point. A process being traced can be either a child or a process attached by the *ptrace*(2) request PT_ATTACH (see *ptrace*(2)). The *wait* system call returns prematurely if a signal is received. If a child or traced process stops or terminates prior to the call on *wait*, return is immediate.

If *stat_loc* is not a null pointer, status information is stored in the location pointed to by *stat_loc*. The status can be used to differentiate between stopped and terminated processes. If the process terminates, the status identifies the cause of termination and passes useful information to the calling process. This is accomplished using the following macros defined in **<wait.h>**, with the status value stored at **stat_loc* as an argument:

WIFEXITED(*stat_val*)	If the process terminated because of an *exit*(2) or *_exit* system call, this macro evaluates to a non-zero value.
WEXITSTATUS(*stat_val*)	If the value of WIFEXITED(*stat_val*) is non-zero, this macro evaluates to the low-order 8 bits of the argument that the process passed to *exit* or *_exit* (see *exit*(2)).
WIFSIGNALED(*stat_val*)	If the process terminated due to the default action of a signal (see *signal*(5)), this macro evaluates to a non-zero value.
WTERMSIG(*stat_val*)	If the value of WIFSIGNALED(*stat_val*) is non-zero, this macro evaluates to the number of the signal that caused the termination.
WCOREDUMP(*stat_val*)	If the value of WIFSIGNALED(*stat_val*) is non-zero, this macro evaluates to a non-zero value if a "core image" was produced (see *signal*(5)).
WIFSTOPPED(*stat_val*)	If the process is stopped, this macro evaluates to a non-zero value.
WSTOPSIG(*stat_val*)	If the value of WIFSTOPPED(*stat_val*) is non-zero, this macro evaluates to the number of the signal that caused the process to stop.

As a single special case, the value stored in **stat_loc* is zero if and only if status is being returned from a terminated process that called *exit* or *_exit* with a value of zero.

If the information stored at the location pointed to by *stat_loc* was stored there by a call to one of the *wait* functions, exactly one of the macros WIFEXITED(**stat_loc*), WIFSIGNALED(**stat_loc*), and WIFSTOPPED(**stat_loc*) evaluates to a non-zero value.

The *waitpid* function behaves identically to *wait* if *pid* has a value of **−1** and *options* has a value of zero. Otherwise its behavior is modified by the values of the *pid* and *options* arguments.

The *pid* argument specifies the set of processes for which status is requested. The *waitpid* function returns only the status of a child process from this set.

- If *pid* is equal to **−1,** status is requested for any child process or attached process. In this respect, *waitpid* is then equivalent to *wait*.
- If *pid* is greater than zero, it specifies the process ID of a single child or attached process for which status is requested.
- If *pid* is equal to zero, status is requested for any child or attached process whose process group ID is equal to that of the calling process.
- If *pid* is less than **−1,** status is requested for any child or attached process whose process group ID is equal to the absolute value of *pid*.

The *options* argument is constructed from the bitwise inclusive OR of zero or more of the following flags:

WNOHANG — If this flag is set, *waitpid* or *wait3* is prevented from suspending the calling process. A value of zero is returned indicating that no child or traced processes have stopped or died.

WUNTRACED — If and only if this flag is set, *waitpid* or *wait3* returns information on child or attached processes that are stopped but not traced (with *ptrace*(2)) because they received a SIGTTIN, SIGTTOU, SIGTSTP, or SIGSTOP signal, and whose status has not yet been reported. Regardless of this flag, status is returned for child or attached processes that have terminated or are stopped and traced and whose status has not yet been reported.

Calling *wait3* is equivalent to calling *waitpid* with the value of *pid* equal to zero. The third parameter to *wait3* is currently unused and must always be a null pointer.

If a parent process terminates without waiting for its child processes to terminate, the parent process ID of each child process is set to 1. This means the initialization process inherits the child processes.

Notes

Earlier HP-UX versions documented the bit encodings of the status returned by *wait* rather than the macros WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WCOREDUMP, WIFSTOPPED, and WSTOPSIG. Applications using those bit encodings will continue to work correctly. However, new applications should use the macros for maximum portability.

In earlier HP-UX versions, the macros WIFSTOPPED, WIFSIGNALED and WIFEXITED have the same definitions as the corrspondingly named macros in the BSD 4.3 and earlier systems. Existing applications that depend on these definitions will continue to work correctly. However, if the application is recompiled, the feature test macro **_BSD** must be turned on for the compilation so that the old definitions of these macros are obtained. New definitions of these macros are in effect by default. The only difference between the old and new definitions is the type of the argument. Type **union wait** is used in the BSD definitions while type **int** is used in the default definitions.

ERRORS

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

[ECHILD]	The calling process to *wait* or *wait3* has no existing child or traced processes, or the calling process to *waitpid* has no existing unwaited-for child or traced processes that match the *pid* argument.
[ECHILD]	For *waitpid*, the process or process group specified by *pid* does not exist or is not a child of the calling process.
[EFAULT]	*Stat_loc* points to an illegal address. The reliable detection of this error is implementation dependent.
[EINVAL]	The *options* argument to *waitpid* or *wait3* is invalid.
[EINVAL]	*Wait3* was passed a non-null pointer value for its third argument.
[EINTR]	The function was interrupted by a signal. The value of the location pointed to by *stat_loc* is undefined.

RETURN VALUE

If *wait* returns due to the receipt of a signal, a value of **−1** is returned to the calling process and **errno** is set to EINTR. If *wait* returns due to a stopped or terminated child or traced process, the process ID of that process is returned to the calling process. If *waitpid* or *wait3* is called, the WNOHANG option is used, and there are no stopped or terminated child or traced processes (as specified by *pid* in the case of *waitpid*), a value of zero is returned. Otherwise, a value of **−1** is returned and **errno** is set to indicate the error.

WARNINGS

The behavior of *wait*, *waitpid*, and *wait3* is affected by setting the SIGCLD signal to SIG_IGN. See WARNINGS section of *signal*(5). Signal handlers that cause system calls to be restarted can affect the EINTR condition described above (see *sigaction*(2), *sigvector*(2), and *bsdproc*(2)).

AUTHOR

Wait, *waitpid*, and *wait3* were developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

Exit conditions (**$?**) in sh(1), exec(2), exit(2), fork(2), pause(2), ptrace(2), signal(5).

STANDARDS CONFORMANCE

wait: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

waitpid: XPG3, POSIX.1, FIPS 151-1

NAME

write, writev − write on a file

SYNOPSIS

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

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

int writev (fildes, iov, iovcnt)
int fildes;
struct iovec *iov;
int iovcnt;
```

DESCRIPTION

Write attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the file descriptor *fildes*. *Writev* performs the same action, but gathers the output data from the *iovlen* buffers specified by the elements of the *iovec* array: iov[0], iov[1], ..., RI iov[iovcnt − 1].

The *iovec* structure for *writev* is defined as follows:

```
struct iovec {
     caddr_t   iov_base;
     int       iov_len;
};
```

Each **iovec** entry specifies the base address and length of an area in memory from which data should be copied. The **iovec** array may be at most MAXIOV long.

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

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

If the O_APPEND file status flag is set, the file offset is set to the end of the file prior to each write.

For ordinary files, if the O_SYNC flag of the file status flags is set, the write does not return until both the file data and the file status are physically updated. For block special files, if O_SYNC is set, the write does not return until the data is physically updated. How the data reaches the physical media is implementation and hardware dependent.

If the number of bytes requested by *write* exceeds the allotted capacity (see *ulimit*(2)) or the physical end of a medium, only the allotted number of bytes are actually written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes will return 20. The next write of a non-zero number of bytes fails (except as noted below).

A write to an ordinary file is prevented if enforcement-mode file and record locking is set, and another process owns a lock on the segment of the file being written:

> If O_NDELAY or O_NONBLOCK is set, the write returns **−1** and sets **errno** to EAGAIN.
>
> If O_NDELAY and O_NONBLOCK are clear, the write does not complete until the blocking record lock is removed.

If the file being written is a pipe (or FIFO), the system-dependent maximum number of bytes that it can store is given by PIPSIZ (defined in **<sys/inode.h>**). The minimum value of PIPSIZ on any HP-UX system is 8192. When writing a pipe, the following conditions apply:

If the O_NDELAY or O_NONBLOCK file status flag is set:

> If *nbyte* is less than or equal to PIPSIZ and sufficient room exists in the pipe or FIFO, the *write* succeeds and returns the number of bytes written;
>
> If *nbyte* is less than or equal to PIPSIZ but insufficient room exists in the pipe or FIFO, the *write* returns having written nothing. If O_NONBLOCK is set, **−1** is returned and **errno** is set to EAGAIN. If O_NDELAY is set, **0** is returned.
>
> If *nbyte* is greater than PIPSIZ and the pipe or FIFO is full, the write returns having written nothing. If O_NONBLOCK is set, **−1** is returned and **errno** is set to EAGAIN. If O_NDELAY is set, **0** is returned.
>
> If *nbyte* is greater than PIPSIZ, and some room exists in the pipe or FIFO, as much data as fits in the pipe or FIFO is written, and *write* returns the number of bytes actually written, an amount less than the number of bytes requested.

If the O_NDELAY and O_NONBLOCK file status flags are clear:

> The *write* always executes correctly (blocking as necessary), and returns the number of bytes written.

RETURN VALUE

Upon successful completion, the number of bytes actually written is returned. Otherwise, **−1** is returned and **errno** is set to indicate the error.

ERRORS

Write fails and the file offset remains unchanged if any of the following conditions is true:

[EBADF] *Fildes* is not a valid file descriptor open for writing.

[EPIPE and SIGPIPE signal]
An attempt is made to write to a pipe that is not open for reading by any process.

[EINTR] A signal was caught during the *write* system call.

[EDEADLK] A resource deadlock would occur as a result of this operation (see *lockf*(2) and *fcntl*(2)).

[EAGAIN] Enforcement-mode file and record locking was set, O_NDELAY was set, and there was a blocking record lock.

[ENOLCK] The system record lock table is full, preventing the write from sleeping until the blocking record lock is removed.

[EIO] The process is in a background process group and is attempting to write to its controlling terminal, TOSTOP is set, the process is neither ignoring or blocking the SIGTTOU signal, and the process group of the process is orphaned.

[ENOSPC] Not enough space on the file system.

In addition, *writev* might return one of the following errors:

[EFAULT] *Iov_base* or *iov* points outside of the allocated address space. The reliable detection of this error is implementation dependent.

[EINVAL] *Iovcnt* is less than or equal to 0, or greater than MAXIOV.

[EINVAL] One of the *iov_len* values in the *iov* array was negative.

[EINVAL] The sum of *iov_len* values in the *iov* array overflowed a 32-bit integer.

Write or *writev* fails, the file offset is updated to reflect the amount of data transferred, and **errno** is set accordingly if one of the following conditions is true:

[EFBIG] An attempt was made to write a file that exceeds the process's file size limit or the maximum file size. See *ulimit*(2).

[EFAULT] *Buf* points outside the process's allocated address space. The reliable detection of this error is implementation dependent.

EXAMPLES

Assuming a process opened a file for writing, the following call to *write*(2) attempts to write *mybufsize* bytes to the file from the buffer to which *mybuf* points.

```
#include <string.h>

int mybufsize, nbytes, fildes;
char *mybuf = "aeiou and sometimes y";
mybufsize = strlen (mybuf);
nbytes = write (fildes, mybuf, mybufsize);
```

WARNINGS

Check all references to *signal*(5) for appropriateness on systems that support *sigvector*(2). *Sigvector*(2) can affect the behavior described on this page.

Character special devices, and raw disks in particular, apply constraints on how *write* can be used. See specific Section (7) manual entries for details on particular devices.

AUTHOR

Write was developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

creat(2), dup(2), lockf(2), lseek(2), open(2), pipe(2), ulimit(2), ustat(2).

STANDARDS CONFORMANCE

write: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

Section 3: Subroutine Libraries

NAME

intro – introduction to subroutines and libraries

SYNOPSIS

#include <stdio.h>

#include <math.h>

DESCRIPTION

This section describes functions found in various libraries, other than those functions that directly invoke HP-UX system primitives, which are described in Section (2) of this volume. Certain major collections are identified by a letter after the section identifier (3):

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

(3G) These functions constitute the graphics library, and are documented in separate manuals.

(3I) These functions constitute the instrument support library.

(3M) These functions constitute the Math Library. They are automatically loaded as needed by the FORTRAN compiler *f77*(1). 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 are applicable to the Internet network, and are part of the standard C library, **libc.a**. Section 3N manual entries are contained in the *Networking Reference*.

(3S) These functions constitute the "standard I/O package" (see *stdio*(3S)). These functions are in the library *libc*, already mentioned. Declarations for these functions may be obtained from the **#include** file **<stdio.h>**.

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

Definitions

A *character* is any bit pattern able to fit into a byte on the machine. The *null character* is a character with value 0, represented in the C language as **\0**. A *character array* is a sequence of characters. A *null-terminated character array* is a sequence of characters, the last of which is the *null character*. A *string* is a designation for a *null-terminated character array*. The *null string* is a character array containing only the null character. A **NULL** pointer is the value that is obtained by casting **0** into a pointer. The C language guarantees that this value will not match that of any legitimate pointer, so many functions that return pointers return it to indicate an error. **NULL** is defined as **0** in **<stdio.h>**; the user can include an appropriate definition if not using **<stdio.h>**.

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

DIAGNOSTICS

Functions in the C and Math Libraries, (3C) and (3M), may return the conventional values **0** or ±**HUGE** (the largest-magnitude single-precision floating-point numbers; **HUGE** is defined in the <**math.h**> header file) when the function is undefined for the given arguments or when the value is not representable. In these cases, the external variable **errno** (see *errno*(2)) is set to the value EDOM or ERANGE. As many of the FORTRAN intrinsic functions use the routines found in the Math Library, the same conventions apply.

WARNINGS

Library routines in libc.a and libm.a often call other routines in these libraries. Prior to HP-UX release 7.0, a user could define a function having the same name as one of these library routines, and this function would be linked in instead of the library version. In this way, a user could effectively replace a library routine with his own (see *matherr*(3M) for a supported example of this). More often, this type of linkage would occur unintentionally, causing unexpected behavior which was difficult to debug.

Starting at Release 7.0, object names in libraries have been modified such that they are much less likely to collide with user names. Therefore, calls to library routines from within other library routines are much more likely to call the actual library routine. (*Matherr*(3M) is the only exception to this.)

In spite of these changes, it is still remotely possible for name conflicts to occur. The *lint*(1) program checker reports name conflicts of this kind as "multiple declarations" of the names in question. Definitions for the Sections (2), (3C), and (3S) are checked automatically. Other definitions can be included by using the **−l** option (for example, **−lm** includes definitions for the Math Library, (3M)). Use of *lint*(1) is highly recommended.

FILES

/lib/libc.a
/lib/libm.a
/usr/lib/libF77.a

SEE ALSO

intro(2), stdio(3S), math(5), hier(5), ar(1), cc(1), f77(1), ld(1), lint(1), nm(1).

The introduction to this manual.

Device I/O Library, manual in *HP-UX Concepts and Tutorials: Device I/O and User Interfacing.*

NAME

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

SYNOPSIS

long a64l (s)
char *s;

char *l64a (l)
long l;

DESCRIPTION

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

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

The leftmost character is the least significant digit. For example,

$$a0 = (38 \times 64^0) + (2 \times 64^1) = 166$$

A64l takes a pointer to a null-terminated base-64 representation and returns a corresponding **long** value. If the string pointed to by *s* contains more than six characters, *a64l* will use the first six.

L64a takes a **long** argument and returns a pointer to the corresponding base-64 representation. If the argument is 0, *l64a* returns a pointer to a null string.

BUGS

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

STANDARDS CONFORMANCE

a64l: SVID2

l64a: SVID2

NAME

abort – generate a software abort fault

SYNOPSIS

#include <stdlib.h>

void abort();

DESCRIPTION

The *abort* function first closes all open files, streams, directory streams, and message catalogue descriptors, if possible, then causes the signal SIGABRT to be sent to the calling process. This may cause a core dump to be generated (see signal(2)).

If the signal SIGABRT is caught, the handling function is executed. If the handling function returns, the action for SIGABRT is then reset to SIG_DFL, and the signal SIGABRT is sent again to the process to ensure that it terminates.

RETURN VALUE

The *abort* function does not return.

ERRORS

No errors are defined.

APPLICATION USAGE

SIGABRT is not intended to be caught.

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 ALSO

adb(1), exit(2), kill(2), raise(2), signal(2). signal(5).

STANDARDS CONFORMANCE

abort: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

abs, labs – return integer absolute value

SYNOPSIS

#include <stdlib.h>

int abs (i)
int i;

long int labs (i)
long int i;

DESCRIPTION

Abs returns the absolute value of its integer operand.

The *labs* function is similar to the *abs* function, except that the argument and the returned value each have type long int.

The largest negative integer returns itself.

WARNINGS

In two's-complement representation, the absolute value of the negative integer with largest magnitude is undefined. Some implementations trap this error, but others simply ignore it.

SEE ALSO

floor(3M).

STANDARDS CONFORMANCE

abs: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

labs: XPG2

NAME

acltostr – convert access control list (ACL) structure to string form

SYNOPSIS

```
#include <acllib.h>

char * acltostr (nentries, acl, form)
int     nentries;
struct  acl_entry acl[];
int     form;
```

Remarks:

To ensure continued conformance with emerging industry standards, features described in this manual entry are likely to change in a future release.

DESCRIPTION

Acltostr converts an access control list from structure form to string representation. *Acltostr* takes a pointer to the first element of an array of ACL entries (*acl*), containing the indicated number (*nentries*) of valid entries (zero or more), and the output form desired (FORM_SHORT or FORM_LONG). It returns a pointer to a static string (overwritten by the next call), which is a symbolic representation of the ACL, ending in a null character. The output forms are described in *acl*(5). In long form, the string returned contains newline characters.

A user ID of ACL_NSUSER and a group ID of ACL_NSGROUP are both represented by %. Like *ls*(1), if an entry contains any other user ID or group ID value not listed in **/etc/passwd** or **/etc/group**, *acltostr* returns a string equivalent of the ID number instead.

Like routines that manage the **/etc/passwd** file, *acltostr* truncates user and group names to eight characters.

Note: *acltostr* is complementary in function to *strtoacl*.

RETURN VALUE

If *acltostr* succeeds, it returns a pointer to a null-terminated string. If *nentries* is zero or less, the string is of zero length. If *nentries* is greater than NACLENTRIES (defined in <**sys/acl.h**>), or if *form* is an invalid value, the call returns (char *) NULL.

EXAMPLES

The following code fragment reads the ACL on file "/users/ggd/test" and prints its short form representation.

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

int nentries;
struct acl_entry acl [NACLENTRIES];

if ((nentries = getacl ("/users/ggd/test", NACLENTRIES, acl)) < 0)
        error (...);

fputs (acltostr (nentries, acl, FORM_SHORT), stdout);
```

AUTHOR

Acltostr was developed by HP.

FILES

/etc/passwd
/etc/group

SEE ALSO

getacl(2), setacl(2), cpacl(3C), chownacl(3C), setaclentry(3C), strtoacl(3C), acl(5).

NAME

almanac – return numeric date information in MPE format

SYNOPSIS

void almanac (date, err, pyear, pmonth, pday, pweekday)
unsigned short date, err[2];
short *pyear, *pmonth, *pday, *pweekday;

DESCRIPTION

Almanac returns numeric date information for a date in the packed date format returned by the calendar(3X) routine. The returned information is:

year of the century
month of the year
day of the month
day of the week

The arguments to *almanac* are used as follows:

date An unsigned short containing the date about which information is to be returned. The year of the century is packed into bits 0 through 6, and the day of the year is packed into bits 7 through 15. The packed date format is:

```
Bits  0                6   7        15
     ---------------------------------
     |                   |           |
     | Year of Century   | Day of Year |
     |___________________|___________|
```

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
1	No parameters are present in which to return values: pday, pmonth, pyear, and pweek all point to zero.
2	Day of the year is out of range.
3	Year of the century is out of range.

pyear A pointer to a short in which the year of the century is returned.

pmonth A pointer to a short in which the month of the year is returned (for example, January is represented by **1** and December is represented by **12**).

pday A pointer to a short in which the day of the month is returned.

pweekday A pointer to a short in which the weekday is returned. Note that **1** will be returned for Sunday and **7** for Saturday.

WARNINGS

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR

Almanac was developed by HP.

SEE ALSO

calendar(3X), nlfmtdate(3X), ctime(3C), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

assert – verify program assertion

SYNOPSIS

#include <assert.h>

assert (expression)
int expression;

DESCRIPTION

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

"Assertion failed: *expression*, file *xyz*, line *nnn*"

on the standard error output and aborts. In the error message, *xyz* is the name of the source file and *nnn* the source line number of the *assert* statement.

Compiling with the preprocessor option **–DNDEBUG** (see *cpp*(1)), or with the preprocessor control statement "**#define** NDEBUG" ahead of the "**#include <assert.h>**" statement, stops assertions from being compiled into the program.

WARNINGS

The expression argument used by *assert* in compatibility mode cannot contain string literals or double quotes without escapes.

SEE ALSO

cpp(1), abort(3C).

STANDARDS CONFORMANCE

assert: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

j0, j1, jn, y0, y1, yn – Bessel functions

SYNOPSIS

```
#include <math.h>

double j0 (x)
double x;

double j1 (x)
double x;

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

double y0 (x)
double x;

double y1 (x)
double x;

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

DESCRIPTION

J0 and *j1* return Bessel functions of *x* of the first kind of orders 0 and 1 respectively. *Jn* returns the Bessel function of *x* of the first kind of order *n*.

Y0 and *y1* return the 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.

ERRORS

Series 300

Non-positive arguments cause *y0*, *y1* and *yn* to return the value **–HUGE_VAL** and to set **errno** to **EDOM**. They also cause a message indicating DOMAIN error to be printed on the standard error output.

Arguments too large in magnitude cause *j0*, *j1*, *jn*, *y0*, *y1* , and *yn* to return 0.0 and set **errno** to **ERANGE**. In addition, a message indicating TLOSS error is printed on the standard error output.

Series 800 (/lib/libm.a)

Non-positive arguments cause *y0*, *y1* , and *yn* to return the value **–HUGE_VAL** and to set **errno** to **EDOM**. They also cause a message indicating DOMAIN error to be printed on the standard error output.

Arguments too large in magnitude cause *j0*, *j1*, *jn*, *y0*, *y1* , and *yn* to return 0.0 and set **errno** to **ERANGE**. In addition, a message indicating TLOSS error is printed on the standard error output.

j0, *j1*, *jn*, *y0*, *y1* , and *yn* return NaN and set **errno** to **EDOM** when *x* is NaN or ±INFINITY . In addition, a message indicating DOMAIN error is printed on the standard error output.

Series 800 (ANSI C /lib/libM.a)

No error messages are printed on the standard error output.

Non-positive arguments cause *y0*, *y1* , and *yn* to return the value NaN and to set **errno** to **EDOM**. IP Arguments too large in magnitude cause *j0*, *j1*, *jn*, *y0*, *y1* , and *yn* to return 0.0 and set **errno** to **ERANGE**.

j0, j1, jn, y0, y1 , and *yn* return NaN and set **errno** to **EDOM** when *x* is NaN or ±INFINITY .

These error-handling procedures can be changed with the function *matherr*(3M).

SEE ALSO

isinf(3M), isnan(3M), matherr(3M).

STANDARDS CONFORMANCE

j0: SVID2, XPG2, XPG3

j1: SVID2, XPG2, XPG3

jn: SVID2, XPG2, XPG3

y0: SVID2, XPG2, XPG3

y1: SVID2, XPG2, XPG3

yn: SVID2, XPG2, XPG3

NAME

blmode – terminal block mode library interface

SYNOPSIS

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

int bfdes;

bfdes = blopen(fildes)
int fildes;

int blclose (bfdes)
int bfdes;

int blread (bfdes, buf, nbyte)
int bfdes;
char *buf;
unsigned nbyte;

int blget (bfdes, arg)
int bfdes;
struct blmodeio *arg;

int blset (bfdes, arg)
int bfdes;
struct blmodeio *arg;
```

DESCRIPTION

This terminal library interface allows support of block mode transfers with HP terminals. Block mode only affects input processing. Therefore, data is written with the standard *write*(2) interface.

In character mode the terminal sends each character to the system as it is typed. However, in block mode data is buffered and possibly edited locally in the terminal memory as it is typed, then sent as a block of data when the <**ENTER**> key is pressed on the terminal. During block mode data transmissions, the incoming data is not echoed by the interface and no special character processing is performed, other than recognizing a data block terminator character. For subsequent character mode transmissions, the existing termio state (see *termio*(7)) will continue to determine echo and character processing.

There are two parts of the block mode protocol, the block mode handshake and the block mode transmission.

Block mode handshake

At the beginning of a read, a *trigger* character is sent to the terminal to notify it that the system wants a block of data. (The *trigger* character, if defined, is sent at the beginning of all reads, character or block mode. It is necessary for block mode reads to work correctly.)

After receiving the *trigger* character, and when the user has typed all the data into the terminal's memory and pressed the <**ENTER**> key, the terminal will send an *alert* character to the system to notify it that the terminal has a block of data to send.

The system may then send user-definable cursor positioning or other data sequences, such as for home cursor or lock keyboard, to the terminal.

The system will then send a second *trigger* character to the terminal. The terminal will then transmit the data block as described in the **Block mode transmission** section.

Block mode transmission

The second part of the block mode protocol is the block mode transmission. After the block mode handshake has successfully completed, the terminal will transmit the data block to the system. During this transmission of data, the incoming data is not echoed by the system and no special character processing is performed, other than recognizing the data block termination character. It is possible to bypass the block mode handshake and have the block mode transmission occur after only the first *trigger* character is sent, see CB_BMTRANS below.

It is possible to intermix both character mode and block mode data transmissions. If CB_BMTRANS (see below) is set, all transfers will be block mode transfers. When CB_BMTRANS is not set, character mode transmissions will be processed as described in *termio*(7). In this case, if an *alert* character is received anywhere in the input data, the transmission mode will be switched to block mode automatically for a single transmission. Any data received before the *alert* will be discarded. The *alert* character may be escaped with a backslash ("\") character.

XON/XOFF flow control

To prevent data loss, XON/XOFF flow control should be used between the system and the terminal. The IXOFF bit (see *termio*(7)) should be set and the terminal strapped appropriately. If flow control is not used, it is possible for incoming data to overflow and be lost. (Note: some older terminals do not support this flow control.)

Read requests

Read requests that receive data from block mode transmissions will not return until the transmission is complete (the terminal has transmitted all characters). If the read is satisfied by byte count or if a data transmission error occurs, all subsequent data will be discarded until the transmission is complete. The read will wait until a terminator character is seen, or a time interval specified by the system has passed that is longer than necessary for the number of characters specified.

The data block terminator character will be included in the data returned to the user, and is included in the byte count. If the number of bytes transferred by the terminal in a block mode transfer exceeds the number of bytes requested by the user, the read will return the requested number of bytes and the remaining bytes will be discarded. The user can determine if data was discarded by checking the last character of the returned data. If the last character is not the terminator character, then more data was received than was requested and data was discarded.

The EIO error can be caused by several events, including errors in transmission, framing, parity, break, and overrun, or if the internal timer expires. The internal timer starts when the second trigger character is sent by the computer, and ends when the terminating character is received by the computer. The length of this timer is determined by the number of bytes requested in the read and the current baud rate, plus an additional ten seconds.

User control of handshaking

If desired, the application program can provide its own handshake mechanism in response to the *alert* character by selecting the OWNTERM mode, see CB_OWNTERM below. With this mode selected, the driver will complete a read request when the *alert* character is received. No data will be discarded before the *alert*, and the *alert* will be returned in the data read. The *alert* character may be escaped with a backslash ("\") character. The second *trigger* will be sent when the application issues the next read.

Blmode control calls

First, the standard *open*(2) call to a tty device must be made to obtain a file descriptor for the subsequent block mode control calls (an *open*(2) will be done automatically by the system for *stdin* on the terminal).

int bfdes;

bfdes = blopen (fildes)
int fildes;

A call to *blopen* must be made before any block mode access is allowed on the specified file descriptor. *Blopen* will initialize the block mode parameters as described below. The return value from *blopen* is a block mode file descriptor that must be passed to all subsequent block mode control calls.

int blclose (bfdes)
int bfdes;

A call to *blclose* must be issued before the standard *close*(2) to ensure proper closure of the device. Otherwise unpredictable results may occur. The argument **bfdes** is the file descriptor returned from a previous *blopen* system call.

int blread (bfdes, buf, nbyte)
int bfdes;
char *buf;
unsigned nbyte;

The *blread* routine has the same parameters as the *read*(2) sytem call. At the beginning of a read, the *cb_trig1c* character (if defined) is sent to the device. If CB_BMTRANS is not set, and no *cb_alertc* character is received, the read data will be processed according to *termio*(7). If CB_BMTRANS is set, or if a non-escaped *cb_alertc* character is received, echo will be turned off for the duration of the transfer, and no further special character processing will be done other than that required for the termination character. The argument **bfdes** is the file descriptor returned from a previous *blopen* system call.

int blget (bfdes, arg)
int bfdes;
struct blmodeio *arg;

A call to *blget* will return the current values of the **blmodeio** structure (see below). The argument **bfdes** is the file descriptor returned from a previous *blopen* system call.

int blset (bfdes, arg)
int bfdes;
struct blmodeio *arg;

A call to *blset* will set the block mode values from the structure whose address is *arg*. The argument **bfdes** is the file descriptor returned from a previous *blopen* system call.

Blmode structure

The two block mode control calls, *blget* and *blset*, use the following structure, defined in <**sys/blmodeio.h**>:

```
#define   NBREPLY   64

struct    blmodeio       {
          unsigned long  cb_flags;            /* Modes */
          unsigned char  cb_trig1c;           /* First trigger */
          unsigned char  cb_trig2c;           /* Second trigger */
          unsigned char  cb_alertc;           /* Alert character */
          unsigned char  cb_termc;            /* Terminating char */
          unsigned char  cb_replen;           /* cb_reply length */
          char           cb_reply[NBREPLY];   /* optional reply */
};
```

The *cb_flags* field controls the basic block mode protocol:

CB_BMTRANS	0000001	Enable mandatory block mode transmission.
CB_OWNTERM	0000002	Enable user control of handshake.

If CB_BMTRANS is set, all transmissions are processed as block mode transmissions. The block mode handshake is not required and data read is processed as block mode transfer data. The block mode handshake may still be invoked by receipt of an *alert* character as the first character seen. A **blread** issued with the CB_BMTRANS bit set will cause any existing input buffer data to be flushed.

If CB_BMTRANS is not set, and if the *alert* character is defined and is detected anywhere in the input stream, the input buffer will be flushed and the block mode handshake will be invoked. The system will then send the *cb_trig2c* character to the terminal, and a block mode transfer will follow. The *alert* character can be escaped by preceding it with a backslash ("\").

If CB_OWNTERM is set, reads will be terminated upon receipt of a non-escaped *alert* character. No input buffer flushing is performed, and the *alert* character is returned in the data read. This allows application code to perform its own block mode handshaking. If the bit is clear, a non-escaped *alert* character will cause normal block mode handshaking to be used.

The initial *cb_flags* value is all-bits-cleared.

There are several special characters (both input and output) that are used with block mode. These characters and the initial values for these characters are described below. Any of these characters may be undefined by setting its value to 0377.

cb_trig1c is the initial *trigger* character sent to the terminal at the beginning of a read request.

cb_trig2c is the secondary *trigger* character sent to the terminal after the *alert* character has been seen.

cb_alertc is the *alert* character sent by the terminal in response to the first *trigger* character. It signifies that the terminal is ready to send the data block. The *alert* character can be escaped by preceding it with a backslash ("\").

cb_termc is sent by the terminal after the block mode transfer has completed. It signifies the end of the data block to the computer.

The *cb_replen* field specifies the length in bytes of the *cb_reply* field. If set to zero, the *cb_reply* string will not be used. The *cb_replen* field is initially set to zero.

The *cb_reply* array contains a string to be sent out after receipt of the *alert* character, but before the second *trigger* character is sent by the computer. Any character may be included in the reply string. The number of characters sent is specified by *cb_replen*. The initial value of all characters in the *cb_reply* array is NULL.

RETURNS

If an error occurs, all calls will return a value of -1 and **errno** will be set to indicate the error. If no error is detected, *blread* will return the number of characters read. All other calls will return 0 upon successful completion.

During a read, it is possible for the user's buffer to be altered even if an error value is returned. The data in the user's buffer should be ignored as it will not be complete. The following errors may be returned by various library calls described in this document.

blopen

[ENOTTY] The file descriptor specified is not related to a terminal device.

blclose

	[ENOTTY]	No previous **blopen** has been issued for the specified file descriptor.
blread		
	[EDEADLK]	A resource deadlock would occur as a result of this operation (see *lockf*(2)).
	[EFAULT]	**Buf** points outside the allocated address space. The reliable detection of this error will be implementation dependent.
	[EINTR]	A signal was caught during the **read** system call.
	[EIO]	An I/O error occured during block mode data transmissions.
	[ENOTTY]	No previous **blopen** has been issued for the specified file descriptor.
blget		
	[ENOTTY]	No previous **blopen** has been issued for the specified file descriptor.
blset		
	[EINVAL]	An illegal value was specified in the structure passed to the system.
	[ENOTTY]	No previous **blopen** has been issued for the specified file descriptor.

WARNINGS

Once **blopen** has been called with a file descriptor and returned successfully, that file descriptor should not subsequently be used as a parameter to the following system calls: *close*(2), *dup*(2), *dup2*(2), *fcntl*(2), *ioctl*(2), *read*(2), or *select*(2) until a **blclose** is called with the same file descriptor as its parameter. Additionally, *scanf*(libc), *fscanf*(libc), *getc*(libc), *getchar*(libc), *fgetc*(libc) and *fgetw*(libc) should not be called for a stream associated with a file descriptor that has been used in a **blopen** call but has not been used in a **blclose** call. These functions call *read*(2) and calling these routines will result in unpredictable behavior.

AUTHOR

Blmode was developed by HP.

SEE ALSO

termio(7).

NAME

bsearch – binary search a sorted table

SYNOPSIS

```
#include <stdlib.h>

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

DESCRIPTION

Bsearch is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table indicating where a datum may be found. The table must be previously sorted in increasing order according to a provided comparison function. *Key* points to a datum instance to be sought in the table. *Base* points to the element at the base of the table. *Nel* is the number of elements in the table. *Size* is the size of each element in the table. *Compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero as accordingly the first argument is to be considered less than, equal to, or greater than the second.

EXAMPLE

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

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

```
#include <stdio.h>

#define TABSIZE         1000

struct node {                   /* these are stored in the table */
        char *string;
        int length;
};
struct node table[TABSIZE];     /* table to be searched */
        .
        .
        .
{
        struct node *node_ptr, node;
        int node_compare( );  /* routine to compare 2 nodes */
        char str_space[20];   /* space to read string into */
        .
        .
        .
        node.string = str_space;
        while (scanf("%s", node.string) != EOF) {
                node_ptr = (struct node *)bsearch((void *)(&node),
                                (void *)table, TABSIZE,
                                sizeof(struct node), node_compare);
                if (node_ptr != NULL) {
```

```
                        (void)printf("string = %20s, length = %d\n",
                              node_ptr->string, node_ptr->length);
                  } else {
                        (void)printf("not found: %s\n", node.string);
                  }
            }
      }
      /* This routine compares two nodes based on an
            alphabetical ordering of the string field.  */
      int
      node_compare(node1, node2)
      struct node *node1, *node2;
      {
            return strcmp(node1->string, node2->string);
      }
```

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-void.
The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.
Although declared as type pointer-to-void, the value returned should be cast into type pointer-to-element.

SEE ALSO

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

DIAGNOSTICS

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

WARNINGS

If the table being searched contains two or more entries that match the selection criteria, a random entry is returned by *bsearch* as determined by the search algorithm.

STANDARDS CONFORMANCE

bsearch: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

calendar – return the MPE calendar date

SYNOPSIS

unsigned short calendar()

DESCRIPTION

This routine returns the calendar date in the format:

```
Bits  0                  6   7         15
     ------------------------------------
     |                    |             |
     | Year of Century    | Day of Year |
     |____________________|_____________|
```

RETURN VALUE

An unsigned short integer containing the calendar format.

WARNINGS

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR

Calendar was developed by HP.

SEE ALSO

portnls(5).

NAME

catgetmsg – get message from a message catalog

SYNOPSIS

#include <nl_types.h>

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

DESCRIPTION

Catgetmsg reads message *msg_num* in set *set_num* from the message catalog indentified by *catd*, a catalog descriptor returned from a previous call to *catopen*(3C). The return message is stored in *buf*, a buffer of length *buflen* bytes.

A message longer than *buflen*-1 bytes is silently truncated. The return message is always terminated with a null byte.

RETURN VALUE

If successful, *catgetmsg* returns a pointer to the message in *buf*. Otherwise, *catgetmsg* returns a pointer to an empty (null) string and sets **errno** to indicate the error. If *buflen* is greater than zero, the pointer returned is *buf*.

ERRORS

Catgetmsg fails and **errno** is set if one of the following conditions is true:

[EBADF] *Catd* is not a valid catalog descriptor.

[EINVAL] *Buflen* is less than **1**.

[EINVAL] *Set_num* and/or *msg_num* are not in the message catalog.

[EINVAL] The message catalog identified by *catd* is corrupted.

[EINTR] A signal was caught during the *read*(2) system call.

[EFAULT] *Buf* points outside the allocated address space. The reliable detection of this error is implementation dependent.

[ERANGE] A message longer than *buflen*-1 bytes was truncated.

AUTHOR

Catgetmsg was developed by HP.

SEE ALSO

catopen(3C), catgets(3C), read(2).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

STANDARDS CONFORMANCE

catgetmsg: XPG2

NAME

catgets – get a program message

SYNOPSIS

#include <nl_types.h>

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

DESCRIPTION

Catgets reads message *msg_num* in set *set_num* from the message catalog identified by *catd*, a catalog descriptor returned from a previous call to *catopen*(3C). *Def_str* points to a default message string returned by *catgets* if the call fails.

A message longer than NL_TEXTMAX bytes is silently truncated. The returned message string is always terminated with a null byte. NL_TEXTMAX is defined in **<limits.h>**.

RETURN VALUE

If the call is successful, *catgets* returns a pointer to an internal buffer area containing the null-terminated message string. If the call is unsuccessful *catgets* returns a pointer to *def_str*.

WARNINGS

Catgets returns a pointer to a static area that is overwritten on each call.

AUTHOR

Catgets was developed by HP.

SEE ALSO

catopen(3C), catgetmsg(3C).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

STANDARDS CONFORMANCE

catgets: XPG2, XPG3

NAME

catopen, catclose – open and close a message catalog for reading

SYNOPSIS

#include <nl_types.h>

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

int catclose (catd)
nl_catd catd;

DESCRIPTION

Catopen opens a message catalog and returns a catalog descriptor. *Name* specifies the name of the message catalog being opened. A *name* containing a / (slash) specifies a path name for the message catalog. Otherwise, the environment variable NLSPATH is used, see *environ*(5). If NLSPATH specifies more than one path, *catopen* returns the catalog descriptor for the first path on which it is able to successfully open the specified message catalog. If NLSPATH does not exist in the environment or if a message catalog cannot be opened for any NLSPATH-specified path, *catopen* uses a systemwide default path. *Name* must not contain "%N".

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

Catclose closes the message catalog *catd*, a message catalog descriptor returned from an earlier successful call of *catopen*.

RETURN VALUE

Catopen returns a message catalog descriptor if successful. Otherwise, a value of **(nl_catd) -1** is returned and **errno** is set to indicate the error.

Catclose returns **0** if successful. Otherwise, a value of **–1** is returned and **errno** is set to indicate the error.

ERRORS

Catopen fails, no message catalog is opened, and **errno** is set for the last path attempted if any of the following conditions is true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named catalog does not exist.
[ENOENT]	The path is null.
[EACCES]	A component of the path prefix denies search permission.
[EACCES]	Read permission is denied for the named file.
[EMFILE]	The maximum number of file descriptors allowed are currently open.
[ENAMETOOLONG]	The length of the specified path name exceeds PATH_MAX bytes, or the length of a component of the path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.
[EINVAL]	The *name* argument contains "%N".

Catclose fails if the following is true:

[EBADF]	*Catd* is not a valid open message-catalog descriptor.

WARNINGS

When using NLSPATH, *catopen* does not provide a default value for LANG.

NOTES

Catgets(3C) can be used to provide default messages when called following a failed *catopen*. *Catgets* will return its *def_str* parameter if it is passed an invalid catalog descriptor.

AUTHOR

Catopen was developed by HP.

FILES

/usr/lib/nls	Message catalog default path.

SEE ALSO

catgetmsg(3C), catgets(3C), environ(5).

STANDARDS CONFORMANCE

catopen: XPG2, XPG3

catclose: XPG2, XPG3

NAME

catread – MPE/RTE-style message catalog support

SYNOPSIS

int catread (fd, set_num, msg_num, msg_buf, buflen [,arg]...)
int fd, set_num, msg_num, buflen;
char *msg_buf, *arg;

DESCRIPTION

Catread reads message number *msg_num* of set *set_num* in the message catalog identified by *fd*, a file descriptor returned from a previous call to *open*(2). The return message is stored in *buf*, a buffer of length *buflen* bytes.

The message read from the catalog can have embedded formatting information in the form ![*digit*]. Exclamation marks must be all numbered or all unnumbered. If exclamation marks are numbered, an exclamation mark followed by digit *n* is replaced by the *n*th *arg*. If exclamation marks are unnumbered, they are replaced by the *args* in serial order. If there are fewer *args* than exclamation marks, the results are undefined. If there are more *args* than exclamation marks, the excess *args* are ignored.

A character in a message may be quoted (that is, made to stand for itself) by preceding it with a tilde (~). To use the special characters **!** or ~ in a message, preceed the special character with ~.

A message longer than *buflen*-**1** bytes is silently truncated. The return message is always terminated with a null byte.

Catread is provided to support message catalog applications from MPE/RTE. (MPE and RTE are HP operating systems.)

RETURN VALUE

If successful, *catread* returns the length, in bytes, of the formatted message in *msg_buf*. Otherwise, if *set_num* or *msg_num* is not found in the catalog, *catread* returns a negative integer.

ERRORS

Catread succeeds, but sets **errno** if the following condition is true:

[ERANGE] Formatted message exceeds *buflen*-**1** bytes.

AUTHOR

Catread was developed by HP.

SEE ALSO

gencat(1), getmsg(3C), hpnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

cfgetospeed, cfsetospeed, cfgetispeed, cfsetispeed – tty 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

These functions set and get the input and output speed codes in the *termios* structure referenced by *termios_p*. The *termios* structure contains these speed codes representing input and output baud rates as well as other terminal related parameters. Setting the parameters on a terminal file do not become effective until **tcsetattr** is successfully called.

Cfgetospeed returns the output speed code from the *termios* structure referenced by *termios_p*.

Cfsetospeed sets the output speed code in the *termios* structure referenced by *termios_p* to *speed*. The speed code for a baud rate of zero, **B0**, is used to terminate the connection. If **B0** is specified, the modem control lines will no longer be asserted. Normally, this will disconnect the line.

Cfgetispeed returns the input speed code from the *termios* structure referenced by *termios_p*.

Cfsetispeed sets the input speed code in the *termios* structure referenced by *termios_p* to *speed*.

RETURN VALUE

Cfgetospeed returns the output speed code from the *termios* structure referenced by *termios_p*.

Cfgetispeed returns the input speed code from the *termios* structure referenced by *termios_p*.

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

ERRORS

Cfsetispeed and *cfsetospeed* will fail when the following is true:

[EINVAL] The value of *speed* is outside the range of possible speed codes as specified in **termios.h**.

WARNINGS

Cfsetispeed and *cfsetospeed* can be used to set speed codes in the *termios* structure that are not supported by the terminal hardware.

SEE ALSO

tcattribute(3C), termio(7).

STANDARDS CONFORMANCE

cfgetispeed: XPG3, POSIX.1, FIPS 151-1

cfgetospeed: XPG3, POSIX.1, FIPS 151-1

cfsetispeed: XPG3, POSIX.1, FIPS 151-1 *cfsetospeed*: XPG3, POSIX.1, FIPS 151-1

NAME

chownacl – change owner and/or group represented in a file's access control list (ACL)

SYNOPSIS

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

void chownacl (nentries, acl, olduid, oldgid, newuid, newgid)
int     nentries;
struct  acl_entry acl[];
int     olduid, oldgid;
int     newuid, newgid;
```

Remarks:

To ensure continued conformance with emerging industry standards, features described in this manual entry are likely to change in a future release.

DESCRIPTION

This routine alters an access control list (ACL) to reflect the change in a file's owner or group ID when an old file is copied to a new file and the ACL is also copied. *Chownacl* transfers ownership (that is, it modifies base ACL entries) like *chown*(2). The algorithm is described below and also in *acl*(5).

The *nentries* parameter is the current number of ACL entries in the *acl*[] array (zero or more; a negative value is treated as zero). The *olduid* and *oldgid* values are the user and group IDs of the original file's owner, typically the *st_uid* and *st_gid* values from *stat*(2). The *newuid* and *newgid* values are the user and group IDs of the new file's owner, typically the return values from *geteuid*(2) and *getegid*(2).

If an ACL entry in *acl*[] has a *uid* of *olduid* and a *gid* of ACL_NSGROUP (that is, an owner base ACL entry), *chownacl* changes *uid* to *newuid* (with exceptions, see below). If an entry has a *uid* of ACL_NSUSER and a *gid* of *oldgid* (that is, a group base ACL entry), *chownacl* changes *gid* to *newgid*. In either case, only the last matching ACL entry is altered; a valid ACL can have only one of each type.

Like *chown*(2), if the new user or group already has an ACL entry (that is, a *uid* of *newuid* and a *gid* of ACL_NSGROUP, or a *uid* of ACL_NSUSER and a *gid* of *newgid*), *chownacl* does not change the old user or group base ACL entry; both the old and new ACL entries are preserved.

As a special case, if *olduid* (*oldgid*) is equal to *newuid* (*newgid*), *chownacl* does not search *acl*[] for an old user (group) base ACL entry to change. Calling it with both *olduid* equal to *newuid* and *oldgid* equal to *newgid* causes *chownacl*(3C) to do nothing.

Suggested Use

This routine is useful in a program that creates a new or replacement copy of a file whose original was (or possibly was) owned by a different user or group, and that copies the old file's ACL to the new file. Copying another user's and/or group's file is equivalent to having the original file's owner and/or group copy and then transfer a file to a new owner and/or group using *chown*(2). This routine is not needed for merely changing a file's ownership; *chown*(2) modifies the ACL appropriately in that case.

If a program also copies file miscellaneous mode bits from an old file to a new one, it must use *chmod*(2). However, since *chmod* deletes optional ACL entries, it must be called before *setacl*(2). Furthermore, to avoid leaving a new file temporarily unprotected, the *chmod* call should set only the file miscellanous mode bits, with all access permission mode bits set to zero (that is, mask the mode with 07000). The *cpacl*(3C) library call encapsulates this operation, and handles remote files appropriately too.

EXAMPLES

The following code fragment gets *stat* information and the ACL from **oldfile**, transfers ownership of **newfile** to the caller, and sets the revised ACL to **newfile**.

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

int nentries;
struct acl_entry acl [NACLENTRIES];
struct stat statbuf;

if (stat ("oldfile", & statbuf) < 0)
        error (...);

if ((nentries = getacl ("oldfile", NACLENTRIES, acl)) < 0)
        error (...);

chownacl (nentries, acl, statbuf.st_uid, statbuf.st_gid,
        geteuid(), getegid());

if (setacl ("newfile", nentries, acl))
        error (...);
```

AUTHOR

Chownacl was developed by HP.

SEE ALSO

chown(2), getacl(2), getegid(2), geteuid(2), setacl(2), stat(2), acltostr(3C), cpacl(3C), setaclentry(3C), strtoacl(3C), acl(5).

NAME

clock – report CPU time used

SYNOPSIS

#include <time.h>

clock_t clock ( )

DESCRIPTION

Clock returns the amount of CPU time (in microseconds) used since the first call to *clock*. The time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed *wait*(2) or *system*(3S). To determine the time in seconds, the value returned by the *clock* function should be divided by the value of the macro CLOCKS_PER_SEC.

The resolution of the clock varies, depending on the hardware and on the software configuration.

If the processor time used is not available or its value cannot be represented, the function returns the value (clock_t)-1.

WARNINGS

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).

DEPENDENCIES

Series 300
: The clock resolution is 20 milliseconds.

Series 800
: The default clock resolution is 10 milliseconds.

SEE ALSO

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

STANDARDS CONFORMANCE

clock: SVID2, XPG2, XPG3, ANSI C

NAME
clock – return the MPE clock value

SYNOPSIS
unsigned int clock()

DESCRIPTION
This routine returns the clock value in the MPE format.

RETURN VALUE
The function returns an unsigned int in the format:

```
Bits  0                 7    8              15
      ----------------------------------------
     |                  |                    |
     | Hour of Day      | Minute of Hour     |
     |__________________|____________________|

Bits  16          23   24                 31
      ---------------------------------------
     |             |                        |
     | Seconds     | Tenths of Seconds      |
     |_____________|________________________|
```

WARNINGS
This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR
clock was developed by HP.

SEE ALSO
nlconvclock(3X), nlfmtclock(3X), portnls(5).

NAME

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

SYNOPSIS

#include <ctype.h>

int toupper (c)
int c;

int tolower (c)
int c;

int _toupper (c)
int c;

int _tolower (c)
int c;

int toascii (c)
int c;

DESCRIPTION

Toupper and *tolower* have as domain the range of *getc*(3S): the integers from −1 through 255. If the argument of *toupper* represents a lowercase letter, the result is the corresponding uppercase letter. If the argument of *tolower* represents an uppercase letter, the result is the corresponding lowercase letter. All other arguments in the domain are returned unchanged. Arguments outside the domain cause undefined results.

The macros *_toupper* and *_tolower* perform the same translations as *toupper* and *tolower*, but have restricted domains and are faster. The domains of *_toupper* and *_tolower* are the integers from 0 through 255. Arguments outside of 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.

WARNING

The *toascii* routine is supplied both as a library function and as a macro defined in the <ctype.h> header. Normally, the macro version will be used. To obtain the library function either use a #undef to remove the macro definition or, if compiling in ANSI C mode, enclose the function name in parenthesis or take its address. The following examples will use the library function for toascii:

```
#include <ctype.h>
#undef toascii
...
main()
{
      ...
      c1 = toascii(c);
      ...
}
```

or

```
#include <ctype.h>
...
main()
{
      int (*conv_func)();
```

```
        ...
        c1 = (toascii)(c);
        ...
        conv_func = toascii;
        ...
    }
```

EXTERNAL INFLUENCES

Locale

The LC_CTYPE category determines the translations to be done.

International Code Set Support

Single-byte character code sets are supported.

AUTHOR

Conv(3C) was developed by AT&T and HP.

SEE ALSO

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

STANDARDS CONFORMANCE

_tolower: SVID2, XPG2, XPG3

_toupper: SVID2, XPG2, XPG3

toascii: SVID2, XPG2, XPG3

tolower: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

toupper: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

cpacl, fcpacl – copy the access control list (ACL) and mode bits from one file to another

SYNOPSIS

int cpacl (fromfile, tofile, frommode, fromuid, fromgid, touid, togid)
char *fromfile, *tofile;
int frommode;
int fromuid, touid;
int fromgid, togid;

int fcpacl (fromfd, tofd, frommode, fromuid, fromgid, touid, togid)
int fromfd, tofd;
int frommode;
int fromuid, touid;
int fromgid, togid;

Remarks:

To ensure continued conformance with emerging industry standards, features described in this manual entry are likely to change in a future release.

DESCRIPTION

Both *cpacl* and *fcpacl* copy the access control list and mode bits (that is, file access permission bits and miscellaneous mode bits; see *chmod*(2)) from one file to another, and transfer ownership much like *chown*(2). *Cpacl* and *fcpacl* take the following parameters:

- Path names (*fromfile* and *tofile*) or open file descriptors (*fromfd* and *tofd*).
- A mode value (*frommode*, typically the *st_mode* value returned by *stat*(2)) containing file miscellaneous mode bits, which are always copied, and file access permission bits, which are copied instead of the access control list if either file is remote.
- User ID and group ID of the file (*fromuid*, *touid* and *fromgid*, *togid*) for transferring ownership. (Typically *fromuid* and *fromgid* are the *st_uid* and *st_gid* values returned by *stat*, and *touid* and *togid* are the return values from *geteuid*(2) and *getegid*(2).)

When both files are local, the *cpacl* routines copy the access control list and call *chownacl*(3C) to transfer ownership from the *fromfile* to the *tofile*, if necessary.

Cpacl (*fcpacl*) handles remote copying (via RFA or NFS) after recognizing failures of *getacl*(2) (*fgetacl*) or *setacl*(2) (*fsetacl*). When copying the mode from *fromfile* (*fromfd*) to *tofile* (*tofd*), *cpacl* copies the entire *frommode* (that is, the file miscellaneous mode bits and the file access permission bits) to *tofile* (*tofd*) using *chmod*(2) (*fchmod*(2)). Some of the miscellaneous mode bits may be turned off; see *chmod*(2).

Cpacl (*fcpacl*) can copy an access control list from *fromfile* (*fromfd*) to *tofile* (*tofd*) without transferring ownership, but ensuring error checking and handling of remote files. This is done by passing *fromuid* equal to *touid* and *fromgid* equal to *togid* (that is, four zeros). For remote files, *fromuid, touid, fromgid,* and *togid* are ignored.

RETURN VALUE

If successful, *cpacl* and *fcpacl* return zero. If an error occurs, they set **errno** to indicate the cause of failure and return a negative value, as follows:

−1 Unable to perform *getacl* (*fgetacl*) on a local *fromfile* (*fromfd*).

−2 Unable to perform *chmod* (*fchmod*) on *tofile* (*tofd*) to set its file miscellaneous mode bits. *Cpacl* (*fcpacl*) attempts this regardless of whether a file is local or remote, as long as *fromfile* (*fromfd*) is local.

−3 Unable to perform *setacl* (*fsetacl*) on a local *tofile* (*tofd*). As a consequence, the file's optional ACL entries are deleted, its file access permission bits are zeroed, and its miscellaneous mode bits might be altered.

−4 Unable to perform *chmod* (*fchmod*) on *tofile* (*tofd*) to set its mode. As a consequence, if *fromfile* (*fromfd*) is local, *tofile*'s (*tofd*'s) optional ACL entries are deleted, its access permission bits are zeroed, and its file miscellaneous mode bits might be altered, regardless of whether the file is local or remote.

EXAMPLES

The following code fragment gets *stat* information on "oldfile" and copies its file miscellaneous bits and access control list to "newfile" owned by the caller. If either file is remote, only the *st_mode* on "oldfile" is copied.

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

struct stat statbuf;

if (stat ("oldfile", & statbuf) < 0)
        error (...);

if (cpacl ("oldfile", "newfile", statbuf.st_mode,
        statbuf.st_uid, statbuf.st_gid, geteuid(), getegid()) < 0)
{
        error (...);
}
```

DEPENDENCIES

RFA and NFS

Fcpacl fails if *tofile* is RFA-remote.

AUTHOR

Cpacl and *fcpacl* were developed by HP.

SEE ALSO

chown(2), getacl(2), getegid(2), geteuid(2), setacl(2), stat(2). acltostr(3C), chownacl(3C), setentry(3C), strtoacl(3C), acl(5).

NAME

crt0.o, gcrt0.o, mcrt0.o, frt0.o, gfrt0.o, mfrt0.o – execution startup routines

DESCRIPTION

The C and Pascal compilers link in *crt0.o*, *gcrt0.o*, or *mcrt0.o* to provide startup capabilities and environment for program execution. All are identical except that *gcrt0.o* and *mcrt0.o* provide additional functionality for *gprof*(1) and *prof*(1) profiling support respectively. Similarly, the Fortran compiler will link in either *frt0.o*, *gfrt0.o*, or *mfrt0.o*.

The following symbols are defined in these routines:

__argc_value A variable of type *int* containing the number of arguments.

__argv_value An array of character pointers to the arguments themselves.

_environ An array of character pointers to the environment in which the program will run. This array is terminated by a null pointer.

DEPENDENCIES

Series 300

The symbols above are shown as they are visible from C. To access them from assembly language, add an additional underscore to the beginning of the symbol. For example, an assembly language program will refer to **__argc_value** as **___argc_value**.

Series 300 startup files also define the following symbols which are listed as when used from assembly language. The state of these variables can be determined from C by using other library routines (see *is_hw_present*(3C)).

flag_68010 A variable of type *short*. Non-zero if the processor is a 68010; zero if not.

float_soft A variable of type *short*. Zero if the HP 98635 floating-point card is present; non-zero if it is not present.

float_loc A constant defining the location in memory of the HP 98635 floating point card.

flag_68881 A variable of type *short*. Non-zero if the HP 68881 floating point coprocessor is present; zero if it is not present.

flag_fpa A variable of type *short*. Non-zero if the HP 98248 floating point card is present; zero if it is not present.

fpa_loc A constant defining the location in memory of the HP 98248 floating point card.

Series 800

All compilers on the Series 800 use the *crt0.o* , *gcrt0.o*, or *mcrt0.o* file; the files *frt0.o*, *gfrt0.o*, and *mfrt0.o* do not exist.

The Series 800 startup files also define the following additional symbols:

$START$ Execution start address.

_start A secondary startup routine for C programs, called from **$START$**, which in turn calls **main**. This routine is contained in the C library rather than the *crt0.o* file. For Pascal and FORTRAN programs, this symbol labels the beginning of the outer block (main program) and is generated by the compilers.

$global$ The beginning address of the program's data area. The startup code loads this address into general register 27.

$UNWIND_START The beginning of the stack unwind table.

$UNWIND_END The end of the stack unwind table.

$RECOVER_START The beginning of the try/recover table.

$RECOVER_END The end of the try/recover table.

The *crt0.o* file defines a null procedure for **_mcount**, so programs compiled with profiling can be linked without profiling.

ORIGIN

AT&T System III

SEE ALSO

cc(1), f77(1), ld(1), pc(1), prof(1), gprof(1), pc(1), profil(2), exec(2), monitor(3C), is_hw_present(3C).

NAME

crypt, setkey, encrypt – generate hashing encryption

SYNOPSIS

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

void setkey (key)
char *key;

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

DESCRIPTION

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

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

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

The argument to the *encrypt* entry is a character array of length 64 containing only the characters with numerical value 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the hashing algorithm using the key set by *setkey*. *Fake* is not used and is ignored, but should be present if *lint*(1) is used.

SEE ALSO

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

BUGS

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

STANDARDS CONFORMANCE

crypt: SVID2, XPG2, XPG3

encrypt: SVID2, XPG2, XPG3

setkey: SVID2, XPG2, XPG3

NAME

ctermid – generate file name for terminal

SYNOPSIS

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

DESCRIPTION

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

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

NOTES

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

SEE ALSO

ttyname(3C).

STANDARDS CONFORMANCE

ctermid: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

ctime, localtime, gmtime, mktime, difftime, asctime, timezone, daylight, tzname, tzset, nl_ctime, nl_cxtime, nl_asctime, nl_ascxtime – convert date and time to string

SYNOPSIS

#include <time.h>

char *ctime (timer)
const time_t *timer;

char *nl_cxtime (timer, format)
const time_t *timer;
const char *format;

char *nl_ctime (timer, format, langid)
const time_t *timer; const char *format;
int langid;

struct tm *localtime (timer)
const time_t *timer;

struct tm *gmtime (timer)
const time_t *timer;

double difftime (time1, time0)
time_t time1, time0;

time_t mktime (timeptr)
struct tm *timeptr;

char *asctime (timeptr)
const struct tm *timeptr;

char *nl_ascxtime (timeptr, format)
const struct tm *timeptr;
const char *format;

char *nl_asctime (timeptr, format, langid)
const struct tm *timeptr;
const char *format;
int langid;

void tzset ( )

extern time_t timezone;

extern int daylight;

extern char *tzname[2];

DESCRIPTION

Asctime converts the broken-down time contained in the structure pointed to by *timeptr* and returns a pointer to a 26-character string in the form:

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

All the fields have constant width.

Ctime converts the calendar time pointed to by *timer*, representing the time in seconds since the Epoch, and returns a pointer to the local time in the form of a string. It is equivalent to:

asctime(localtime(timer))

Localtime and *gmtime* return pointers to **tm** structures, described below. *Localtime* corrects for the time zone and any summer time zone adjustments (such as Daylight Savings Time in the

USA), according to the contents of the TZ environment variable (see Environment Variables below). *Gmtime* converts directly to Coordinated Universal Time (UTC), which is the time the HP-UX system uses.

Difftime returns the difference in seconds between two calendar times: *time1 - time0*.

Mktime converts the broken-down time, expressed as local time, in the structure pointed to by *timeptr* into a calendar time value with the same encoding as that of the values returned by *time*(2). The original values of the **tm_wday** and **tm_yday** components of the structure are ignored, and the original values of the other components are not restricted to the ranges indicated below. A positive or zero value for **tm_isdst** causes *mktime* initially to presume that Daylight Saving Time, respectively, is or is not in effect for the specified time. A negative value for **tm_isdst** causes *mktime* to attempt to determine whether Daylight Saving Time is in effect for the specified time. On successful completion, all the components are set to represent the specified calendar time, but with their values forced to the ranges indicated below; the final value of **tm_mday** is not set until **tm_mon** and **tm_year** are determined. *Mktime* returns the specified calendar time encoded as a value of type **time_t**. If the calendar time cannot be represented, the function returns the value **(time_t)-1** and sets errno to **ERANGE**. Note the value **(time_t)-1** also corresponds to the time 23:59:59 on Dec 31, 1969 (plus or minus time zone and Daylight Saving Time adjustments), thus it is necessary to check both the return value and errno to reliably detect an error condition.

The <**time.h**> header file contains declarations of all relevant functions and externals. It also contains the **tm** structure, which includes the following members:

```
int tm_sec;      /* seconds after the minute - [0,61] */
int tm_min;      /* minutes after the hour - [0,59] */
int tm_hour;     /* hours - [0,23] */
int tm_mday;     /* day of month - [1,31] */
int tm_mon;      /* month of year - [0,11] */
int tm_year;     /* years since 1900 */
int tm_wday;     /* days since Sunday - [0,6] */
int tm_yday;     /* days since January 1 - [0,365] */
int tm_isdst;    /* daylight savings time flag */
```

The value of **tm_isdst** is positive if a summer time zone adjustment such as Daylight Savings Time is in effect, zero if not in effect, and negative if the information is not available.

Tzset sets the values of the external variables *timezone*, *daylight* and *tzname* according to the contents of the TZ environment variable (independent of any time value). The functions *localtime*, *mktime*, *ctime*, *nl_ctime*, *nl_cxtime*, *asctime*, *nl_asctime*, *nl_ascxtime*, and *strftime*(3C) call *tzset* and use the values returned in the external variables described below for their operations. *Tzset* may also be called directly by the user.

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* is non-zero only if you have specified a summer time zone adjustment in your TZ environment variable. The external variable *tzname*[2] contains the local standard and local summer time zone abbreviations as specified by the TZ environment variable.

Nl_cxtime extends the capabilities of *ctime*. The *format* specification allows the date and time to be output in a variety of ways. *Format* uses the field descriptors and field width and precision specifications defined in *strftime*(3C). If the format is the null string, the D_T_FMT string defined by *langinfo*(5) is used. *Nl_cxtime* is provided for historical reasons only; its use is not recommended.

Nl_ctime performs in a manner similar to *nl_cxtime*, but effectively first calls *langinit* (see *nl_init*(3C)) to load the program's locale according to the language specified by *langid*. *Nl_ctime* also appends a newline to the formatted string. *Nl_ctime* is provided for historical reasons only; its use is not recommended.

Nl_ascxtime, like *nl_cxtime*, allows the date string to be formatted. However, like *asctime*, *nl_asctime* takes a pointer to a **tm** structure as its argument. *Nl_ascxtime* is provided for historical reasons only; its use is not recommended.

Nl_asctime performs like *nl_ascxtime*, but first calls *langinit* (see *nl_init*(3C)) to load the program's locale according to the language specified by *langid*. *Nl_asctime* also appends a newline to the formatted string. *Nl_asctime* is provided for historical reasons only; its use is not recommended.

EXTERNAL INFLUENCES

Locale

The LC_TIME category determines for the functions *nl_cxtime*, *nl_ctime*, *nl_ascxtime* and *nl_asctime* the characters to be substituted for the directives described in *strftime*(3C) as being from the locale. It also determines the default output format used when a null format string is supplied to these functions.

The LC_CTYPE category determines the interpretation of the bytes within *format* as single and/or multi-byte characters.

Environment Variables

The function *tzset* uses the contents of TZ to set the values of the external variables *timezone*, *daylight* and *tzname*. TZ also determines the time zone name substituted for the **%Z** and **%z** directives and the time zone adjustments performed by *localtime*, *mktime*, *ctime*, *nl_ctime* and *nl_cxtime*. Two methods for specifying a time zone within TZ are described in *environ*(5).

International Code Set Support

Single- and multi-byte character code sets are supported.

WARNINGS

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

The range of tm_sec ([0,61]) extends to 61 to allow for the occasional one or two leap seconds. However, the "seconds since the Epoch" value returned by *time*(2) and passed as the *timer* argument does not include accumulated leap seconds. The **tm** structure generated by *localtime* and *gmtime* will never reflect any leap seconds. Upon successful completion, *mktime* will force the value of the **tm_sec** component to the range [0,59].

The use of *strftime*(3C) is recommended in place of the *ctime*, *nl_cxtime*, *nl_ctime*, *asctime*, *nl_ascxtime*, and *nl_asctime* routines defined here.

AUTHOR

Ctime was developed by AT&T and HP.

SEE ALSO

time(2), nl_init(3C), setlocale(3C), strftime(3C), tztab(4), environ(5), hpnls(5), lang(5), langinfo(5).

STANDARDS CONFORMANCE

ctime: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

asctime: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

daylight: SVID2, XPG2, XPG3

difftime: ANSI C

gmtime: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

localtime: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C *mktime*: XPG3, POSIX.1, FIPS 151-1, ANSI C

nl_ascxtime: XPG2

nl_cxtime: XPG2

timezone: XPG2, XPG3

tzname: XPG2, XPG3, POSIX.1, FIPS 151-1

tzset: XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

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

SYNOPSIS

#include <ctype.h>

int isalpha (c)
int c;

. . .

DESCRIPTION

These functions classify character-coded integer values according to the rules of the coded character set identified by the last successful call to *nl_init*(3C). Each function is a predicate returning non-zero for true, zero for false.

If *nl_init*(3C) has not been called successfully, characters are classified according to the rules of the default ASCII 7-bit coded character set (see *nl_init*(3C)).

Isascii is defined on all integer values; the other functions are defined for the range **–1** (**EOF**) to **255**.

isalpha	*c* is a letter.
isupper	*c* is an uppercase letter.
islower	*c* is a lowercase letter.
isdigit	*c* is a decimal digit (in ASCII: characters [0-9]).
isxdigit	*c* is a hexadecimal digit (in ASCII: characters [0-9], [A-F] or [a-f]).
isalnum	*c* is an alphanumeric (letters or digits).
isspace	*c* is a character that creates "white space" in displayed text (in ASCII: space, tab, carriage return, new-line, vertical tab, and form-feed).
ispunct	*c* is a punctuation character (in ASCII: any printing character except the space character (040), digits, letters).
isprint	*c* is a printing character.
isgraph	*c* is a visible character (in ASCII: printing characters, excluding the space character (040)).
iscntrl	*c* is a control character (in ASCII: character codes less than 040 and the delete character (0177)).
isascii	*c* is any ASCII character code between 0 and 0177, inclusive.

DIAGNOSTICS

If the argument to any of these functions is outside the domain of the function, the result is undefined.

WARNING

These functions are supplied both as library functions and as macros defined in the <ctype.h> header. Normally, the macro versions will be used. To obtain the library function either use a #undef to remove the macro definition or, if compiling in ANSI C mode, enclose the function name in parenthesis or take its address. The following example will use the library functions for isalpha, isdigit, and isspace:

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

```
...
main()
{
        int (*ctype_func)();
        ...
        if ( isalpha(c) )
        ...
        if ( (isdigit)(c) )
        ...
        ctype_func = isspace;
        ...
}
```

EXTERNAL INFLUENCES

Locale

The LC_CTYPE category determines the classification of character type.

International Code Set Support

Single-byte character code sets are supported.

AUTHOR

Ctype was developed by AT&T and HP.

SEE ALSO

nl_init(3C), ascii(5).

STANDARDS CONFORMANCE

isalnum: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

isalpha: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

isascii: SVID2, XPG2, XPG3

iscntrl: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

isdigit: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

isgraph: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

islower: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

isprint: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

ispunct: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

isspace: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

isupper: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

isxdigit: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

curses – CRT screen handling and optimization package

SYNOPSIS

#include <curses.h>

cc [*flags*] *file* ... **–lcurses** [*libraries*]

DESCRIPTION

These routines provide a method for updating screens with reasonable optimization. To initialize *curses* routines, the *initscr()* routine must be called before calling any other routine that deals with windows and screens. The *endwin()* routine should be called before exiting. To get character-at-a-time input without echoing, (most interactive, screen oriented-programs need this) after calling *initscr()* the program should call *"nonl(); cbreak(); noecho();"*

The full *curses* interface permits manipulation of data structures called "windows", which can be thought of as two-dimensional arrays of characters representing all or part of a CRT screen. A default window called **stdscr** is supplied, and others can be created using **newwin**. Windows are referred to by variables declared **WINDOW** *, the type **WINDOW** is defined in **<curses.h>** to be a C structure. These data structures are manipulated by using functions described below, among which the most basic are **move**, and **addch**. (More general versions of these functions are included. Their names begin with 'w', allowing the programmer to specify a window. The routines not beginning with 'w' affect **stdscr**.) Then *refresh()* is called, telling the routines to make the user's CRT screen resemble **stdscr**.

Mini-Curses is a subset of curses which does not allow manipulation of more than one window. To invoke this subset, use **–DMINICURSES** as an option to the *cc*(1) command. This level is smaller and faster than full curses.

If the environment variable TERMINFO is defined, any program using curses will check for a local terminal definition before checking in the standard place. For example, if the standard place is **/usr/lib/terminfo**, and TERM is set to "vt100", the compiled file is normally found in **/usr/lib/terminfo/v/vt100**. (The "v" is copied from the first letter of "vt100" to avoid creation of huge directories.) However, if TERMINFO is set to **/usr/mark/myterms**, *curses* first checks **/usr/mark/myterms/v/vt100**, and if that fails, checks **/usr/lib/terminfo/v/vt100**. This is useful for developing experimental definitions, or when write permission in **/usr/lib/terminfo** is not available.

Functions

All routines listed here can be called when using the full curses. Those marked with an asterisk can be called when using Mini-Curses.

addch(ch)*	add a character to *stdscr* (Like putchar. Wraps to next line at end of line)
addstr(str)*	calls addch with each character in *str*
attroff(attrs)*	turn off attributes named
attron(attrs)*	turn on attributes named
attrset(attrs)*	set current attributes to *attrs*
baudrate( )*	current terminal speed
beep( )*	sound beep on terminal
box(win, vert, hor)	draw a box around edges of *win*. *vert* and *hor* are chars to use for vert. and hor. edges of box
clear( )	clear *stdscr*
clearok(win, bf)	clear screen before next redraw of *win*
clrtobot( )	clear to bottom of *stdscr*
clrtoeol( )	clear to end of line on *stdscr*
cbreak( )*	set cbreak mode

delay_output(ms)*	insert ms millisecond pause in output
delch()	delete a character
deleteln()	delete a line
delwin(win)	delete *win*
doupdate()	update screen from all wnooutrefresh
echo()*	set echo mode
endwin()*	end window modes
erase()	erase *stdscr*
erasechar()	return user's erase character
fixterm()	restore tty to "in curses" state
flash()	flash screen or beep
flushinp()*	throw away any typeahead
getch()	get a char from tty
getstr(str)	get a string through *stdscr*
gettmode()	establish current tty modes
getyx(win, y, x)	get (y, x) co-ordinates
has_ic()	true if terminal can do insert character
has_il()	true if terminal can do insert line
idlok(win, bf)*	use terminal's insert/delete line if bf != 0
inch()	get char at current (y, x) co-ordinates
initscr()*	initialize screens
insch(c)	insert a char
insertln()	insert a line
intrflush(win, bf)	interrupts flush output if bf is TRUE
keypad(win, bf)	enable keypad input
killchar()	return current user's kill character
leaveok(win, flag)	OK to leave cursor anywhere after refresh if flag!=0 for *win*; otherwise cursor must be left at current position.
longname()	return verbose name of terminal
meta(win, flag)*	allow meta characters on input if flag != 0
move(y, x)*	move to (y, x) on *stdscr*
mvaddch(y, x, ch)	move(y, x) then addch(ch)
mvaddstr(y, x, str)	similar...
mvcur(oldrow, oldcol, newrow, newcol)	low level cursor motion
mvdelch(y, x)	like delch, but move(y, x) first
mvgetch(y, x)	etc.
mvgetstr(y, x)	
mvinch(y, x)	
mvinsch(y, x, c)	
mvprintw(y, x, fmt, args)	
mvscanw(y, x, fmt, args)	
mvwaddch(win, y, x, ch)	
mvwaddstr(win, y, x, str)	
mvwdelch(win, y, x)	
mvwgetch(win, y, x)	
mvwgetstr(win, y, x)	
mvwin(win, by, bx)	
mvwinch(win, y, x)	
mvwinsch(win, y, x, c)	
mvwprintw(win, y, x, fmt, args)	
mvwscanw(win, y, x, fmt, args)	

newpad(nlines, ncols)	create a new pad with given dimensions
newterm(type, outfd, infd)	set up new terminal of given type to output on outfd, using input (it needed) from infd
newwin(lines, cols, begin_y, begin_x)	create a new window
nl()*	set newline mapping
nocbreak()*	unset cbreak mode
nodelay(win, bf)	enable nodelay input mode through getch
noecho()*	unset echo mode
nonl()*	unset newline mapping
noraw()*	unset raw mode
overlay(win1, win2)	overlay win1 on win2
overwrite(win1, win2)	overwrite win1 on top of win2
pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)	like prefresh but with no output until doupdate called
prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)	refresh from pad starting with given upper left corner of pad with output to given portion of screen
printw(fmt, arg1, arg2, ...)	printf on *stdscr*
raw()*	set raw mode
refresh()*	make current screen look like *stdscr*
resetterm()*	set tty modes to "out of curses" state
resetty()*	reset tty flags to stored value
saveterm()*	save current modes as "in curses" state
savetty()*	store current tty flags
scanw(fmt, arg1, arg2, ...)	scanf through *stdscr*
scroll(win)	scroll *win* one line
scrollok(win, flag)	allow terminal to scroll if flag != 0
set_term(new)	now talk to terminal new
setscrreg(t, b)	set user scrolling region to lines t through b
setterm(type)	establish terminal with given type
setupterm(term, filenum, errret)	
standend()*	clear standout mode attribute
standout()*	set standout mode attribute
subwin(win, lines, cols, begin_y, begin_x)	create a subwindow
touchwin(win)	change all of *win*
traceoff()	turn off debugging trace output
traceon()	turn on debugging trace output
typeahead(fd)	use file descriptor fd to check typeahead
unctrl(ch)*	printable version of *ch*
waddch(win, ch)	add char to *win*
waddstr(win, str)	add string to *win*
wattroff(win, attrs)	turn off *attrs* in *win*
wattron(win, attrs)	turn on *attrs* in *win*
wattrset(win, attrs)	set attrs in *win* to *attrs*
wclear(win)	clear *win*
wclrtobot(win)	clear to bottom of *win*
wclrtoeol(win)	clear to end of line on *win*
wdelch(win, c)	delete char from *win*
wdeleteln(win)	delete line from *win*
werase(win)	erase *win*
wgetch(win)	get a char through *win*

wgetstr(win, str)	get a string through *win*
winch(win)	get char at current (y, x) in *win*
winsch(win, c)	insert char into *win*
winsertln(win)	insert line into *win*
wmove(win, y, x)	set current (y, x) co-ordinates on *win*
wnoutrefresh(win)	refresh but no screen output
wprintw(win, fmt, arg1, arg2, ...)	printf on *win*
wrefresh(win)	make screen look like *win*
wscanw(win, fmt, arg1, arg2, ...)	scanf through *win*
wsetscrreg(win, t, b)	set scrolling region of *win*
wstandend(win)	clear standout attribute in *win*
wstandout(win)	set standout attribute in *win*

Terminfo Level Routines

These routines should be called by programs that need to deal directly with the *terminfo*(4) database. Due to the low level of this interface, its use is discouraged. Initially, *setupterm* should be called to define the set of terminal-dependent variables defined in *terminfo*(4). The header files <**curses.h**> and <**term.h**> should be included to get the definitions for these strings, numbers, and flags. Parameterized strings should be passed through *tparm* to instantiate them. All *terminfo*(4) strings (including the output of *tparm*) should be printed with *tputs* or *putp* . Before exiting, *resetterm* should be called to restore the tty modes. (Programs desiring shell escapes or suspending with control-Z can call *resetterm* before the shell is called and *fixterm* after returning from the shell.)

fixterm()	restore tty modes for terminfo use (called by setupterm)
resetterm()	reset tty modes to state before program entry
setupterm(term, fd, rc)	read in database. Terminal type is the character string *term*, all output is to HP-UX System file descriptor *fd*. A status value is returned in the integer pointed to by *rc*: 1 is normal. The simplest call would be **setupterm(0, 1, 0)** which uses all defaults.
tparm(str, p1, p2, ..., p9)	instantiate string str with parms p_i.
tputs(str, affcnt, putc)	apply padding info to string *str*. *affcnt* is the number of lines affected, or 1 if not applicable. *Putc* is a putchar-like function to which the characters are passed, one at a time.
putp(str)	a handy function that calls tputs (str, 1, putchar)
vidputs(attrs, putc)	output the string to put terminal in video attribute mode *attrs*, which is any combination of the attributes listed below. Chars are passed to putchar-like function *putc*.
vidattr(attrs)	Like vidputs but outputs through putchar
set_curterm(term)	set the database pointed to by term
del_curterm(term)	free the space pointed to by term

Termcap Compatibility Routines

These routines were included as a conversion aid for programs that use termcap. Calling parameters are the same as for termcap. They are emulated using the *terminfo*(4) database. Their use in new software is not recommended because they may be deleted in future HP-UX releases.

tgetent(bp, name)	look up termcap entry for name
tgetflag(id)	get boolean entry for id
tgetnum(id)	get numeric entry for id
tgetstr(id, area)	get string entry for id
tgoto(cap, col, row)	apply parms to given cap
tputs(cap, affcnt, fn)	apply padding to cap calling fn as putchar

Attributes

The following video attributes can be passed to the functions *attron,attroff,attrset*.

A_STANDOUT	Terminal's best highlighting mode
A_UNDERLINE	Underlining
A_REVERSE	Reverse video
A_BLINK	Blinking
A_DIM	Half bright
A_BOLD	Extra bright or bold
A_BLANK	Blanking (invisible)
A_PROTECT	Protected
A_ALTCHARSET	Alternate character set

NLS Attributes

The following NLS attributes might be returned by *inch*:

A_FIRSTOF2	First byte of 16-bit character
A_SECOF2	Second byte of 16-bit character

Function Keys

The following function keys may be returned by *getch* if *keypad* has been enabled. Note that not all of these are currently supported due to lack of definitions in *terminfo* or the terminal not transmitting a unique code when the key is pressed.

Name	*Value*	*Key name*
KEY_BREAK	0401	break key (unreliable)
KEY_DOWN	0402	The four arrow keys ...
KEY_UP	0403	
KEY_LEFT	0404	
KEY_RIGHT	0405	
KEY_HOME	0406	Home key (upward+left arrow)
KEY_BACKSPACE	0407	backspace (unreliable)
KEY_F0	0410	Function keys. Space reserved for up to 64 keys.
KEY_F(n)	(KEY_F0+(n))	Formula for fn.
KEY_DL	0510	Delete line
KEY_IL	0511	Insert line
KEY_DC	0512	Delete character
KEY_IC	0513	Insert char or enter insert mode
KEY_EIC	0514	Exit insert char mode
KEY_CLEAR	0515	Clear screen
KEY_EOS	0516	Clear to end of screen
KEY_EOL	0517	Clear to end of line
KEY_SF	0520	Scroll 1 line forward
KEY_SR	0521	Scroll 1 line backwards (reverse)
KEY_NPAGE	0522	Next page
KEY_PPAGE	0523	Previous page
KEY_STAB	0524	Set tab
KEY_CTAB	0525	Clear tab

KEY_CATAB	0526	Clear all tabs
KEY_ENTER	0527	Enter or send (unreliable)
KEY_SRESET	0530	soft (partial) reset (unreliable)
KEY_RESET	0531	reset or hard reset (unreliable)
KEY_PRINT	0532	print or copy
KEY_LL	0533	home down or bottom (lower left)

WARNINGS

The plotting library *plot*(3X) and the curses library *curses*(3X) both use the names erase() and move(). The *curses* versions are macros. If you need both libraries, put the *plot*(3X) code in a different source file than the *curses*(3X) code, and/or #undef move() and erase() in the *plot*(3X) code.

HP supports only terminals listed on the current list of supported devices. However, non-supported and supported terminals can be in the *terminfo*(4) database. If you use such unsupported terminals, they may not work correctly.

The **endwin** routine does not release memory allocated by the **initscr** routine. Repeated calls to **initscr** can cause a program to use more memory than was intended.

Some of these routines call *malloc*(3C) or *malloc*(3X) to allocate memory, and can therefore fail for any of the reasons described in the corresponding manual entries.

SEE ALSO

terminfo(4).

Using Curses and Terminfo, tutorial in *HP-UX Concepts and Tutorials: Device I/O and User Interfacing.*

STANDARDS CONFORMANCE

curses: SVID2, XPG2, XPG3

NAME

cuserid – get character login name of the user

SYNOPSIS

#include <stdio.h>

char *cuserid (s)
char *s;

DESCRIPTION

Cuserid generates a character-string representation of the user name corresponding to the effective user **ID** of the 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, *cuserid* returns a NULL pointer; if *s* is not a NULL pointer, a null character **(\0)** will be placed at *s*[0].

SEE ALSO

geteuid(2), getlogin(3C) getpwuid(3C).

STANDARDS CONFORMANCE

cuserid: XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

cvtnum – convert string to floating point number

SYNOPSIS

#include <cvtnum.h>

int cvtnum(src,dst,typ,rnd,ptr,inx)
unsigned char *src,*dst,ptr;**
int typ,rnd,*inx;

DESCRIPTION

The function *cvtnum* converts an ASCII character string to a number in one of four floating point formats: single precision, double precision, extended precision, or packed decimal string.

The string pointed to by *src* is the string representation of a standard number, an infinity, or a not-a-number. A standard number begins with an optional sign followed by a string of digits optionally containing a decimal point. It may then have an optional **e** or **E** followed by an optional sign followed by an integer. Infinities are represented by **INF** preceded by an optional sign. The string for a not-a-number is an optional sign followed by **NaN** followed by any number of hexadecimal digits enclosed in parentheses.

The result is moved to *dst* and will be of the size and format as defined for the 68881 floating-point coprocessor.

typ indicates the type of conversion to be done. It may be one of four values: **C_SNGL**, **C_DBLE**, **C_EXT**, or **C_DPACK** indicating single precision, double precision, extended precision and packed decimal string respectively.

rnd specifies the type of rounding mode and may be one of four values: **C_NEAR**, **C_POS_INF**, **C_NEG_INF**, or **C_TOZERO** indicating round to nearest, to positive infinity, to negative infinity and to zero respectively.

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

If *inx* is not (int *)NULL, *cvtnum* will use this to return an indication of the inexactness of the conversion. A zero indicates exact; a non-zero value, inexact.

SEE ALSO

scanf(3S), strtod(3C), strtol(3C)
MC68881 Floating-Point Coprocessor User's Manual

DIAGNOSTICS

If no errors occur or no non-standard conversions are done, *cvtnum* returns 0. Otherwise, it will return one of the following:

C_BADCHAR - Illegal character or unexpected end of string
C_OVER - Overflow
C_UNDER - Underflow
C_INF - Infinity
C_QNAN - Quiet NaN
C_SNAN - Signalling NaN

NAME

datalock – lock process into memory after allocating data and stack space

SYNOPSIS

#include <sys/lock.h>
int datalock (datsiz, stsiz);
int datsiz, stsiz;

DESCRIPTION

Datalock allocates at least *datsiz* bytes of data space and *stsiz* bytes of stack space, then locks the program in memory. The data space is allocated with either *malloc*(3C) or *malloc*(3X) (whichever is linked with the program). After the program is locked, this space is released with *free* (on *malloc*(3C)) or *free* (on *malloc*(3X)), making it available for use. This allows the calling program to use that much space dynamically without receiving the *SIGSEGV* signal.

The effective user ID of the calling process must be super-user or be a member of or have an effective group ID of a group having PRIV_MLOCK access to use this call (see *getprivgrp*(2)).

EXAMPLES

The following call to *datalock* allocates 4096 bytes of data space and 2048 bytes of stack space and then locks the process in memory:

```
datalock (4096, 2048);
```

RETURN VALUE

Returns −1 if *malloc* cannot allocate enough memory or *plock*(2) returned an error.

WARNINGS

Multiple datalocks may not be the same as one big one.

Methods for calculating the required size are not yet well developed.

AUTHOR

Datalock was developed by the Hewlett-Packard Company.

SEE ALSO

getprivgrp(2), plock(2).

NAME

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

SYNOPSIS

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

dbminit(file)
char *file;

datum fetch(key)
datum key;

store(key, content)
datum key, content;

delete(key)
datum key;

datum firstkey()

datum nextkey(key)
datum key;

dbmclose()
```

DESCRIPTION

These functions maintain key/content pairs in a data base. The functions will handle very large (a billion blocks (block = 1024 bytes)) databases and will locate a keyed item in one or two file system accesses. This package is superseded by the newer *ndbm*(3X) library, which manages multiple databases. The functions can be accessed by giving the **–ldbm** option to *ld*(1) or *cc*(1).

Key and *content* parameters are described by the **datum** type. 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 of keys and has **.dir** as its suffix. The second file contains all data and has **.pag** as its suffix.

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

Once open, the data stored under a key is accessed by *fetch* and data is placed under a key by *store*. Storing data on an existing key will replace the existing data. 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))
```

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

DIAGNOSTICS

All functions that return an *int* indicate errors with negative values and success with zero. Routines that return a *datum* indicate errors with a null *dptr.*

WARNINGS

The **.pag** file will contain holes so that its apparent size is about four times its actual content.

Some older UNIX systems create real file blocks for these holes when touched. These files cannot be copied by normal means (such as *cp*(1), *cat*(1), *tar*(1), or *ar*(1)) without expansion.

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 if 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.

AUTHOR

Dbm(3X) was developed by the University of California, Berkeley.

SEE ALSO

ndbm(3X).

NAME

dial, undial – 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;    /*Will hold the name of the device used
                                   to make a connection */
    int             dev_len;    /* The length of the device used to
                                   make connection */
} CALL;
```

The CALL element *speed* is intended only for use with an outgoing dialed call, in which case its value should be either 300 or 1200 to identify the 113A modem, or the high- or low-speed setting on the 212A modem. Note that the 113A modem or the low-speed setting of the 212A modem will transmit at any rate between 0 and 300 bits per second. However, the high-speed setting of the 212A modem transmits and receivers 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* 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 need not be specified as it will be determined 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 symbols described below. The termination symbol will be supplied by the *dial* function, and should not be included in the *telno* string passed to *dial* in the CALL structure.

Permissible codes

0-9	dial 0-9
* or :	dial *
# or ;	dial #
-	4-second delay for second dial tone
e or <	end-of-number
w or =	wait for secondary dial tone

f flash off hook for 1 second

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

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

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	/* automatic call unit (acu) problem (open() failure) */
L_PROB	−6	/* line problem (open() failure) */
NO_Ldv	−7	/* can't open LDEVS file */
DV_NT_A	−8	/* requested device not available */
DV_NT_K	−9	/* requested device not known */
NO_BD_A	−10	/* no device available at requested baud */
NO_BD_K	−11	/* no device known at requested baud */

WARNINGS

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

The above routine uses <*stdio.h*>, which causes unexpected increases in the size of programs, not otherwise using standard I/O.

DEPENDENCIES

HP Clustered Environment

Dial is not supported on client nodes of an HP Cluster.

Series 300

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*(1) 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 reads should be checked for **(errno==EINTR)**, and the read possibly reissued.

FILES

/usr/lib/uucp/Devices
/usr/spool/uucp/LCK..*tty-device*

SEE ALSO

uucp(1), alarm(2), read(2), write(2), termio(7).

UUCP, a tutorial in *HP-UX Concepts and Tutorials.*

NAME

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

SYNOPSIS

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

DIR *opendir(dirname)
char *dirname;

struct dirent *readdir(dirp)
DIR *dirp;

long telldir(dirp)
DIR *dirp;

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

void rewinddir(dirp)
DIR *dirp;

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

DESCRIPTION

This library package provides functions that allow programs to read directory entries without having to know the actual directory format associated with the file system. Because these functions allow programs to be used portably on file systems with different directory formats, this is the recommended way to read directory entries.

Opendir opens the directory *dirname* and associates a directory stream with it. *Opendir* returns a pointer used to identify the directory stream in subsequent operations. The *opendir* routine allocates memory using *malloc*(3C) or *malloc*(3X), depending on which is linked with the program.

Readdir returns a pointer to the next directory entry. It returns a NULL pointer upon reaching the end of the directory or detecting an invalid *seekdir* operation. See *dirent*(5) for a description of the fields available in a directory entry.

Telldir returns the current location (encoded) associated with the directory stream to which *dirp* refers.

Seekdir sets the position of the next *readdir* operation on the directory stream to which *dirp* refers. The *loc* argument is a location within the directory stream obtained from *telldir*. The position of the directory stream is restored to where it was when *telldir* returned that *loc* value. Values returned by *telldir* are valid only while the **DIR** pointer from which they are derived remains open. If the directory stream is closed and then reopened, the *telldir* value might be invalid.

Rewinddir resets the position of the directory stream to which *dirp* refers 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 have done.

Closedir closes the named directory stream and then frees the structure associated with the **DIR** pointer.

RETURN VALUE

Upon successful completion, *opendir* returns a pointer to an object of type **DIR** referring to an open directory stream. Otherwise, it returns a NULL pointer and sets the global variable **errno** to indicate the error.

Upon successful completion, *readdir* returns a pointer to an object of type **struct dirent** describing a directory entry. Upon reaching the end of the directory, *readdir* returns a NULL pointer and does not change the value of **errno**. Otherwise, it returns a NULL pointer and sets **errno** to indicate the error.

Upon successful completion, *telldir* returns a long value indicating the current position in the directory. Otherwise it returns **−1** and sets **errno** to indicate the error.

Upon successful completion, *closedir* returns a value of **0**. Otherwise, it returns a value of **−1** and sets **errno** to indicate the error.

ERRORS

Opendir might fail if any of the following is true:

[EACCES]	Search permission is denied for a component of *dirname*, or read permission is denied for *dirname*.
[EFAULT]	*Dirname* points outside the allocated address space of the process. The reliable detection of this error is implementation dependent.
[ELOOP]	Too many symbolic links were encountered in translating the path name.
[EMFILE]	Too many open file descriptors are currently open for the calling process.
[ENAMETOOLONG]	A component of *dirname* exceeds PATH_MAX bytes, or the entire length of *dirname* exceeds PATH_MAX − 1 bytes while _POSIX_NO_TRUNC is in effect.
[ENFILE]	Too many open file descriptors are currently open on the system.
[ENOENT]	A component of the *dirname* does not exist.
[ENOMEM]	The *malloc* routine failed to provide sufficient memory to process the directory.
[ENOTDIR]	A component of *dirname* is not a directory.
[ENOENT]	The *dirname* argument points to an empty string.

Readdir might fail if any of the following is true:

[EBADF]	The *dirp* argument does not refer to an open directory stream.
[ENOENT]	The directory stream to which *dirp* refers is not located at a valid directory entry.
[EFAULT]	*dirp* points outside the allocated address space of the process.

Telldir might fail if the following is true:

[EBADF]	The *dirp* argument does not refer to an open directory stream.

Closedir might fail if the following is true:

[EBADF]	The *dirp* argument does not refer to an open directory stream.
[EFAULT]	*dirp* points outside the allocated address space of the process.

Rewinddir might fail if the following is true:

[EFAULT]	*dirp* points outside the allocated address space of the process.

EXAMPLES

The following code searches the current directory for an entry *name*:

```
DIR *dirp;
struct dirent *dp;
```

```
dirp = opendir(".");
while ((dp = readdir(dirp)) != NULL) {
        if (strcmp(dp->d_name, name) == 0) {
                (void) closedir(dirp);
                return FOUND;
        }
}
(void) closedir(dirp);
return NOT_FOUND;
```

WARNINGS

Readdir or *getdirentries*(2) are the only ways to access remote NFS directories. Attempting to read a remote directory using *read*(2) with NFS returns **−1** and sets **errno** to EISDIR.

APPLICATION USAGE

The header file required for these functions and the type of the return value from the *readdir* function has been changed for compatibility with System V Release 3 and the *X/Open Portability Guide*. See *ndir*(5) for a description of the header file **<ndir.h>**, which is provided to allow existing HP−UX applications to compile unmodified.

New applications should use the **<dirent.h>** header file for portability to System V and X/Open systems.

AUTHOR

Directory was developed by AT&T, HP, and the University of California, Berkeley.

SEE ALSO

close(2), getdirentries(2), lseek(2), open(2), read(2), dir(4), dirent(5), ndir(5).

STANDARDS CONFORMANCE

closedir: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

opendir: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

readdir: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

rewinddir: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

seekdir: XPG2, XPG3

telldir: XPG2, XPG3

NAME

div, ldiv – integer division and remainder

SYNOPSIS

#include <stdlib.h>

div_t div (numer, denom)
int numer, denom;

ldiv_t ldiv (numer, denom)
long int numer, denom;

DESCRIPTION

The *div* function computes the quotient and remainder of the division of the numerator *numer* by the denominator *denom*. If the division is inexact, the sign of the resulting quotient is that of the algebraic quotient, and the magnitude of the resulting quotient is the largest integer less than the magnitude of the algebraic quotient. If the result can be represented, the result is returned in a structure of type div_t (defined in stdlib.h) having members *quot* and *rem* for the quotient and remainder respectively. Both members have type int and values such that *quot* * *denom* + *rem* = *numer*. If the result cannot be represented, the behavior is undefined.

The *ldiv* function is similar to the *div* function, except that the arguments each have type long int and the result is returned in a structure of type ldiv_t (defined in stdlib.h) having long int members *quot* and *rem* for the quotient and remainder respectively.

WARNINGS

The behavior is undefined if *denom* is 0.

SEE ALSO

floor(3M).

STANDARDS CONFORMANCE

div: ANSI C

ldiv: ANSI C

NAME

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

SYNOPSIS

```
double drand48 ( )

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

long lrand48 ( )

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

long mrand48 ( )

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

void srand48 (seedval)
long seedval;

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

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

DESCRIPTION

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

In the following discussion, the formal mathematical notation [0.0, 1.0) indicates an interval including 0.0 but not including 1.0.

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

$$X_{n+1} = (aX_n + c) \bmod m \qquad n \geq 0$$

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

$$a = 5DEECE66D_{16} = 273673163155_8$$
$$c = B_{16} = 13_8.$$

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; that is why they must be initialized prior to being invoked. The functions *erand48, nrand48* and *jrand48* require the calling program to provide storage for the successive X_i values in the array specified as an argument when the functions are invoked. That is why these routines do not have to be initialized; the calling program merely has to place the desired initial value of X_i into the array and pass it as an argument. By using different arguments, functions *erand48, nrand48* and *jrand48* allow separate modules of a large program to generate several *independent* streams of pseudo-random numbers, 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).

STANDARDS CONFORMANCE

drand48: SVID2, XPG2, XPG3

erand48: SVID2, XPG2, XPG3

jrand48: SVID2, XPG2, XPG3

lcong48: SVID2, XPG2, XPG3

lrand48: SVID2, XPG2, XPG3

mrand48: SVID2, XPG2, XPG3

nrand48: SVID2, XPG2, XPG3

seed48: SVID2, XPG2, XPG3

srand48: SVID2, XPG2, XPG3

NAME

ecvt, fcvt, gcvt, nl_gcvt – convert floating-point number to string

SYNOPSIS

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

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

char *gcvt (value, ndigit, buf)
double value;
int ndigit;
char *buf;

char *nl_gcvt (value, ndigit, buf, langid)
double value;
int ndigit;
char *buf;
int langid;

DESCRIPTION

Ecvt converts *value* to a null-terminated string of *ndigit* digits and returns a pointer to the string. The high-order digit is non-zero, unless the value is zero. The low-order digit is rounded. The position of the radix character relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). The radix character 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.

One of three non-digit characters strings could be returned if the converted value is out of range. A "--" or "++" is returned if the value is larger than the exponent can contain, and is negative, or positive, respectively. The third string is returned if the number is illegal, a zero divide for example. The result value is Not A Number (NAN) and would return a "?" character.

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

Gcvt converts the *value* to a null-terminated string in the array pointed to by *buf* and returns *buf*. It produces *ndigit* significant digits in FORTRAN F-format if possible, or E-format otherwise. A minus sign, if required, and a radix character will be included in the returned string. Trailing zeros are suppressed. The radix character is determined by the currently loaded NLS environment (see *setlocale*(3C)). If *setlocale* has not been called successfully, the default NLS environment, "C", is used (see *lang*(5)). The default environment specifies a period (.) as the radix character.

Nl_gcvt differs from *gcvt* only by first calling *langinit* (see *nl_init*(3C)) to load the NLS environment according to the language specified by *langid*.

WARNINGS

The values returned by *ecvt* and *fcvt* point to a single static data array whose content is overwritten by each call.

Nl_gcvt is provided for historical reasons only; its use is not recommended.

EXTERNAL INFLUENCES

Locale

The LC_NUMERIC category determines the value of the radix character within the current NLS environment.

AUTHOR

Ecvt and *fcvt* were developed by AT&T. *Gcvt* was developed by AT&T and HP. *Nl_gcvt* was developed by HP.

SEE ALSO

setlocale(3C), printf(3S), hpnls(5), lang(5).

STANDARDS CONFORMANCE

ecvt: XPG2

fcvt: XPG2

gcvt: XPG2

NAME

end, etext, edata – last locations in program

SYNOPSIS

extern _end;
extern end;
extern _etext;
extern etext;
extern _edata;
extern edata;

DESCRIPTION

These names refer neither to routines nor to locations with interesting contents. The address of the symbols **_etext** and **etext** is the first address above the program text, the address of **_edata** and **edata** is the first address above the initialized data region, and the address of **_end** and **end** is the first address above the uninitialized data region.

The linker defines these symbols with the appropriate values if they are referenced by the program but not defined. The linker will issue an error if the user attempts to define **_etext**, **_edata**, or **_end**.

When execution begins, the program break (the first location beyond the data) coincides with **_end**, but the program break may be reset by the routines of *brk*(2), *malloc*(3C), standard input/output (*stdio*(3S)), the profile (**−p**) option of *cc*(1), and so on. Thus, the current value of the program break should be determined by **sbrk(0)** (see *brk*(2)).

WARNINGS

In C, these names must look like addresses. Thus, you would write **&end** instead of **end** to access the current value of *end*.

SEE ALSO

cc(1), ld(1), brk(2), malloc(3C), stdio(3S).

STANDARDS CONFORMANCE

end: XPG2

edata: XPG2

etext: XPG2

NAME

erf, erfc – error function and complementary error function

SYNOPSIS

```
#include <math.h>

double erf (x)
double x;

double erfc (x)
double x;
```

DESCRIPTION

Erf returns the error function of x, defined as $\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt$.

Erfc, which returns $1.0 - erf(x)$, is provided because of the extreme loss of relative accuracy if $erf(x)$ is called for large x and the result subtracted from 1.0 (for example, for $x = 5$, twelve places are lost).

DEPENDENCIES

Series 800 (/lib/libm.a and ANSI C /lib/libM.a)

Erf returns 1.0 when x is +INFINITY , or −1.0 when x is −INFINITY .

Erfc returns 0.0 when x is +INFINITY , or 2.0 when x is −INFINITY .

ERRORS

Series 800 (/lib/libm.a and ANSI C /lib/libM.a)

Erf and *erfc* return NaN and set **errno** to **EDOM** when x is NaN.

SEE ALSO

isinf(3M), isnan(3M), exp(3M).

STANDARDS CONFORMANCE

erf: SVID2, XPG2, XPG3

erfc: SVID2, XPG2, XPG3

NAME

exp, log, log10, pow, sqrt – exponential, logarithm, power, square root functions

SYNOPSIS

#include <math.h>

double exp (x)
double x;

double log (x)
double x;

double log10 (x)
double x;

double pow (x, y)
double x, y;

double sqrt (x)
double x;

DESCRIPTION

Exp returns e^x.

Log returns the natural logarithm of x. The value of x must be positive.

Log10 returns the logarithm base ten of x. The value of x must be positive.

Pow returns x^y. If x is 0.0, y must be positive. If x is negative, y must be an integer.

Sqrt returns the non-negative square root of x. The value of x must not be negative.

DEPENDENCIES

Series 300

The algorithms used are those from HP 9000 BASIC.

Series 800 (/lib/libm.a and ANSI C /lib/libM.a)

Exp returns:

- +INFINITY when x is +INFINITY ,
- 0.0 when x is –INFINITY .

Log and *log10* return +INFINITY when x is +INFINITY .

Pow returns +INFINITY when:

- Absolute value of x is greater than 1.0 and y is +INFINITY ,
- Absolute value of x is less than 1.0 and y is –INFINITY ,
- x is +INFINITY and y is greater than 0.0, or
- x is –INFINITY and y is an even integer.

Pow returns –INFINITY when x is –INFINITY and y is an odd integer.

Pow returns 0.0 when:

- Absolute value of x is greater than 1.0 and y is –INFINITY ,
- absolute value of x is less than 1.0 and y is +INFINITY ,
- x is +INFINITY and y is less than 0.0.

Sqrt returns +INFINITY when x is +INFINITY .

ERRORS

Series 300

Exp returns **HUGE_VAL** when the correct value would overflow, or 0.0 when the correct value would underflow, and sets **errno** to **ERANGE**.

Log and *log10* return −**HUGE_VAL** and set **errno** to **EDOM** when *x* is non-positive. A message indicating DOMAIN error (or SING error when *x* is 0.0) is printed on the standard error output.

Pow returns 0.0 and sets **errno** to EDOM when *x* is 0.0 and *y* is non-positive, or when *x* is negative and *y* is not an integer. In these cases a message indicating DOMAIN error is printed on the standard error output. When the correct value for *pow* would overflow or underflow, *pow* returns ±HUGE_VAL or 0.0 respectively, and sets **errno** to **ERANGE**.

Sqrt returns 0.0 and sets **errno** to **EDOM** when *x* is negative. A message indicating DOMAIN error is printed on the standard error output.

Series 800 (/lib/libm.a)

Exp returns **HUGE_VAL** when the correct value would overflow, or 0.0 when the correct value would underflow, and sets **errno** to **ERANGE**. NaN is returned and **errno** is set to **EDOM** when *x* is NaN.

Log and *log10* return −**HUGE_VAL** and set **errno** to **EDOM** when *x* is non-positive. NaN is returned and **errno** is set to **EDOM** when *x* is NaN or −INFINITY . A message indicating DOMAIN error (or SING error when *x* is 0.0) is printed on the standard error output in these cases.

Pow returns 0.0 and sets **errno** to **EDOM** when *x* is 0.0 and *y* is negative, or when *x* is negative and *y* is not an integer. NaN is returned and **errno** is set to **EDOM** when *x* or *y* is NaN. In these cases a message indicating DOMAIN error is printed on the standard error output. When the correct value for *pow* would overflow or underflow, *pow* returns ±HUGE_VAL or 0.0 respectively, and sets **errno** to **ERANGE**.

Sqrt returns NaN and sets **errno** to **EDOM** when *x* is negative, NaN or −INFINITY . A message indicating DOMAIN error is printed on the standard error output.

Series 800 (ANSI C /lib/libM.a)

No error messages are printed on the standard error output.

Exp returns HUGE_VAL when the correct value would overflow, or 0.0 when the correct value would underflow, and sets **errno** to ERANGE . NaN is returned and **errno** is set to EDOM when *x* is NaN.

Log and *log10* return NaN and set **errno** to **EDOM** when *x* is negative, −INFINITY, or NaN. −HUGE_VAL is returned and **errno** is set to EDOM when *x* is 0.0.

Pow returns 1.0 and sets **errno** to **EDOM** when x and y are both 0.0. **HUGE_VAL** is returned and **errno** is set to **EDOM** when *x* is 0.0 and *y* is negative. NaN is returned and **errno** is set to **EDOM** when *x* is negative and *y* is not an integer or when *x* or *y* is NaN. When the correct value for *pow* would overflow or underflow, *pow* returns ±HUGE_VAL or 0.0 respectively, and sets **errno** to ERANGE .

Sqrt returns NaN and sets **errno** to EDOM when *x* is negative, NaN or −INFINITY .

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

hypot(3M), isinf(3M), isnan(3M), matherr(3M), sinh(3M).

STANDARDS CONFORMANCE

exp: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C *log*: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

log10: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

pow: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

sqrt: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

fclose, fflush – close or flush a stream

SYNOPSIS

#include <stdio.h>

int fclose (stream)
FILE *stream;

int fflush (stream)
FILE *stream;

DESCRIPTION

Fclose causes any buffered data for the named *stream* to be written out, and the *stream* to be closed. Buffers allocated by the standard input/output system may be freed.

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

If *stream* points to an output stream or an update stream in which the most recent operation was output, *fflush* causes any buffered data for the *stream* to be written to that file; otherwise any buffered data is discarded. The *stream* remains open.

If *stream* is a null pointer, the *fflush* function performs this flushing action on all currently open streams.

DIAGNOSTICS

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

SEE ALSO

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

STANDARDS CONFORMANCE

fclose: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

fflush: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

ferror, feof, clearerr – stream status inquiries

SYNOPSIS

#include <stdio.h>

int ferror (stream)
FILE
***stream;**

int feof (stream)
FILE
***stream;**

void clearerr (stream)
FILE
***stream;**

DESCRIPTION

Ferror returns non-zero when an I/O error has previously occurred reading from or writing to the named *stream*, otherwise zero. Unless cleared by *clearerr*, or unless the specific *stdio* routine so indicates, the error indication lasts until the stream is closed.

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

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

WARNINGS

All these routines are implemented as both library functions and macros. The macro versions, which are used by default, are defined in <stdio.h>. To obtain the library function either use a #undef to remove the macro definition or, if compiling in ANSI-C mode, enclose the function name in parenthesis or use the function address. For following example illustrates each of these methods :

```
#include <stdio.h>
#undef ferror
...
main()
{
        int (*find_error()) ();
        ...
        return_val=ferror(fd);
        ...
        return_val=(feof)(fd1);
        ...
        find_error = feof;
};
```

SEE ALSO

open(2), fopen(3S).

STANDARDS CONFORMANCE

ferror: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

clearerr: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

feof: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

fgetpos, fsetpos – save and restore a file position indicator for a stream

SYNOPSIS

#include <stdio.h>

int fgetpos (stream, pos)
FILE *stream;
fpos_t *pos;

int fsetpos (stream, pos)
FILE *stream;
const fpos_t *pos;

DESCRIPTION

Fgetpos stores the current value of the file position indicator for the stream pointed to by *stream* in the object pointed to by *pos.* The value stored contains information usable by *fsetpos* for repositioning the stream to its position at the time of the call to *fgetpos.*

Fsetpos sets the file position indicator for the stream pointed to by *stream* according to the value of the object pointed to by *pos,* which shall be a value set by an earlier call to *fgetpos* on the same stream.

A successful call to *fsetpos* clears the end-of-file indicator for the stream and undoes any effects of *ungetc*(3S) on the same stream. After a *fsetpos* call, the next operation on a update stream may be either input or output.

RETURN VALUES

If successful, these functions return zero; otherwise non-zero.

WARNINGS

Failure may occur if these functions are used on a file that has not been opened via *fopen;* in particular, they may not be used on a terminal, or on a file opened via *popen*(3S).

SEE ALSO

fseek(3S), fopen(3S), popen(3S), ungetc(3S).

STANDARDS CONFORMANCE

fgetpos: ANSI C

NAME

fileno – map stream pointer to file descriptor

SYNOPSIS

#include <stdio.h>

int fileno (stream)
FILE
***stream;**

DESCRIPTION

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

The following symbolic values in <unistd.h> define the file descriptors associated with *stdin*, *stdout*, and *stderr* when a program is started :

STDIN_FILENO	Value of zero for standard input, *stdin*.
STDOUT_FILENO	Value of 1 for standard output, *stdout*.
STDERR_FILENO	Value of 2 for standard error, *stderr*.

DIAGNOSTICS

Upon error, *fileno* will return a -1.

SEE ALSO

open(2), fopen(3S).

STANDARDS CONFORMANCE

fileno: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

floor, ceil, fmod, fabs – floor, ceiling, remainder, absolute value functions

SYNOPSIS

```
#include <math.h>

double floor (x)
double x;

double ceil (x)
double x;

double fmod (x, y)
double x, y;

double fabs (x)
double x;
```

DESCRIPTION

Floor returns the largest integer (as a double-precision number) not greater than x.

Ceil returns the smallest integer not less than x.

Fmod returns the floating-point remainder (f) of the division of x by y, where f has the same sign as x, such that $x = iy + f$ for some integer i, and $|f| < |y|$.

Fabs returns the absolute value of x, $|x|$.

DEPENDENCIES

Series 300

Fmod returns x if y is 0.0 or if x/y would overflow.

Series 800 (/lib/libm.a)

When x is ±INFINITY , *floor* and *ceil* return ±INFINITY respectively.

Fabs returns +INFINITY when x is ±INFINITY .

Fmod returns x if y is 0.0, if x/y would overflow, or if x/y would underflow (including when y is ±INFINITY).

Series 800 (ANSI C /lib/LibM.a)

When x is ±INFINITY , *floor* and *ceil* return ±INFINITY respectively.

Fabs returns +INFINITY when x is ±INFINITY .

Fmod returns 0.0 if x/y would overflow, or x if x/y would underflow (including when y is ±INFINITY).

ERRORS

Series 800 (/lib/libm.a)

Floor and *ceil* return NaN and set **errno** to **EDOM** when x is NaN.

Fmod returns NaN and sets **errno** to **EDOM** when x or y is NaN, or when x is ±INFINITY .

Fabs returns NaN and sets **errno** to **EDOM** when x is NaN.

Series 800 (ANSI C /lib/libM.a)

Floor and *ceil* return NaN and set **errno** to **EDOM** when x is NaN.

Fmod returns NaN and sets **errno** to **EDOM** when y is 0.0, when x or y is NaN, or when x is ±INFINITY .

Fabs returns NaN and sets **errno** to **EDOM** when x is NaN.

SEE ALSO

abs(3C), isinf(3M), isnan(3M).

STANDARDS CONFORMANCE

floor: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

ceil: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

fabs: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

fmod: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

fopen, freopen, fdopen – open or re-open a stream file; convert file to stream

SYNOPSIS

#include <stdio.h>

FILE *fopen (file_name, type)
const char *file_name, *type;

FILE *freopen (file_name, type, stream)
const char *file_name, *type;
FILE *stream;

FILE *fdopen (filedes, type)
int filedes;
const char *type;

DESCRIPTION

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

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

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

"r"	open for reading
"w"	truncate to zero length or create for writing
"a"	append; open for writing at end of file, or create for writing
"rb"	open binary file for reading
"wb"	truncate to zero length or create binary file for writing
"ab"	append; open binary file for writing at end-of-file, or create binary file
"r+"	open for update (reading and writing)
"w+"	truncate to zero length or create for update
"a+"	append; open or create for update at end-of-file
"r+b" *or* "rb+"	open binary file for update (reading and writing)
"w+b" *or* "wb+"	truncate to zero length or create binary file for update
"a+b" *or* "ab+"	append; open or create binary file for update at end-of-file

Freopen substitutes the named file in place of the open *stream*. The original *stream* is closed, regardless of whether the open ultimately succeeds. *Freopen* returns a pointer to the FILE structure associated with *stream* and makes an implicit call to *clearerr* (see *ferror*(3S)).

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

Fdopen associates a stream with a file descriptor. File descriptors are obtained from *open*(2), *dup*(2), *creat*(2), or *pipe*(2), which open files but do not return pointers to a FILE structure stream. Streams are necessary input for many of the Section (3S) library routines. The *type* of stream must agree with the mode of the open file. The meanings of *type* used in the *fdopen* call are exactly as specified above, except that "w", "w+", "wb", and "wb+" do not cause truncation of the file.

When a file is opened for update, both input and output may be done on the resulting *stream*. However, output may not be directly followed by input without an intervening call to the *fflush* function or to a file positioning function (*fseek*, *fsetpos*, or *rewind*), and input may not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file.

When a file is opened for append (i.e., when *type* is "a" or "a+"), it is impossible to overwrite information already in the file. All output is written at the end of the file, regardless of intervening class to the *fseek* function. If two separate processes open the same file for append, each process can 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.

DIAGNOSTICS

Fopen and *freopen* return a NULL pointer if *file_name* cannot be accessed, if there are too many open files, or if the arguments are incorrect.

Fdopen returns a **NULL** upon failure.

NOTES

On HP-UX the binary file *types* are equivalent to their non-binary counterparts. For example, the "r" and "rb" types are equivalent.

SEE ALSO

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

STANDARDS CONFORMANCE

fopen: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

fdopen: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

freopen: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

fread, fwrite – buffered binary input/output to a stream file

SYNOPSIS

#include <stdio.h>

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

size_t fwrite (ptr, size, nitems, stream)
const char *ptr;
size_t size, nitems;
FILE *stream;

DESCRIPTION

Fread copies, into an array pointed to by *ptr, nitems* items of data from the named input *stream*, where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. *Fread* stops appending bytes if an end-of-file or error condition is encountered while reading *stream*, or if *nitems* items have been read. *Fread* leaves the file pointer in *stream*, if defined, pointing to the byte following the last byte read if there is one. *Fread* does not change the contents of *stream*.

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

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

SEE ALSO

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

DIAGNOSTICS

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

STANDARDS CONFORMANCE

fread: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

fwrite: SVID2, XPG2, XPG3

NAME

frexp, ldexp, modf – split floating-point into mantissa and exponent

SYNOPSIS

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

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

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

DESCRIPTION

Every non-zero number can be written uniquely as $x * 2^n$, where the "mantissa" (fraction) x is in the range $0.5 \leq |x| < 1.0$, and the "exponent" n is an integer.

Frexp returns the mantissa of a double *value*, and stores the exponent indirectly in the location pointed to by *eptr*. If *value* is zero, both results returned by *frexp* are zero.

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

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

DIAGNOSTICS

If *ldexp* would cause overflow, ±**HUGE** is returned (according to the sign of *value*), and **errno** is set to **ERANGE**.
If *ldexp* would cause underflow, zero is returned and **errno** is set to **ERANGE**.

STANDARDS CONFORMANCE

frexp: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

ldexp: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

modf: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

fseek, rewind, ftell – reposition a file pointer in a stream

SYNOPSIS

```
#include <stdio.h>

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

void rewind (stream)
FILE *stream;

long ftell (stream)
FILE *stream;
```

DESCRIPTION

Fseek sets the position of the next input or output operation on the *stream*. The new position, measured in bytes from the beginning of the file, is obtained by adding *offset* to the position specified by *ptrname*. The specified position is the beginning of the file for SEEK_SET, the current position for SEEK_CUR, or end-of-file for SEEK_END.

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

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

After *fseek* or *rewind*, the next operation on a file opened for update may be either input or output. *Fseek* clears the **EOF** indicator for the *stream*. *Rewind* does an implicit *clearerr* (on *ferror*(3S)) call.

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

DIAGNOSTICS

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

Ftell returns -1 for error conditions.

WARNING

Although on HP-UX 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 operating 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.

SEE ALSO

lseek(2), ferror(3S), fgetpos(3S), fopen(3S), fsetpos(3S), popen(3S), ungetc(3S).

STANDARDS CONFORMANCE

fseek: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

ftell: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

rewind: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

ftw, ftwh – walk a file tree

SYNOPSIS

```
#include <ftw.h>

int ftw (path, fn, depth)
char *path;
int (*fn) ( );
int depth;

int ftwh (path, fn, depth)
char *path;
int (*fn) ( );
int depth;
```

DESCRIPTION

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

Ftw visits a directory before visiting any of its descendants.

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

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

Ftwh performs the same function as *ftw* but *ftwh* also traverses hidden directories (context dependent files, see *cdf*(4)).

ERRORS

ftw( ) will fail if any of the following occurs:

[EACCES]	Search permission is denied for any component of *path*.
[ENAMETOOLONG]	The length of the specified path name exceeds PATH_MAX bytes, or the length of a component of the path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.
[ENOENT]	*Path* points to the name of a file that does not exist, or points to an empty string.
[ENOTDIR]	A component of *path* is not a directory.

ftw( ) may fail if:

[EINVAL]	The value of the *depth* argument is invalid.

In addition, if the function pointed to by *fn* encounters system errors, **errno** may be set accordingly.

WARNINGS

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

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

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.

AUTHOR

Ftw was developed by AT&T. *Ftwh* was developed by HP.

SEE ALSO

stat(2), malloc(3C), cdf(4).

STANDARDS CONFORMANCE

ftw: SVID2, XPG2, XPG3

NAME

gamma, lgamma, signgam – log gamma function

SYNOPSIS

```
#include <math.h>

double gamma (x)
double x;

double lgamma (x)
double x;

extern int signgam;
```

DESCRIPTION

Gamma returns $\ln(|\Gamma(x)|)$, where $\Gamma(x)$ is defined as $\int_0^\infty e^{-t} t^{x-1} dt$. The sign of $\Gamma(x)$ is returned in the external integer *signgam*. The argument x must not be a non-positive integer. (*Gamma* is defined over the reals excluding the non-positive integers).

The following C program fragment can be used to calculate Γ:

```
if ((y = gamma(x)) > LN_MAXDOUBLE)
        error( );
y = signgam * exp(y);
```

where if y is greater than LN_MAXDOUBLE, as defined in the *<values.h>* header file, *exp*(3M) will return a range error.

ERRORS

Series 300

For non-positive integer arguments *gamma* returns HUGE_VAL and sets **errno** to **EDOM**. A message indicating SING error is printed on the standard error output.

If the correct value would overflow, *gamma* returns HUGE_VAL and sets **errno** to **ERANGE**.

Series 800 (/lib/libm.a)

For non-positive integer arguments, *gamma* returns HUGE_VAL and sets **errno** to **EDOM**. A message indicating SING error is printed on the standard error output.

If the correct value would overflow, *gamma* returns HUGE_VAL and sets **errno** to **ERANGE**.

Gamma returns NaN and sets **errno** to **EDOM** when x is NaN, or returns +INFINITY and sets **errno** to **EDOM** when x is ±INFINITY . A message indicating DOMAIN error is printed on the standard error output.

Series 800 (ANSI C /lib/libM.a)

No error messages are printed on the standard error output.

For non-positive integer arguments *gamma* returns HUGE_VAL and sets **errno** to **EDOM**. A message indicating SING error is printed on the standard error output.

If the correct value would overflow, *gamma* returns HUGE_VAL and sets **errno** to **ERANGE**.

Gamma returns NaN and sets **errno** to **EDOM** when x is NaN, or returns +INFINITY and sets **errno** to **EDOM** when x is ±INFINITY .

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

exp(3M), isinf(3M), isnan(3M), matherr(3M), values(5).

STANDARDS CONFORMANCE

gamma: SVID2, XPG2, XPG3

signgam: SVID2, XPG2, XPG3

NAME

getc, getchar, fgetc, getw – get character or word from a stream file

SYNOPSIS

#include <stdio.h>

int getc (stream)
FILE *stream;

int getchar ()

int fgetc (stream)
FILE *stream;

int getw (stream)
FILE *stream;

DESCRIPTION

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

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. *int* in C) 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).

DIAGNOSTICS

These functions return the constant **EOF** at end-of-file or upon an error. Because **EOF** is a valid integer, *ferror*(3S) should be used to detect *getw* errors.

WARNING

The *getc* and *getchar* routines are implemented as both library functions and macros. The macro versions, which are used by default, are defined in <stdio.h>. To obtain the library function either use a #undef to remove the macro definition or, if compiling in ANSI-C mode, enclose the function name in parenthesis or use the function address. For following example illustrates each of these methods :

```
#include <stdio.h>
#undef getc
...
main()
{
        int (*get_char()) ();
        ...
        return_val=getc(c,fd);
        ...
        return_val=(getc)(c,fd1);
        ...
        get_char = getchar;
};
```

If the integer value returned by *getc*, *getchar*, or *fgetc* is stored into a character variable and then compared against the integer constant **EOF**, the comparison may never succeed, because sign-extension of a character on widening to integer is machine-dependent.

The macro version of *getc* incorrectly treats a *stream* argument with side effects. In particular, **getc(*f++)** does not work sensibly. The function version of *getc* or *fgetc* should be used instead.
Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent, and may not be read using *getw* on a different processor.

STANDARDS CONFORMANCE

getc: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

fgetc: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

getchar: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

getw: SVID2, XPG2, XPG3

NAME

getccent, getcccid, getccnam, setccent, endccent, fgetccent – get HP Cluster configuration entry

SYNOPSIS

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

struct cct_entry *getccent()

struct cct_entry *getcccid(cid)
cnode_t cid;

struct cct_entry *getccnam(name)
char *name;

void setccent()

void endccent()

struct cct_entry *fgetccent(f)
FILE *f;
```

DESCRIPTION

Getccent, *getcccid*, and *getccnam* each return a pointer to an object with the following structure containing the broken-out fields in the **/etc/clusterconf** file. The file contains a list of **cct_entry** structures, defined in the **<cluster.h>** header file. The **cct_entry** structure includes the following fields:

```
u_char machine_id[M_IDLEN];     /* Unique machine ID */
cnode_t cnode_id;               /* cnode ID */
char cnode_name[15];            /* cnode name */
char cnode_type;                /* 'r'=cluster server
                                   'c'=all other cluster nodes */
cnode_t swap_serving_cnode;     /* swap cnode */
int kcsp;                       /* default number of CSPs to create
                                   see csp(1M) */
```

The constant **M_IDLEN** is defined in **<cluster.h>** .

Getccent when first called opens the cluster configuration file **/etc/clusterconf** and returns a pointer to the first **cct_entry** structure in the file. Thereafter, it returns a pointer to the next **cct_entry** structure in the file. Successive calls can be used to search the entire file. *Getcccid* searches from the beginning of the file until an entry whose cnode ID matches *cid* is found and returns a pointer to the particular structure in which it was found. *Getccnam* searches from the beginning of the file until a cnode name matching *name* is found and returns a pointer to the particular structure in which it was found. If an **EOF** or an error is encountered on reading, these functions return a **NULL** pointer.

A call to *setccent* has the effect of rewinding the cluster configuration file to the beginning of the file to allow repeated searches. *Endccent* can be called to close the cluster configuration file when processing is complete.

Fgetccent returns a pointer to the next **cct_entry** structure in the stream *f*, which matches the format of **/etc/clusterconf**.

DIAGNOSTICS

A **NULL** pointer is returned on **EOF** or error.

WARNINGS

The above routines use <**stdio.h**>, which causes them to increase the size of programs not otherwise using standard I/O, more than might be expected.

All information is contained in a static area overwritten with each call; thus information must be copied if it is to be saved.

AUTHOR

Getccent was developed by HP.

FILES

/etc/clusterconf

SEE ALSO

csp(1M), clusterconf(4).

NAME

getcdf – return the expanded path that matches a path name

SYNOPSIS

char *getcdf (path, buf, size)
char *path;
char *buf;
int size;

DESCRIPTION

Getcdf returns a pointer to the expanded path matching the path name in *path*. The path name can be a context dependent file (CDF). If *path* is a CDF, a path name with all hidden directories expanded is returned. If *path* is not a CDF, the original path name is returned.

The value of *size* must be at least one greater than the length of the path name to be returned.

If *buf* is not a NULL pointer, *getcdf* copies the expanded path name into array *buf*. If *buf* is a NULL pointer, *getcdf* obtains *size* bytes of space using *malloc*(3C). In this case, the pointer returned by *getcdf* can be used as an argument in a subsequent call to *free (see malloc*(3C)).

DIAGNOSTICS

Returns **NULL** with **errno** set if *size* is not large enough, or the path name in *buf* does not exist or cannot be accessed.

EXAMPLES

```
char *path, *cdf, *getcdf();
int size;
.
.
.
if ((cdf = getcdf(path, NULL, size)) == NULL) {
        perror("getcdf");
        exit(1);
}
printf("%s\n", cdf);
```

AUTHOR

Getcdf was developed by HP.

SEE ALSO

showcdf(1), malloc(3C), cdf(4), context(5).

NAME

getcwd, gethcwd − get pathname of current working directory

SYNOPSIS

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

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

DESCRIPTION

Getcwd places the absolute pathname of the current working directory in the array pointed to by *buf*, and returns *buf*. The value of *size* must be at least one greater than the length of the pathname to be returned.

If *buf* is a NULL pointer, *getcwd* will obtain *size* bytes of space using *malloc*(3C). In this case, the pointer returned by *getcwd* may be used as the argument in a subsequent call to *free* (see *malloc*(3C)). Invoking *getcwd* with *buf* as a null pointer is not recommended as this functionality may be subject to later withdrawal.

Gethcwd works the same as *getcwd* except the returned directory pathname will list all hidden directories (context dependent files, see *cdf*(4)).

RETURN VALUE

Upon successful completion, *getcwd* returns a pointer to the current directory pathname. Otherwise, it returns **NULL** with *errno* set if *size* is not large enough, or if an error occurs in a lower-level function.

ERRORS

Getcwd will fail if any of the following is true:

[EINVAL]	The *size* argument is zero or negative.
[ERANGE]	The *size* argument is greater than zero, but is smaller than the length of the pathname.
[ENAMETOOLONG]	The length of the specified path name exceeds PATH_MAX bytes, or the length of a component of the path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.

Getcwd may fail if any of the following is true:

[EACCES]	Read or search permission is denied for a component of pathname.
[EFAULT]	*Buf* points outside the allocated address space of the process. *Getcwd* may not always detect this error.
[ENOMEM]	The *malloc* routine failed to provide *size* bytes of memory.

EXAMPLES

```
char *cwd, *getcwd();
char buf[PATH_MAX+1];
   .
   .
   .
if ((cwd = getcwd((buf *)NULL, PATH_MAX+1)) == NULL) {
        perror("pwd");
        exit(1);
```

```
        }
        puts(cwd);
```

AUTHOR

Getcwd was developed by AT&T. *Gethcwd* was developed by HP.

SEE ALSO

pwd(1), malloc(3C), cdf(4).

STANDARDS CONFORMANCE

getcwd: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

getenv – return value for environment name

SYNOPSIS

#include <stdlib.h>
char *getenv (name)
char *name;

DESCRIPTION

Getenv searches the environment list (see *environ*(5)) for a string of the form *name=value*, and returns a pointer to the *value* in the current environment if such a string is present, otherwise a NULL pointer. *Name* may be either the desired name, null-terminated, or of the form *name=value*, in which case *getenv* uses the portion to the left of the "=" as the search key.

WARNINGS

Getenv returns a pointer to static data which may be overwritten by subsequent calls.

SEE ALSO

exec(2), putenv(3C), environ(5).

EXTERNAL INFLUENCES

Locale

The LC_CTYPE category determines the interpretation of characters in *name* as single- and/or multi-byte characters.

International Code Set Support

Single- and multi-byte character code sets are supported.

STANDARDS CONFORMANCE

getenv: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

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

SYNOPSIS

#include <checklist.h>

struct checklist *getfsent()

struct checklist *getfsspec(spec)
char *spec;

struct checklist *getfsfile(file)
char *file;

struct checklist *getfstype(type)
char *type;

int setfsent()

int endfsent()

DESCRIPTION

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

Getfsent, *getfsspec*, *getfsfile*, and *getfstype* each returns a pointer to an object with the following structure containing the broken-out fields of a line in the **/etc/checklist** file. The structure is declared in the <**checklist.h**> header file:

```
struct checklist {
      char    *fs_spec;     /* special file name */
      char    *fs_bspec;    /* block special file name */
      char    *fs_dir;      /* file sys directory name */
      char    *fs_type;     /* type: ro, rw, sw, xx */
      int     fs_passno;    /* fsck pass number */
      int     fs_freq;      /* backup frequency */
};
```

The fields have meanings described in *checklist*(4). If the block special file name, the file system directory name, and the type are not all defined on the associated line in **/etc/checklist**, these routines will return pointers to NULL in the **fs_bspec**, **fs_dir** and **fs_type** fields. If the pass number or the backup frequency field are not present on the line, these routines will return **–1** in the corresponding structure member. **Fs_freq** is reserved for future use.

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 file system file name is found, or until EOF is encountered. *Getfstype* does likewise, matching on the file system type field.

DIAGNOSTICS

A null pointer is returned on EOF, invalid entry or error.

WARNINGS

Since all information is contained in a static area, it must be copied to be saved.

AUTHOR

Getfsent was developed by HP and the University of California, Berkeley.

FILES

/etc/checklist

SEE ALSO

checklist(4).

NAME

getgrent, getgrgid, getgrnam, setgrent, endgrent, fgetgrent – get group file entry

SYNOPSIS

#include <grp.h>

struct group *getgrent ( )

struct group *getgrgid (gid)
gid_t gid;

struct group *getgrnam (name)
char *name;

void setgrent ( )

void endgrent ( )

struct group *fgetgrent (f)
FILE *f;

DESCRIPTION

The *getgrent*, *getgrgid*, and *getgrnam* functions locate an entry in the **/etc/group** file, and return a pointer to an object of type **struct group**.

The **group** structure is defined in <**grp.h**> and includes the following members:

```
char    *gr_name;     /* the name of the group */
char    *gr_passwd;   /* the encrypted group password */
gid_t   gr_gid;       /* the numerical group ID */
char    **gr_mem;     /* null-terminated array of pointers to member names */
```

Getgrent when first called returns a pointer to the first **group** structure in the file; thereafter, it returns a pointer to the next **group** structure in the file. In this way, successive calls can be used to search the entire file. *Getgrent* opens the **/etc/group** file prior to doing its work and leaves the file open afterward; *setgrent* has the effect of rewinding this file to allow repeated searches; *endgrent* can be called to close the file when processing is complete.

Getgrgid searches from the beginning of the file until a numeric group ID matching *gid* is found, and returns a pointer to the particular structure in which it was found.

Getgrnam searches from the beginning of the file until a group name matching *name* is found, and returns a pointer to the particular structure in which it was found.

Fgetgrent returns a pointer to the next **group** structure in the standard I/O stream *f*, which should be open for reading, and its contents should match the format of **/etc/group**.

NETWORKING FEATURES

NFS

If an entry beginning with a plus sign (+) or a minus sign (-) is found, these routines try to use the Yellow Pages network database for data. See *group*(4) for proper syntax and operation.

RETURN VALUE

The *getgrent*, *getgrgid*, *getgrnam*, and *fgetgrent* functions return a **NULL** pointer if an end-of-file or error is encountered on reading. Otherwise, the return value points to an internal static area containing a valid **group** structure.

WARNINGS

The above routines use <**stdio.h**> and the Yellow Pages library. This causes them to increase the size of programs that do not otherwise use standard I/O and Yellow Pages more than might ordinarily be expected.

The value returned by these functions points to a single static area that is overwritten by each call to any of the functions. It must be copied if it is to be saved.

DEPENDENCIES

NFS

FILES

/etc/yp/domainname/group.byname
/etc/yp/domainname/group.bygid

SEE ALSO

ypcat(1) in *Networking Reference Manual*.

FILES

/etc/group

SEE ALSO

getgroups(2), getpwent(3C), stdio(3S), group(4).

STANDARDS CONFORMANCE

getgrent: SVID2, XPG2

endgrent: SVID2, XPG2

fgetgrent: SVID2, XPG2

getgrgid: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

getgrnam: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

setgrent: SVID2, XPG2

NAME

getlogin − get login name

SYNOPSIS

char *getlogin ( );

DESCRIPTION

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

If *getlogin* is called within a process that is not attached to a terminal, it returns a **NULL** pointer. The recommended procedure to obtain the user name associated with the real user ID of the calling process is to call *getlogin* , and if that fails to call *getpwuid*. The function *cuserid* may be used to obtain the user name associated with the effective user ID of the calling process.

ERRORS

Getlogin will fail if any of the following is true:

[EBADF]	An invalid file descriptor was obtained.
[EMFILE]	Too many file descriptors are in use by this process.
[ENFILE]	The system file table is full.

FILES

/etc/utmp

SEE ALSO

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

DIAGNOSTICS

Getlogin returns the **NULL** pointer if *name* is not found.

WARNINGS

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

STANDARDS CONFORMANCE

getlogin: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

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

SYNOPSIS

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

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

struct mntent *getmntent(filep)
FILE *filep;

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

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

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

DESCRIPTION

These routines replace the *getfsent* routines for accessing the file system description file **/etc/checklist**. 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*(3C). *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 *checklist*(4).

```
struct mntent {
        char    *mnt_fsname;    /* file system name */
        char    *mnt_dir;       /* file system path prefix */
        char    *mnt_type;      /* hfs, nfs, swap, or xx */
        char    *mnt_opts;      /* ro, suid, etc. */
        int     mnt_freq;       /* dump frequency, in days */
        int     mnt_passno;     /* pass number on parallel fsck */
        long    mnt_time;       /* When file system was mounted; */
                                  /* see mnttab(4). */
};
```

Addmntent adds the **mntent** structure *mnt* to the end of the open file *filep*. Note that *filep* must be opened for writing. *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.

The following definitions are provided in **<mntent.h>**:

```
#define MNT_CHECKLIST       "/etc/checklist"
#define MNT_MNTTAB          "/etc/mnttab"
```

```
#define MNTMAXSTR          128          /* Max size string in mntent */

#define MNTTYPE_HFS        "hfs"        /* HFS file system */
#define MNTTYPE_NFS        "nfs"        /* Network file system */
#define MNTTYPE_SWAP       "swap"       /* Swap device */
#define MNTTYPE_SWAPFS     "swapfs"     /* File system swap */
#define MNTTYPE_IGNORE     "ignore"     /* Ignore this entry */

#define MNTOPT_DEFAULTS    "defaults"   /* Use all default options */
#define MNTOPT_RO          "ro"         /* Read only */
#define MNTOPT_RW          "rw"         /* Read/write */
#define MNTOPT_SUID        "suid"       /* Set uid allowed */
#define MNTOPT_NOSUID      "nosuid"     /* No set uid allowed */
```

The following definitions are provided for file system swap in <**mntent.h**>:

```
#define MNTOPT_MIN   "min"   /* minimum file system swap */
#define MNTOPT_LIM   "lim"   /* maximum file system swap */
#define MNTOPT_RES   "res"   /* reserve space for file system */
#define MNTOPT_PRI   "pri"   /* file system swap priority */
```

NETWORKING FEATURES

NFS

The following definitions are provided in <**mntent.h**>:

```
#define MNTOPT_BG         "bg"       /* Retry mount in background */
#define MNTOPT_FG         "fg"       /* Retry mount in foreground */
#define MNTOPT_RETRY      "retry"    /* Number of retries allowed */
#define MNTOPT_RSIZE      "rsize"    /* Read buffer size in bytes */
#define MNTOPT_WSIZE      "wsize"    /* Write buffer size in bytes*/
#define MNTOPT_TIMEO      "timeo"    /* Timeout in 1/10 seconds */
#define MNTOPT_RETRANS    "retrans"  /* Number of retransmissions */
#define MNTOPT_PORT       "port"     /* Server's IP NFS port */
#define MNTOPT_SOFT       "soft"     /* Soft mount */
#define MNTOPT_HARD       "hard"     /* Hard mount */
#define MNTOPT_INTR       "intr"     /* Interruptable hard mounts */
#define MNTOPT_NOINTR     "nointr"   /* Uninterruptable hard mounts*/
#define MNTOPT_DEVS       "devs"     /* Device file access allowed */
#define MNTOPT_NODEVS     "nodevs"   /* No device file access allowed */
```

RETURN VALUE

Setmntent returns a null pointer on error. *Getmntent* returns a null pointer on error or EOF. Otherwise, *getmntent* returns a pointer to a mntent structure. Some of the fields comprising a mntent structure are optional in **/etc/checklist** and **/etc/mnttab**. In the supplied structure, such missing character pointer fields are set to NULL and missing integer fields are set to **−1**. *Addmntent* returns **1** on error. *Endmntent* returns **1**.

WARNINGS

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

AUTHOR

Addmntent, endmntent, getmntent, hasmntopt and *setmntent* were developed by The University of California, Berkeley, Sun Microsystems, Inc. and HP.

FILES

/etc/checklist
/etc/mnttab

SEE ALSO

checklist(4), getfsent(3X), mnttab(4).

NAME
getmsg – get message from a catalog

SYNOPSIS
char *getmsg (fildes, set_num, msg_num, buf, buflen)
int fildes, set_num, msg_num, buflen;
char buf[];

DESCRIPTION
Getmsg reads message *msg_num* in set *set_num* from the message catalog identified by *fildes*, a file descriptor returned from a previous call to *open*(2). The returned message is stored in *buf*, a buffer of size *buflen* bytes.

A message longer than *buflen*-1 bytes is silently truncated. The returned message is always terminated with a null byte.

RETURN VALUE
If successful, *getmsg* returns a pointer to the message in *buf*. Otherwise, if *fildes* is invalid or if *set_num* or *msg_num* is not in the catalog, *getmsg* returns a pointer to an empty string.

WARNINGS
This routine is provided for historical reasons only. Use of the equivalent routine *catgetmsg*(3C) is recommended.

AUTHOR
Getmsg was developed by HP.

SEE ALSO
gencat(1), insertmsg(1), read(2), catgetmsg(3C), catopen(3C), nl_catopen(3C), hpnls(5).

EXTERNAL INFLUENCES
International Code Set Support
Single- and multi-byte character code sets are supported.

NAME

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

SYNOPSIS

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

extern char *optarg;
extern int optind, opterr;

DESCRIPTION

Getopt returns the next option letter in *argv* (starting from *argv*[1]) that matches a letter in *optstring*. *Optstring* is a string of recognized option letters; if a letter is followed by a colon, the option is expected to have an argument that may or may not 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. The external variable *optind* is initialized to 1 before the first call to the function *getopt*.

When all options have been processed (i.e., up to the first non-option argument), *getopt* returns **EOF**. The special option -- may be used to delimit the end of the options; **EOF** will be returned, and -- will be skipped.

DIAGNOSTICS

Getopt prints an error message on *stderr* and returns a question mark (?) when it encounters an option letter not included in *optstring*. This error message may be disabled by setting *opterr* to zero.

EXAMPLES

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 options **f** and **o**, both of which require arguments:

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

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

WARNINGS

Options can be any ASCII characters except colon (:), question mark (?), or null (\0). It is impossible to distinguish between a ? used as a legal option, and the character that *getopt* returns when it encounters an invalid option character in the input.

SEE ALSO

getopt(1).

EXTERNAL INFLUENCES

Locale

The LC_CTYPE category determines the interpretation of option letters as single and/or multi-byte characters.

International Code Set Support

Single- and multi-byte character code sets are supported with the exception of multi-byte character file names.

STANDARDS CONFORMANCE

getopt: SVID2, XPG2, XPG3

optarg: SVID2, XPG2, XPG3

opterr: SVID2, XPG2, XPG3

optind: SVID2, XPG2, XPG3

NAME

getpass – read a password

SYNOPSIS

char *getpass (prompt)
char *prompt;

DESCRIPTION

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

FILES

/dev/tty

SEE ALSO

crypt(3C).

WARNING

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

WARNINGS

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

STANDARDS CONFORMANCE

getpass: SVID2, XPG2, XPG3

NAME

getpw – get name from UID

SYNOPSIS

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

DESCRIPTION

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

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

NETWORKING FEATURES

NFS

This routine is implemented using *getpwuid*(3C) and therefore uses the Yellow Pages network database as described in *passwd*(4).

DIAGNOSTICS

Getpw returns non-zero on error.

WARNINGS

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

AUTHOR

Getpw was developed by AT&T and HP.

FILES

/etc/passwd

SEE ALSO

getpwent(3C), passwd(4).

STANDARDS CONFORMANCE

getpw: XPG2

NAME

getpwent, getpwuid, getpwnam, setpwent, endpwent, fgetpwent – get password file entry

SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwent ( )

struct passwd *getpwuid (uid)
uid_t uid;

struct passwd *getpwnam (name)
char *name;

void setpwent ( )

void endpwent ( )

struct passwd *fgetpwent (f)
FILE *f;
```

DESCRIPTION

The *getpwent*, *getpwuid*, and *getpwnam* functions locate an entry in the **/etc/passwd** file, and return a pointer to an object of type **struct passwd**.

The **passwd** structure is defined in <**pwd.h**> and includes the following members:

```
char    *pw_name;
char    *pw_passwd;
int     pw_uid;
int     pw_gid;
char    *pw_age;
char    *pw_comment;
char    *pw_gecos;
char    *pw_dir;
char    *pw_shell;
long    pw_audid;
int     pw_audflg;
```

The **pw_comment** field is unused; the others have meanings described in *passwd*(4).

Getpwent when first called returns a pointer to the first **passwd** structure in the file; thereafter, it returns a pointer to the next **passwd** structure in the file. In this way, successive calls can be used to search the entire file. *Getpwent* opens the **/etc/passwd** file prior to doing its work and leaves the file open afterward; *setpwent* has the effect of rewinding this file to allow repeated searches; *endpwent* can be called to close the file when processing is complete.

Getpwuid searches from the beginning of the file until a numeric user ID matching *uid* is found, and returns a pointer to the particular structure in which it was found.

Getpwnam searches from the beginning of the file until a login name matching *name* is found, and returns a pointer to the particular structure in which it was found.

Fgetpwent returns a pointer to the next **passwd** structure in the standard I/O stream *f*, which should be open for reading, and its contents should match the format of **/etc/passwd**.

SECURITY FEATURES

If the secure password file (**/.secure/etc/passwd**) exists on the system and the calling process has permission to access it, the *getpwent* routines fill in the encrypted password, audit ID, and audit flag from the corresponding entry in that file.

If the secure password file exists but the caller does not have permission to read the it, the encrypted password field is set to * and the audit ID and audit flag are set to -1.

If the secure password file does not exist, the encrypted password in **/etc/passwd** is returned and the audit ID and audit flag are set to -1.

In situations where it is not necessary to get information from the regular password file, *getspwent*(3C) is significantly faster because it avoids unnecessary searches of the regular password file, and does not use the Yellow Pages database.

Putpwent affects only **/etc/passwd**; the audit ID and audit flag in the password structure are ignored. *Putspwent*(3C) must be used to modify **/.secure/etc/passwd**.

NETWORKING FEATURES

NFS

If an entry beginning with a plus sign ('+') or a minus sign ('-') is found, these routines try to use the Yellow Pages network database for data. See *passwd*(4) for proper syntax and operation.

RETURN VALUE

The *getpwent*, *getpwuid*, *getpwnam*, and *fgetpwent* functions return a **NULL** pointer if an end-of-file or error is encountered on reading. Otherwise, the return value points to an internal static area containing a valid **passwd** structure.

WARNINGS

The above routines use **<stdio.h>** and the Yellow Pages library, which causes them to increase the size of programs, not otherwise using standard I/O and Yellow Pages, more than might be expected.

The value returned by these functions points to a single static area that is overwritten by each call to any of the functions, so it must be copied if it is to be saved.

DEPENDENCIES

NFS

FILES

/etc/yp/domainname/passwd.byname
/etc/yp/domainname/passwd.byuid

SEE ALSO

ypcat(1).

AUTHOR

Getpwent, *getpwuid*, *getpwnam*, *setpwent*, *endpwent*, and *fgetpwent* were developed by AT&T and HP.

FILES

/etc/passwd

SEE ALSO

cuserid(3S), getlogin(3C), getgrent(3C), stdio(3S), passwd(4), getspwent(3C), putspwent(3C).

ypcat(1), spasswd(4) in *HP-UX Networking Reference.*

STANDARDS CONFORMANCE

getpwent: SVID2, XPG2

endpwent: SVID2, XPG2

fgetpwent: SVID2, XPG2

getpwnam: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

getpwuid: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

setpwent: SVID2, XPG2

NAME

gets, fgets – get a string from a stream

SYNOPSIS

#include <stdio.h>

char *gets (s)
char *s;

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

DESCRIPTION

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

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

SEE ALSO

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

DIAGNOSTICS

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a NULL pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a NULL pointer is returned. Otherwise *s* is returned.

STANDARDS CONFORMANCE

gets: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

fgets: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

getspwent, getspwuid, getspwaid, getspwnam, setspwent, endspwent, fgetspwent – get secure password file entry

SYNOPSIS

#include <pwd.h>

struct s_passwd *getspwent ( )

struct s_passwd *getspwuid (uid)
uid_t uid;

struct s_passwd *getspwaid (aid)
aid_t aid;

struct s_passwd *getspwnam (name)
char *name;

void setspwent ( )

void endspwent ( )

struct s_passwd *fgetspwent (f)
FILE *f;

DESCRIPTION

These privileged routines provide access to the secure password file in a manner similar to the way *getpwent*(3C) routines handle the regular password file, (**/etc/passwd**).

These routines are particularly useful in situations where it is not necessary to get information from the regular password file. *Getspwent*(3C) routines run significantly faster than *getpwent*(3C) routines because they avoid unnecessary scanning of the password file and use of Yellow Pages.

Getspwent, *getspwuid, getspwaid* and *getspwnam* each return a pointer to an object. The **s_passwd** structure is written in the **/.secure/etc/passwd** file, and consists of five fields per line, as follows:

```
struct s_passwd {
      char   *pw_name;        /* login name */
      char   *pw_passwd;      /* encrypted password */
      char   *pw_age;         /* password age */
      int     pw_audid;       /* audit ID   */
      int     pw_audflg;      /* audit flag 1=on, 0=off */
};
```

Since the **s_passwd** structure is declared in the **<pwd.h>** header file, it is unnecessary to redeclare it.

When first called, *getspwent* returns a pointer to each **s_passwd** structure in **/.secure/etc/passwd** in sequence; subsequent calls can be used to search the entire file. *Getspwuid* searches each entry from the beginning of the file until it finds a numerical user ID matching *uid;* then it returns a pointer to the particular structure in which *uid* is found. Similarly, *getspwaid* searches for a numerical audit ID matching *aid* and returns a pointer to the particular structure in which *aid* is found. (See *spasswd*(4) for details on this field.) *Getspwnam* 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 *name* is found.

A call to *setspwent* resets the file pointer to the beginning of the **/.secure/etc/passwrd** file to allow repeated searches. *Endspwent* can be called to close the secure password file when processing is complete.

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

DIAGNOSTICS

Getspwent returns a **NULL** pointer if any of these routines encounter an end-of-file or error while searching, or if the effective user ID of the calling process is not zero.

WARNINGS

The above routines use <**stdio.h**>, which causes them to increase the size of programs by more than might be expected.

Since all information is contained in a static area, it must be copied to be saved.

AUTHOR

Getspwent was developed by HP.

FILES

/.secure/etc/passwd

SEE ALSO

getgrent(3C), getlogin(3C), getpwent(3C), putspwent(3C), passwd(4), spasswd(4).

ypcat(1) in *HP-UX Networking Reference.*

NAME

getutent, getutid, getutline, pututline, setutent, endutent, utmpname – access utmp file entry

SYNOPSIS

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

struct utmp *getutent ( )

struct utmp *getutid (id)
struct utmp *id;

struct utmp *getutline (line)
struct utmp *line;

void pututline (utmp)
struct utmp *utmp;

void setutent ( )

void endutent ( )

void utmpname (file)
char *file;
```

DESCRIPTION

Getutent, getutid and *getutline* each return a pointer to a structure of the following type:

```
struct utmp {
        char ut_user[8];              /* User login name */
        char ut_id[4];                /* /etc/inittab id (usually line #) */
        char ut_line[12];             /* device name (console, lnxx) */
        pid_t ut_pid;                 /* process id */
        short ut_type;                /* type of entry */
        struct exit_status {
                short                 e_termination;/* Process termination status */
                short                 e_exit;  /* Process exit status */
                } ut_exit;            /* The exit status of a process */
                                      /* marked as DEAD_PROCESS. */
        unsigned short ut_reserved1;          /* Reserved for future use */
        time_t ut_time;               /* time entry was made */
        char ut_host[16];             /* host name, if remote; NOT SUPPORTED */
        unsigned long ut_addr;        /* Internet addr of host, if remote */
};
```

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, *getutid* will return a pointer to the first entry whose type is one of these four and whose *ut_id* field matches *id->ut_id*. If the end of file is reached without a match, it fails.

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.

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

Endutent closes the currently open file.

Utmpname allows the user to change the name of the file examined, from **/etc/utmp** to any other 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.

The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made. Each call to either *getutid* or *getutline* sees the routine examine the static structure before performing more I/O. If the contents of the static structure match what it is searching for, it looks no further. For this reason to use *getutline* to search for multiple occurrences, it would be necessary to zero out the static after each success, or *getutline* would just return the same pointer over and over again. There is one exception to the rule about removing the structure before further reads are done. The implicit read done by *pututline* (if it finds that it is not already at the correct place in the file) will not hurt the contents of the static structure returned by the *getutent*, *getutid* or *getutline* routines, if the user has just modified those contents and passed the pointer back to *pututline*.

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

RETURNS

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

A **NULL** pointer is also returned if the size of the file is not an integral multiple of **sizeof(struct utmp)**.

WARNINGS

Some vendors' versions of *getutent* erase the *utmp* file if the file exists but is not an integral multiple of **sizeof(struct utmp)**. Given the possiblity of user error in providing a name to *utmpname* (such as giving improper arguments to *who*(1)), HP-UX does not do this, but instead returns an error indication.

FILES

/etc/utmp
/etc/wtmp

SEE ALSO

ttyslot(3C), utmp(4).

STANDARDS CONFORMANCE

endutent: SVID2, XPG2

getutent: SVID2, XPG2

getutid: SVID2, XPG2

getutline: SVID2, XPG2

pututline: SVID2, XPG2

setutent: SVID2, XPG2
utmpname: SVID2, XPG2

NAME

gpio_get_status – return status lines of GPIO card

SYNOPSIS

int gpio_get_status (eid)
int eid;

DESCRIPTION

Gpio_get_status enables you to read the status register of the GPIO interface associated with the device file identified by *eid*. *Eid* is an entity identifier of an open GPIO device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2). The current state of each status line on the interface card is mapped to the value returned, with STS0 mapped to the least significant bit. Only *x* least-significant bits are used, where *x* is the number of status lines available on the hardware interface being used.

DEPENDENCIES

Series 300

For the 98622A, *x* is 2.

Series 800

For the 27114A, *x* is 2.

For the 27114B, *x* is 6.

For the 28651A, *x* is 5.

RETURN VALUE

Gpio_get_status returns the value of the status register of the GPIO interface associated with *eid*, and −1 if an error was encountered.

ERRORS

Gpio_get_status fails if one of the following conditions is true and sets **errno** accordingly:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to a GPIO device file.

NAME

gpio_set_ctl – set control lines on GPIO card

SYNOPSIS

int gpio_set_ctl (eid, value)
int eid, value;

DESCRIPTION

Gpio_set_ctl enables you to set the control register of a GPIO interface. *Eid* is an entity identifier of an open GPIO device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Value* is the value to be written into the control register of the GPIO interface associated with *eid*.

Value is mapped onto the control lines on the interface card, with the least significant bit mapped to CTL0. Only the x least significant bits are used, where x is the number of control lines available on the hardware interface being used.

DEPENDENCIES

Series 300
For the 98622A, x is 2.

Series 800
For the 27114A, x is 3.

For the 27114B, x is 6.

For the 28651A, x is 5.

RETURN VALUE

Gpio_set_ctl returns 0 if successful, and −1 if an error was encountered.

ERRORS

Gpio_set_ctl fails if one of the following conditions is true and sets **errno** accordingly:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to a GPIO device file.

NAME

HankakuZenkaku, ZenkakuHankaku – translate characters

SYNOPSIS

```
#include <jlib.h>

unsigned char *HankakuZenkaku (s1, s2, mode)
unsigned char *s1, *s2;
int mode;

unsigned char *ZenkakuHankaku (s1, s2)
unsigned char *s1, *s2;
```

DESCRIPTION

The arguments *s1* and *s2* point to strings (arrays of characters terminated by a null character).

HankakuZenkaku copies string *s1* to *s2* translating 8-bit special, 8-bit alphanumeric, and HANKAKU KATAKANA characters to the corresponding 16-bit ones. HANKAKU KATAKANA characters are translated to HIRAGANA or KATAKANA characters based on *mode*. The argument *mode* must be one of the following:

toHiragana	translate to HIRAGANA characters (default)
toKatakana	translate to KATAKANA characters

where default indicates which value is taken if neither value is specified.

If some characters in *s1* can be translated into the set of 8-bit special, 8-bit alphanumeric, and HANKAKU KATAKANA characters, *ZenkakuHankaku* copies string *s1* to *s2* translating them. Otherwise, it copies without translating.

DIAGNOSTICS

Each function returns *s2* upon successful completion. Otherwise, a NULL pointer is returned.

WARNINGS

ZenkakuHankaku copies the following characters, which are expressed by section-point code, without translating them because there are no characters corresponding to them in *kana*(8).

04-78
04-80
04-81
05-78
05-80
05-81
05-85
05-86

A KATAKANA character with DAKUON or HANDAKUON is translated to two HANKAKU KATAKANA characters followed by DAKUTEN or HANDAKUTEN, and *vice versa*.

Neither function checks for overflow of any input string. In *HankakuZenkaku* the length of the resultant string is not greater than twice the length of *s1*. In *ZenkakuHankaku* the resultant string is shorter than *s1*.

SEE ALSO

open_jlib(3X)

NAME

Henkan, JiKouho, Kakutei, HenkanOwari, SetUserDict – KANA to KANJI conversion routines

SYNOPSIS

```
#include <jlib.h>

Bun *Henkan (ed, string, len, buf, size, mode)
int ed;
unsigned char *string, *buf;
int len, size, mode;

int JiKouho (ed, pb, nb)
int ed;
Bun *pb;
int nb;

int Kakutei (ed, pb, nb, nk)
int ed;
Bun *pb;
int nb, nk;

int HenkanOwari (ed, pb)
int ed;
Bun *pb;

int SetUserDict (ed, dp, mode)
int ed;
UserDict dp;
int mode;
```

DESCRIPTION

Ed is an environment descriptor obtained from a *open_kana_kan* call.

Henkan performs KANA to KANJI conversion for *string*. *String* is an array of characters terminated by a null character. *Len* is the length of the first BUNSETSU in bytes. If a positive value is assigned to *len*, *henkan* takes its value as the length of the first BUNSETSU. Otherwise, it is ignored. *Henkan* puts the resultant string (an array of characters terminated by a null character) into *buf* and returns a pointer to the **Bun** structure. *Size* is the size of *buf* in bytes. *Mode* is a flag having one of the following values:

JIDOU	enable automatic KANA to KANJI conversion.
IKKATSU	disable automatic KANA to KANJI conversion (default).

where default is the value used when neither value is specified.

The Bun structure declared in the <**jlib.h**> header file includes the following fields:

```
int nbunsetsu;         /* number of BUNSETSU */
int nvalid;            /* number of BUNSETSU with validity */
Bunsetsu **bunsetsu;   /* BUNSETSU table */
```

where *nvalid* is effective when *mode* is **JIDOU**.

The Bunsetsu structure includes the following fields:

```
unsigned char *yomi;    /* YOMI (an array of characters ter-
                           minated by a null character)*/
Kouho kouho;            /* KOUHO associated with YOMI */
Kouhogun *kouhogun;     /* a pointer to Kouhogun structure */
```

where the Kouho structure includes the following field:

unsigned char *hyouki;	/* HYOUKI (an array of characters terminated by a null character) */

and *kouhogun* is set to a NULL pointer by *henkan.*

JiKouho is used to get all KOUHOs for any BUNSETSU in the given sentence associated with *pb*. *Pb* is a pointer to a Bun structure obtained from a *Henkan* call. *Nb* is an index for the BUNSETSU table in the Bun structure. *JiKouho* sets a pointer to the Kouhogun structure in the Bun structure. The structure declared in the **<jlib.h>** header file includes the following fields:

```
int nkouho;        /* number of KOUHO */
Kouho **kouho;     /* KOUHO table */
```

The KOUHO before the last one and the last one are spelled by HIRAGANA and KATAKANA respectively.

Henkan and *JiKouho* allocate space themselves. After *HenkanOwari* is performed, this space is made available for further allocation. The argument to *HenkanOwari* is a pointer to a Bun structure obtained from a *Henkan* call.

Kakutei is used to update HINDO information after *Kakutei* is performed. The KOUHO appears with higher priority in further conversion. *Pb* is a pointer to a Bun structure. *Nb* is an index for the BUNSETSU table in the Bun structure and *Nk* is an index for the KOUHO table in the Kouhogun structure.

SetUserDict is used to enable or disable consulting of a user dictionary in addition to a system dictionary during the KANA to KANJI conversion. The last enabled dictionary is consulted first. *Dp* is a dictionary pointer returned by *J_UD_open*. *Mode* specifies the action to be taken, and must be one of the following:

UDoff disable consulting of a user dictionary

UDon enable user dictionary consulting (default)

where default indicates which value is used when neither value is specified.

DIAGNOSTICS

Henkan returns a NULL pointer upon conversion failure. **jlib_errno** is set to indicate the error:

[JNOSPC]	Not enough space to return the result.
[JNOBUF]	No more space to put the resultant string into *buf.*
[JBADSIZ]	*Size* is equal to or less than 0.
[JINVAL]	*String* is too long.
[JNOTRESPOND]	A server does not respond.
[JBADED]	*Ed* is not a valid environment descriptor.

JiKouho returns 0 upon successful completion. Otherwise, -1 is returned and **jlib_errno** is set to indicate the error:

[JNOSPC]	Not enough space in memory to return the result.
[JINVAL]	*Nb* is invalid.
[JNOTRESPOND]	A server does not respond.
[JBADED]	*Ed* is not a valid environment descriptor.

Kakutei returns 0 upon successful completion. Otherwise, -1 is returned and **jlib_errno** is set to indicate the error:

[JNOTRESPOND]	A server does not respond.
[JINVAL]	*Nb* or *nk* is invalid.
[JBADED]	*Ed* is not a valid environment descriptor.

HenkanOwari returns 0 upon successful completion. Otherwise, -1 is returned and **jlib_errno** is set to indicate the error.

[JNOTRESPOND]	A server does not respond.
[JBADED]	*Ed* is not a valid environment descriptor.

SetUserDict returns 0 upon successful completion. Otherwise, -1 is returned and **jlib_errno** is set to indicate the error:

[JBADED]	*Ed* is not a valid environment descriptor.
[JUDBADDP]	*Dp* is not a valid dictionary pointer.
[JNOTRESPOND]	A server does not respond.

EXAMPLES

The following example shows typical use for the above routines. After KANA to KANJI conversion is performed, one of the following actions is taken for each BUNSETSU:

- If the BUNSETSU matchs what you want, *Kakutei* is invoked.
- Otherwise, *JiKouho* is invoked to get all KOUHOs, and *Kakutei* is invoked for the KOUHO matching what you want.

HenkanOwari is invoked with the return value of the previous *Henkan* call.

```
for (;;) {                                /* top level */
    /* get sentences */

    /* conversion (KANA to KANJI) */
    if ((p = Henkan (ed, string, 0, buf, BUFSIZ, IKKATSU)) == NULL)
        error();

    /* accept result or not */
    n = p->nbunsetsu;                     /* number of BUNSETSU */
    /* for each BUNSETSU */
    for (i = 0; i < n; i++) {
        cp = p->bunsetsu[i];
        if ( /* acceptable */ )
            Kakutei (ed, p, i, 0);        /* accept */
        else {
            /* get alternatives */
            if (JiKouho (ed, p, i) == -1)
                error();
            /* select one of them */
            /* assume it to be k-th entry */
            Kakutei (ed, p, i, k-1);
        }
    } /*  end of for (i = 0; i < n; i++)  */

    HenkanOwari (ed, p);

} /*  end of for (;;)  */
```

SEE ALSO

open_kana_kan(3X), J_UD_open(3X)

NAME

HiraganaKatakana, KatakanaHiragana – translate characters

SYNOPSIS

#include <jlib.h>

unsigned char *HiraganaKatakana (s1, s2)
unsigned char *s1, *s2;

unsigned char *KatakanaHiragana (s1, s2)
unsigned char *s1, *s2;

DESCRIPTION

The arguments *s1* and *s2* point to strings (arrays of characters terminated by a null character).

HiraganaKatakana copies string *s1* to *s2* translating HIRAGANA characters in *s1* to corresponding KATAKANA characters. *KatakanaHiragana* copies string *s1* to *s2* translating KATAKANA characters in *s1* to corresponding HIRAGANA characters.

Speaking in another way, HIRAGANA characters from 04-01 to 04-83 in section-point code is translated to corresponding KATAKANA characters from 05-01 to 05-83 by *HiraganaKatakana*. *KatakanaHiragana* does just the opposite. Here is an illustartion of what *HiraganaKatakana* and *KatakanaHiragana* do:

	HiraganaKatakana	
HIRAGANA	-->	KATAKANA
04-15	<--	05-15
	KatakanaHiragana	

Characters except 04-01 to 04-83 in *HiraganaKatakana*, and characters except 05-01 to 05-83 in *KatakanaHiragana* are copied without translating.

DIAGNOSTICS

Each function returns *s2* upon successful completion. Otherwise, a NULL pointer is returned.

WARNINGS

KatakanaHiragana copies the three 16-bit KATAKANA characters expressed by the following section-point codes without translating, because there are no HIRAGANA characters corresponding to them.

05-84
05-85
05-86

SEE ALSO

open_jlib(3X)

NAME

hpib_abort – stop activity on specified HP-IB bus

SYNOPSIS

int hpib_abort (eid);
int eid;

DESCRIPTION

Hpib_abort terminates activity on the addressed HP-IB bus by pulsing the IFC line. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call.

Hpib_abort also sets the REN line and clears the ATN line. The status of the SRQ line is not affected. The interface must be the system controller of the bus.

RETURN VALUE

Hpib_abort returns 0 (zero) if successful, or −1 if an error was encountered.

ERRORS

Hpib_abort fails under the following circumstances, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to an HP-IB raw bus device file.

[EIO] the specified interface is not the system controller.

[ETIMEDOUT] a timeout occurs.

[EACCES] The interface associated with this *eid* is locked by another process and *O_NDELAY* is set for this *eid* (see io_lock(3I)).

DEPENDENCIES

Series 300:

The HP 98625A/B HP-IB interface does not clear the ATN line.

EIO is returned if a timeout occurs.

Series 800:

If the interface is not currently the system controller, *hpib_abort* sets errno to [EACCES] instead of to [EIO].

AUTHOR

Hpib_abort was developed by the Hewlett-Packard Company.

NAME

hpib_address_ctl – set the HP-IB bus address for an interface

SYNOPSIS

int hpib_address_ctl (eid, ba);
int eid, ba;

DESCRIPTION

Hpib_address_ctl sets the HP-IB bus address of the interface associated with *eid* to *ba*. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Ba* is an integer and must be in the range of [0-30].

The new bus address will remain in effect until a reboot, an *io_reset* call, or another *hpib_address_ctl* call occurs. When a *reboot* or *io_reset* call occurs, the HP-IB bus address reverts to its powerup value.

RETURN VALUE

Hpib_address_ctl returns 0 (zero) if successful, or −1 if an error was encountered.

ERRORS

Hpib_address_ctl fails under the following circumstances and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	a timeout occurred.
[EINTR]	the request was interrupted by a signal.
[EINVAL]	*ba* is not in the range of 0-30.

AUTHOR

Hpib_address_ctl was developed by the Hewlett-Packard Company.

SEE ALSO

io_reset(3I).

NAME

hpib_atn_ctl – control the Attention line on HP-IB

SYNOPSIS

int hpib_atn_ctl (eid, flag);
int eid, flag;

DESCRIPTION

Hpib_atn_ctl enables/disables the Attention (ATN) line depending upon the value of *flag*. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Flag* is an integer which, if non-zero, enables the ATN line, and otherwise disables it.

RETURN VALUE

Hpib_atn_ctl returns 0 (zero) if successful, or −1 if an error was encountered.

ERRORS

Hpib_atn_ctl fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	the interface is not the active controller or a timeout occurred.

AUTHOR

Hpib_atn_ctl was developed by the Hewlett-Packard Company.

NAME

hpib_bus_status – return status of HP-IB interface

SYNOPSIS

#include <dvio.h>

int hpib_bus_status (eid, status);
int eid, status;

DESCRIPTION

Hpib_bus_status enables you to determine various status information about an HP-IB channel. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Status* is an integer determining what status information is returned for a particular call. The values defined for *status* and their associated meanings are:

REMOTE_STATUS
Is the channel currently in remote state?

SRQ_STATUS
What is the current state of the SRQ line?

NDAC_STATUS
What is the current state of the NDAC line?

SYS_CONT_STATUS
Is the channel currently system controller?

ACT_CONT_STATUS
Is the channel currently active controller?

TALKER_STATUS
Is the channel currently addressed as talker?

LISTENER_STATUS
Is the channel currently addressed as listener?

CURRENT_BUS_ADDRESS
What is the channel's bus address?

The remote state status is not defined when the interface is the active controller, although reading remote state status in such a situation is not an error. Determining the status of the NDAC line is not available on all machines, and its use is therefore discouraged to ensure compatibility among various systems. Machines which do not support sensing the NDAC line return an error.

RETURN VALUE

Hpib_bus_status's return value depends upon the value of *status*. If *status* is **CURRENT_BUS_ADDRESS**, then the return value is either the HP-IB bus address or -1 if an error occurred. If *status* is any of the other values, then the return value is 0 if the condition is false (the line is clear), 1 if the condition is true (the line is set), or -1 if an error occurred.

ERRORS

Hpib_bus_status fails under the following conditions, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to an HP-IB raw bus device file.

[EINVAL] *status* is not one of the values specified above.

DEPENDENCIES

Series 300

The status of those lines being driven by the interface is undefined, although reading them in such a situation is not an error. Non-active controllers cannot sense the SRQ line. Active listeners cannot sense the NDAC line.

The HP 98625A/B HP-IB interface cannot determine the current state of the NDAC line. Attempts to read this line will fail and set **errno** (see *errno*(2)) to EINVAL.

AUTHOR

Hpib_bus_status was developed by HP.

NAME

hpib_card_ppoll_resp – control response to parallel poll on HP-IB

SYNOPSIS

int hpib_card_ppoll_resp (eid,flag);
int eid,flag;

DESCRIPTION

Hpib_card_ppoll_resp enables an interface to enable (or disable) itself for parallel polls. It also controls the sense, and determines the line on which the response is sent. This gives the interface the ability to either ignore or respond to a parallel poll depending upon whether or not it is enabled to respond.

Eid is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Flag* is an integer having one of the following bit patterns:

Bit Pattern	**Meaning**
10000	Disable parallel poll response.
0SPPP	Enable parallel poll response, where S = sense of the response, and PPP = 3-bit binary number specifying the line on which the response is sent where the octal values 0 through 7 correspond to lines DIO1 through DIO8.

RETURN VALUE

Hpib_card_ppoll_resp returns 0 (zero) if successful, or −1 if an error was encountered.

ERRORS

Hpib_card_ppoll_resp fails under the following circumstances, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EACCES] The interface associated with this *eid* is locked by another process and *O_NDELAY* is set for this *eid* (see *io_lock*(3I)).

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to an HP-IB raw bus device file.

[EINVAL] the device cannot respond on the line number specified by *flag*.

[ETIMEDOUT] a timeout occurs.

DEPENDENCIES

Series 300

The HP 98625A/B HP-IB interface supports only enabling and disabling the parallel poll response (bit 4 of *flag*). The sense and response line number are not programmable on this card.

EIO is returned if a timeout occurs.

Series 800

Since the sense and response line number are not programmable on the HP27110B HP-IB interface, the equivalent parallel poll configuration commands are sent over the HP-IB to the interface. Therefore, this function will fail if the interface is not active controller.

AUTHOR

Hpib_card_ppoll_resp was developed by HP.

SEE ALSO

hpib_ppoll(3I), hpib_ppoll_resp_ctl(3I).

NAME

hpib_eoi_ctl – control EOI mode for HP-IB file

SYNOPSIS

int hpib_eoi_ctl (eid, flag);
int eid, flag;

DESCRIPTION

Hpib_eoi_ctl enables you to turn EOI mode on or off. *Eid* is an entity identifier of an open HP-IB raw device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Flag* is an integer which, if non-zero, enables EOI mode, and otherwise disables it.

EOI mode causes the last byte of all subsequent write operations to be written out with the EOI line asserted, signifying the end of the data transmission. By default, EOI mode is disabled when the device file is opened.

Entity ids for the same device file obtained by separate *open*(2) requests have their own EOI modes associated with them. Entity ids for the same device file obtained by *dup*(2) or inherited by a *fork*(2) request share the same EOI mode. In the latter case, if one process enables EOI mode, then EOI mode is in effect for all such entity ids.

RETURN VALUE

Hpib_eoi_ctl returns a 0 (zero) if successful, or −1 if an error was encountered.

ERRORS

Hpib_eoi_ctl fails under any of the following circumstances and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to an HP-IB device file.

DEPENDENCIES

Series 800

EOI mode is enabled when the device file is first opened.

AUTHOR

Hpib_eoi_ctl was developed by HP.

NAME

hpib_io – perform I/O with an HP-IB channel from buffers

SYNOPSIS

```
#include <dvio.h>
int hpib_io(eid, iovec, iolen)
int eid;
struct iodetail *iovec;
int iolen;
```

DESCRIPTION

Hpib_io enables you to perform and control read and/or write operations on the specified HP-IB bus. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Iovec* is a pointer to an array of structures of the form:

```
struct iodetail {
        char    mode;
        char    terminator;
        int     count;
        char    *buf;
};
```

The *iodetail* structure is defined in the include file **dvio.h**. *Iolen* specifies the number of structures in *iovec*.

The *mode* parameter in the *iodetail* structure describes what is to be done during I/O on the buffer pointed to by *buf*. *Mode* is constructed by OR-ing flags from the following list:

Only one of the following two flags *must* be specified:

HPIBREAD
: Perform a read of the HP-IB bus, placing data into the accompanying buffer.

HPIBWRITE
: Perform a write to the HP-IB bus, using data from the accompanying buffer.

The following flags may be used in most combinations (not all combinations are valid), or not at all:

HPIBATN
: Data is written with ATN enabled.

HPIBEOI
: Data written is terminated with EOI (this flag is ignored when HPIBATN is enabled).

HPIBCHAR
: Data read is terminated with the character given in the *terminator* element of the *iodetail* structure.

Terminator describes the termination character, if any, that should be checked for on input. *Count* is an integer specifying the maximum number of bytes to be transferred.

A read operation terminates when either *count* is matched, an EOI is detected, or the designated *terminator* is detected (if HPIBCHAR is set in *mode*).

A write operation terminates when *count* is matched, and the final byte is sent with EOI asserted (if HPIBEOI is set in *mode*).

If HPIBATN is set in *mode*, then write operations occur with ATN enabled. Setting HPIBATN for a read operation is ignored and has no effect.

The members of the *iovec* array are accessed in order.

RETURN VALUES

If all transactions are successful, *hpib_io* returns a zero and updates the *count* element in each structure in the *iovec* array to reflect the actual number of bytes read or written.

If an error is encountered during a transaction defined by an element of *iovec*, *hpib_io* returns without completing any transactions that might follow. In particular, if an error occurs, *hpib_io* returns a −1, and the *count* element of the transaction which caused the error is set to −1.

ERRORS

Hpib_io fails under any of the following circumstances, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to an HP-IB raw bus device file.

[ETIMEDOUT] a timeout occurs.

[EIO] *eid* is not the active controller.

DEPENDENCIES

Series 300:

EIO is returned if a timeout occurs.

Series 800:

If the interface is not currently the active controller, *hpib_io* sets **errno** to [EACCES] instead of to [EIO].

AUTHOR

Hpib_io was developed by the Hewlett-Packard Company.

NAME

hpib_parity_ctl – enable/disable odd parity on ATN commands

SYNOPSIS

int hpib_parity_ctl (eid, flag);
int eid, flag;

DESCRIPTION

Hpib_parity_ctl enables/disables the sending of odd parity for ATN command sequences depending upon the value of *flag*. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Flag* is an integer which, if non-zero, enables odd parity and otherwise disables it.

Entity ids for the same device file obtained by separate *open*(2) requests have their own parity flags associated with them. Entity ids for the same device file obtained by *dup*(2) or inherited by a *fork*(2) request share the same parity flag. In the latter case, if one process changes the parity flag, the new parity flag is in effect for all such entity ids.

RETURN VALUE

Hpib_parity_ctl returns 0 (zero) if successful, or −1 if an error was encountered.

ERRORS

Hpib_parity_ctl fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to an HP-IB raw bus device file.

AUTHOR

Hpib_parity_ctl was developed by the Hewlett-Packard Company.

NAME

hpib_pass_ctl – change active controllers on HP-IB

SYNOPSIS

int hpib_pass_ctl (eid, ba)
int eid, ba;

DESCRIPTION

Hpib_pass_ctl passes control of a bus to an inactive controller on that bus. The inactive controller becomes the active controller of that bus. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Ba* is the bus address of the intended device.

Not all devices can accept control. Pass control passes only active control of the bus. It cannot pass system control of the bus. The specified interface must be the current active controller but need not be the system controller. The pass control operation does not suspend your program if the inactive controller does not take active control of the bus. However, the interface is no longer active controller.

RETURN VALUE

Hpib_pass_ctl returns 0 (zero) if successful, or −1 if an error was encountered.

ERRORS

Hpib_pass_ctl fails under any of the following circumstances, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	the interface is not the active controller.
[ETIMEDOUT]	a timeout occurs.
[EINVAL]	*ba* is not a valid HP-IB bus address.
[EACCES]	The interface associated with this *eid* is locked by another process and *O_NDELAY* is set for this *eid* (see io_lock(3I)).

DEPENDENCIES

Series 300:

EIO is returned if a timeout occurs.

Series 800:

If the interface is not currently the active controller, *hpib_pass_ctl* sets **errno** to [EACCES] instead of to [EIO].

AUTHOR

Hpib_pass_ctl was developed by the Hewlett-Packard Company.

NAME

hpib_ppoll – conduct parallel poll on HP-IB bus

SYNOPSIS

int hpib_ppoll (eid);
int eid;

DESCRIPTION

Hpib_ppoll conducts a parallel poll on an HP-IB bus. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call.

Devices enabled to respond and that are in need of service can then assert the appropriate DIO line. This enables the controller to determine which devices, if any, need service at a given time. *Hpib_ppoll* delays for 25 microseconds before returning with the response. The interface must be the active controller to conduct a parallel poll.

RETURN VALUE

Hpib_ppoll returns an integer value whose least significant byte corresponds to the byte formed by the 8 data input/output (DIO) lines. Devices enabled to respond to a parallel poll do so on the appropriate DIO line. DIO line 1 corresponds to the least significant bit in the response byte; line 8 to the most significant bit. A −1 return value indicates that an error occurred.

ERRORS

Hpib_ppoll fails under the following situations, and sets *errno* (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	the interface is not current the active controller.

AUTHOR

Hpib_ppoll was developed by the Hewlett-Packard Company.

NAME

hpib_ppoll_resp_ctl – define interface parallel poll response

SYNOPSIS

int hpib_ppoll_resp_ctl (eid, response)
int eid, response;

DESCRIPTION

Eid is an entity identifier of an open HP-IB raw bus device file, obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call.

Hpib_ppoll_resp_ctl defines a response to be sent when an active controller performs a parallel poll on an HP-IB interface. The value of *response* indicates whether this computer does or does not need service. A non-zero *response* value indicates that service is required. This statement only sets up a potential response; no actual response is generated when the statement is executed. The sense of the response and the line number to respond on are set by *hpib_card_ppoll_resp*(3I) or by the active controller.

RETURN VALUE

Hpib_ppoll_resp_ctl returns 0 if the response is successfully set, or -1 if an error has occured.

ERRORS

Hpib_ppoll_resp_ctl fails under the following situations, and sets *errno* (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to a raw HP-IB device file.
[EACCES]	The interface associated with this *eid* is locked by another process and *O_NDELAY* is set for this *eid* (see io_lock(3I)).

AUTHOR

Hpib_ppoll_resp_ctl was developed by the Hewlett-Packard Company.

SEE ALSO

hpib_ppoll(3I), hpib_card_ppoll_resp(3I)

NAME

hpib_ren_ctl – control the Remote Enable line on HP-IB

SYNOPSIS

int hpib_ren_ctl (eid, flag);
int eid, flag;

DESCRIPTION

Hpib_ren_ctl enables/disables the Remote Enable (REN) line depending upon the value of *flag*. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Flag* is an integer which, if non-zero, enables the REN line, and otherwise disables it.

Hpib_ren_ctl, in conjunction with *hpib_send_cmnd*(3I), enables you to place devices into the remote state or local state. The REN line is normally enabled at all times, and is in this state at power-up. Only the system controller may enable/disable the REN line.

RETURN VALUE

Hpib_ren_ctl returns 0 (zero) if successful, or −1 if an error was encountered.

ERRORS

Hpib_ren_ctl fails under the following circumstances, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to an HP-IB raw bus device file.

[EIO] the interface is not the system controller.

AUTHOR

Hpib_ren_ctl was developed by the Hewlett-Packard Company.

NAME

hpib_rqst_srvce – allow interface to enable SRQ line on HP-IB

SYNOPSIS

int hpib_rqst_srvce (eid, cv);
int eid, cv;

DESCRIPTION

Hpib_rqst_srvce specifies the response byte that the interface sends when it is serially polled by the active controller. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Cv* is an integer control value representation of the desired response byte.

Hpib_rqst_srvce optionally enables the SRQ line depending upon the response byte. If bit 6 of the response byte is set, the SRQ line is enabled. It remains enabled until the active controller conducts a serial poll or until the computer executes the request function with bit 6 cleared. The SRQ line is not enabled, however, as long as the interface is active controller. If bit 6 is set, the interface remembers its response byte, and enables the SRQ line when control is passed to another device on the bus.

The response byte looks as follows:

Bit	Meaning
0	SPOLL bit (least significant bit of response byte)
1	SPOLL bit
2	SPOLL bit
3	SPOLL bit
4	SPOLL bit
5	SPOLL bit
6	SRQ line
7	SPOLL bit (most significant bit of response byte)

RETURN VALUE

Hpib_rqst_srvce returns 0 (zero) if successful, or −1 if an error was encountered.

ERRORS

Hpib_rqst_srvce fails under the following circumstances, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to an HP-IB raw bus device file.

[ETIMEDOUT] a timeout occurs.

[EACCES] The interface associated with this *eid* is locked by another process and *O_NDELAY* is set for this *eid* (see io_lock(3I)).

DEPENDENCIES

Series 300

The HP 98625A/B HP-IB interface card allows only bit 6 to be set. All other bits are cleared.

EIO is returned if a timeout occurs.

Series 800

The HP 27110B HP-IB interface card allows only bit 6 to be set. All other bits are cleared.

AUTHOR

Hpib_rqst_srvce was developed by the Hewlett-Packard Company.

NAME

hpib_send_cmnd – send command bytes over HP-IB

SYNOPSIS

int hpib_send_cmnd (eid, ca, length);
int eid, length;
char *ca;

DESCRIPTION

Hpib_send_cmnd enables you to send arbitrary bytes of information on the HP-IB with the ATN line asserted. This enables you to configure and control the bus. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Ca* is a character pointer to a string of bytes to be written to the HP-IB bus as commands. *Length* is an integer specifying the number of bytes in the string pointed to by *ca*.

The interface must currently be the active controller in order to send commands over the bus.

Note that for all HP-IB interfaces, both built-in and plug-in, the most significant bit of each byte is overwritten with a parity bit. All commands are written with odd parity.

RETURN VALUE

Hpib_send_cmnd returns 0 (zero) if successful, or −1 if an error was encountered.

ERRORS

Hpib_send_cmnd fails under the following circumstances, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	the interface is not currently the active controller.
[ETIMEDOUT]	a timeout occurs.
[EACCES]	The interface associated with this *eid* is locked by another process and *O_NDELAY* is set for this *eid* (see io_lock(3I)).
[EINVAL]	The value specified for *length* is invalid, either less than or equal to 0 or greater than MAX_HPIB_COMMANDS as defined in <dvio.h>.

DEPENDENCIES

Series 300:

EIO is returned if a timeout occurs.

Series 800:

If the interface is not currently the active controller, *hpib_send_cmnd* sets **errno** to [EACCES] instead of to [EIO].

AUTHOR

Hpib_send_cmnd was developed by Hewlett-Packard Company.

SEE ALSO

hpib_parity_ctl(3I).

NAME

hpib_spoll – conduct a serial poll on HP-IB bus

SYNOPSIS

int hpib_spoll (eid, ba);
int eid, ba;

DESCRIPTION

Hpib_spoll conducts a serial poll of the specified device. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Ba* is the bus address of the intended device.

Hpib_spoll polls a single device for its response byte. The information stored in the response byte is device specific with the exception of bit 6. If bit 6 of the response byte is set, the addressed device has asserted the SRQ line, and is requesting service. (Note that the least significant bit of the response byte is bit 0.)

Not all devices respond to the serial poll function. Consult the device documentation. Specifying a device that does not support serial polling may cause a timeout error or suspend your program indefinitely. The interface cannot serial poll itself. The interface must be the active controller.

RETURN VALUE

If *hpib_spoll* is successful, the device response byte is returned in the least significant byte of the return value. Otherwise, −1 is returned, indicating an error.

ERRORS

Hpib_spoll fails under the following circumstances, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBAD]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	the interface is not the active controller.
[ETIMEDOUT]	the device polled did not respond before timeout.
[EINVAL]	*ba* is the address of the polling interface itself.
[EACCES]	The interface associated with this *eid* is locked by another process and *O_NDELAY* is set for this *eid* (see io_lock(3I)).

DEPENDENCIES

Series 300:

EIO is returned if a timeout occurs.

Series 800:

If the interface is not currently the active controller, *hpib_spoll* sets **errno** to [EACCES] instead of to [EIO].

AUTHOR

Hpib_spoll was developed by the Hewlett-Packard Company.

SEE ALSO

hpib_rqst_srvce(3I).

NAME

hpib_status_wait – wait until the requested status condition becomes true

SYNOPSIS

#include <dvio.h>

int hpib_status_wait (eid, status);
int eid,status;

DESCRIPTION

Hpib_status_wait enables you to wait until a specific condition has occurred before returning. *Eid* is an entity identifier of an open HP-IB raw bus device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Status* is an integer specifying what information is returned. The possible values for *status* and their associated meanings are:

WAIT_FOR_SRQ
Wait until the SRQ line is enabled.

WAIT_FOR_CONTROL
Wait until this channel is the active controller.

WAIT_FOR_TALKER
Wait until this channel is addressed as talker.

WAIT_FOR_LISTENER
Wait until this channel is addressed as listener.

The wait is subject to the current timeout in effect. If a timeout occurs before the desired condition occurs, the function returns with an error.

RETURN VALUE

Hpib_status_wait returns zero when the condition requested becomes true. A value of −1 is returned if an error occurs. A −1 is also returned if a timeout occurs before the desired condition becomes true.

ERRORS

Hpib_status_wait fails under the following circumstances, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to an HP-IB raw bus device file.

[ETIMEDOUT] a timeout occurs.

[EINVAL] *status* contains an invalid value.

[EACCES] The interface associated with this *eid* is locked by another process and *O_NDELAY* is set for this *eid* (see io_lock(3I)).

DEPENDENCIES

Series 300:

EIO is returned if a timeout occurs.

The following error is also defined:

[EIO] the device is active controller and *status* specifies WAIT_FOR_TALKER or WAIT_FOR_LISTENER.

AUTHOR

Hpib_status_wait was developed by the Hewlett-Packard Company.

NAME

hpib_wait_on_ppoll – wait until a particular parallel poll value occurs

SYNOPSIS

int hpib_wait_on_ppoll (eid, mask, sense)
int eid, mask, sense;

DESCRIPTION

Hpib_wait_on_ppoll waits for a parallel poll response to occur on one or more lines. *Eid* is an entity identifier of an open HP-IB raw bus device file.

The *mask* argument specifies on which lines the parallel poll response is expected. The value of *mask* is viewed as an eight-bit binary number where the least significant bit corresponds to line DIO1; the most significant bit to DIO8. For example, if you want to wait for a response on lines DIO2 and DIO6, the corresponding binary number is 00010010, so a hexadecimal value of 12 should be passed as the *mask* argument.

The *sense* argument specifies what response is expected on the selected lines. The value of *sense* is constructed in the same way as *mask*; eight bits for eight lines. If a bit in *sense* is set, the function returns when the line corresponding to that bit is *cleared*. If a bit in *sense* is clear, the function returns when the corresponding line is *set*. Using the previous example, if *mask* is 0x12 and *sense* is 00000010 (0x02 hexadecimal), the function will return when line DIO5 is set, or when line DIO2 is clear.

RETURN VALUE

Hpib_wait_on_ppoll returns a value of −1 if an error or timeout condition occurs. Upon successful completion, the function returns the response byte XOR-ed with the *sense* value and AND-ed with the *mask*.

ERRORS

Hpib_wait_on_ppoll fails and sets **errno** to indicate the error if any of the following is true:

[EACCES]	The interface associated with this *eid* is locked by another process and *O_NDELAY* is set for this *eid* (see *io_lock*(3I)).
[EBADF]	The *eid* argument is not a valid open entity identifier.
[ENOTTY]	The *eid* argument does not refer to an HP-IB raw bus device file.
[EINVAL]	An invalid mask is received.
[EIO]	The interface is not currently the active controller.
[EIO]	A timeout occurs (Series 300 only).
[ETIMEDOUT]	A timeout occurs (Series 800 only).

DEPENDENCIES

Series 800:

If the interface is not currently the active controller, *hpib_wait_on_ppoll* sets errno to [EACCES] instead of to [EIO].

Series 300:

[EIO] is returned if a timeout occurs.

AUTHOR

Hpib_wait_on_ppoll was developed by HP.

NAME

hpibegin, hpiclose, hpicontrol, hpidelete, hpiend, hpierror, hpifind, hpifindset, hpiget, hpiinfo, hpilock, hpimemo, hpiopen, hpiput, hpiundo, hpiupdate, chpibegin, chpiclose, chpicontrol, chpidelete, chpiend, chpierror, chpifind, chpifindset, chpiget, chpiinfo, chpilock, chpimemo, chpiopen, chpiput, chpiundo, chpiupdate – ALLBASE/HP-UX HPIMAGE programmatic calls

REMARKS

The ALLBASE/HP-UX product must be previously installed on the system for *hpimage* programmatic calls to function.

DESCRIPTION

This set of calls invokes the appropriate *hpimage* procedure or function calls for programmatically accessing an ALLBASE/HP-UX HPIMAGE network database. FORTRAN and Pascal calls are invoked with the calls that begin with "hpi." C calls are invoked with the calls that begin with "chpi." The following descriptions apply to the C calls as well:

hpibegin Designates the beginning of a transaction, and optionally writes user information to the log file.

hpiclose Terminates access to a database or a data set.

hpicontrol Enables or disables the return of chain information.

hpidelete Deletes an entry from the database.

hpiend Defines the end of a transaction, commits the transaction, and optionally writes user information to the log file.

hpierror Supplies a natural language message that interprets the status array as set by any *hpimage procedure.*

hpifind Locates the first and last entries of a data chain in preparation for accessing that chain.

hpifindset Locates entries satisfying a given expression in preparation for access to those entries.

hpiget Retrieves an entry in a data set.

hpiinfo Provides structural information about the database being accessed.

hpilock Locks a database or one or more data sets for exclusive access.

hpimemo Writes user information to the log file.

hpiopen Initiates access to a database.

hpiput Adds a new entry to a data set.

hpiundo Undoes an uncommitted tranaction and optionally writes user information to the log file. This procedure also defines the end of a transaction.

hpiupdate Modifies an existing entry in a database.

The *hpimage* programmatic calls can be executed by all system users.

AUTHOR

The *hpimage* programmatic calls were developed by Hewlett-Packard.

FILES

/usr/bin/hpdbdaemon	cleanup daemon program file
/usr/bin/hpimage	*HPIMAGE* program file
/usr/lib/hpica000	*HPIMAGE* message catalog file

SEE ALSO

ALLBASE/HP-UX HPIMAGE Reference Manual.

NAME

HPPACADDD, HPPACCMPD, HPPACCVAD, HPPACCVBD, HPPACCVDA, HPPACCVDB, HPPACDIVD, HPPACLONGDIVD, HPPACMPYD, HPPACNSLD, HPPACSLD, HPPACSRD, HPPACSUBD – 3000-mode packed-decimal library

SYNOPSIS

#include <hppac.h>

DESCRIPTION

This set of calls invokes the library functions for emulating 3000-mode (MPE V/E) packed-decimal operations. These functions are in library "libcl" which is searched when the option **−lcl** is used with *cc*(1) or *ld*(1).

HPPACADDD	Performs packed-decimal addition.
HPPACCMPD	Compares two packed-decimal numbers.
HPPACCVAD	Converts an ASCII representation to packed-decimal.
HPPACCVBD	Converts a binary representation to packed-decimal.
HPPACCVDA	Converts a packed-decimal number to ASCII.
HPPACCVDB	Converts a packed-decimal number to binary.
HPPACDIVD	Performs packed-decimal division.
HPPACLONGDIVD	Performs packed-decimal division (alternate routine).
HPPACMPYD	Performs packed-decimal multiplication.
HPPACNSLD	Performs a packed-decimal normalizing left shift.
HPPACSLD	Performs a packed-decimal left shift.
HPPACSRD	Performs a packed-decimal right shift.
HPPACSUBD	Performs packed-decimal subtraction.

AUTHOR

The *HPPAC* library was developed by HP.

SEE ALSO

Compiler Library/XL Reference Manual

NAME

hsearch, hcreate, hdestroy – manage hash search tables

SYNOPSIS

#include <search.h>

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

int hcreate (nel)
unsigned nel;

void hdestroy ( )

DESCRIPTION

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

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

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

EXAMPLE

The following example will read in strings followed by two numbers and store them in a hash table, discarding duplicates. It will then read in strings and find the matching entry in the hash table and print it out.

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

struct info {          /* this is the info stored in the table */
        int age, room; /* other than the key. */
};
#define NUM_EMPL    5000    /* # of elements in search table */

main( )
{
        /* space to store strings */
        char string_space[NUM_EMPL*20];

        /* space to store employee info */
        struct info info_space[NUM_EMPL];

        /* next avail space in string_space */
        char *str_ptr = string_space;

        /* next avail space in info_space */
```

```
        struct info *info_ptr = info_space;
        ENTRY item, *found_item, *hsearch( );
        /* name to look for in table */

        char name_to_find[30];
        int i = 0;

        /* create table */
        (void) hcreate(NUM_EMPL);
        while (scanf("%s%d%d", str_ptr, &info_ptr->age,
                &info_ptr->room) != EOF && i++ < NUM_EMPL) {

                /* put info in structure, and structure in item */
                item.key = str_ptr;
                item.data = (char *)info_ptr;
                str_ptr += strlen(str_ptr) + 1;
                info_ptr++;

                /* put item into table */
                (void) hsearch(item, ENTER);
        }

        /* access table */
        item.key = name_to_find;
        while (scanf("%s", item.key) != EOF) {
            if ((found_item = hsearch(item, FIND)) != NULL) {

                /* if item is in the table */
                (void)printf("found %s, age = %d, room = %d\n",
                        found_item->key,
                        ((struct info *)found_item->data)->age,
                        ((struct info *)found_item->data)->room);
            } else {
                (void)printf("no such employee %s\n",
                        name_to_find);
            }
        }
}
```

SEE ALSO

bsearch(3C), lsearch(3C), malloc(3C), string(3C), tsearch(3C), malloc(3X).

DIAGNOSTICS

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

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

WARNING

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

BUGS

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

STANDARDS CONFORMANCE

hsearch: SVID2, XPG2, XPG3

hcreate: SVID2, XPG2, XPG3
hdestroy: SVID2, XPG2, XPG3

NAME

hypot – Euclidean distance function

SYNOPSIS

#include <math.h>

double hypot (x, y)
double x, y;

DESCRIPTION

Hypot returns sqrt(x * x + y * y), taking precautions against unwarranted overflows.

DEPENDENCIES

Series 800 (/lib/libm.a and ANSI C /lib/libM.a)

Hypot returns +INFINITY when *x* or *y* is ±INFINITY .

ERRORS

Series 300

When the correct value would overflow, *hypot* returns HUGE_VAL and sets **errno** to ERANGE.

Series 800 (/lib/libm.a and ANSI C /lib/libM.a)

When the correct value would overflow, *hypot* returns HUGE_VAL and sets **errno** to ERANGE.

Hypot returns NaN and sets **errno** to EDOM when *x* or *y* is NaN.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

isinf(3M), isnan(3M), matherr(3M).

STANDARDS CONFORMANCE

hypot: SVID2, XPG2, XPG3

NAME

iconvsize, iconvopen, iconvclose, iconvlock, ICONV, ICONV1, ICONV2 – code set conversion routines

SYNOPSIS

```
#include <iconv.h>

int iconvsize (tocode, fromcode)
char *tocode;
char *fromcode;

iconvd iconvopen (tocode, fromcode, table, d1, d2)
char *tocode;
char *fromcode;
unsigned char *table;
int d1;
int d2;

int iconvclose (cd)
iconvd cd;

int iconvlock( cd, direction, lock, s)
iconvd cd;
int direction;
int lock;
char *s;

int ICONV (cd, inchar, inbytesleft, outchar, outbytesleft)
iconvd cd;
unsigned char **inchar;
int *inbytesleft;
unsigned char **outchar;
int *outbytesleft;

int ICONV1 (cd, to, from, buflen)
iconvd cd;
unsigned char *to;
unsigned char *from;
int buflen;

int ICONV2 (cd, to, from, buflen)
iconvd cd;
unsigned char *to;
unsigned char *from;
int buflen;
```

DESCRIPTION

Iconvsize finds the size of a table if one is needed to convert characters from the code set specified by the **fromcode** argument to the code set specified by the **tocode** argument. If a conversion table is needed and the table exists, the size of the table in bytes is returned. If a table is needed and the table does not exist, a -1 is returned. If a conversion table is not needed, a 0 is returned.

Iconvopen performs all initializations that have to be done to convert characters from the code set specified by the **fromcode** argument to the code set specified by the **tocode** argument and returns a conversion descriptor of type *iconvd* that identifies the conversion. Up to **MAX_CD** conversions can be open simultaneously. See *iconv*(1) for HP supplied **fromcode** and **tocode** names and their corresponding code sets. For conversions that require a table, the **table** argument is a pointer to the start of the conversion table. It is the caller's responsibility to allocate sufficient memory for the table which is given by *iconvsize*. For conversions that do not require a table, the **table** argument must be a NULL pointer. The *iconvsize* function can be used to determine if a table is needed. For multi-byte code sets, a "converted from" character is mapped to a default character (**d1** or **d2**) if it does not have an equivalent in the "converted to" code set. The multi-byte code sets currently supported can have character lengths of one or two bytes. If a one-byte character is unmapped, then the default one-byte character **d1** is used. Similarly, if a two-byte character is unmapped, then the default two-byte character **d2** is used. Default characters are used since different multi-byte code sets typically do not have the same number of characters which makes a one-to-one mapping difficult. Also unused sections in multi-byte code sets are usually reserved for future use. A different approach is taken with single-byte code sets. For single-byte code sets, it is assumed that the translation table forces a one-to-one mapping between the "from" and "to" characters. No default characters are used with single-byte code sets. This one-to-one mapping guarantees that the conversion is reversible. For example, if the output of a ROMAN8 to ISO 8859/1 conversion is converted back to ROMAN8, then the result of this double conversion is the same as the original data.

Iconvclose closes the conversion descriptor **cd** freeing it up for a subsequent **iconvopen**. It is the caller's responsibility to de-allocate any table associated with the **cd** conversion descriptor.

If needed, code set lock-shift information for the conversion identified by **cd** can be initialized by the *iconvlock* function. If **direction** is 0, then string **s** is used as a lock-shift sequence for the "converted from" or input data. If **direction** is 1, then string **s** is used as a lock-shift sequence for the "converted to" or output data. Currently, three lock-shift sequences can be used in a conversion: lock-shift 0, lock-shift 1 and lock-shift 2. These are identified by the **lock** parameter values 0, 1 and 2. The *iconvlock* function also resets any state information to the initial shift state.

ICONV fetches a character in the "converted from" code set from an input buffer, converts the character to the "converted to" code set and places it plus any lock-shift information into an output buffer. The descriptor **cd** identifies the conversion. The contents of **inchar** points to a single- or multi-byte character in the input buffer and **inbytesleft** points to the number of bytes from the input character to the end of the buffer. The contents of **outchar** points to the next available space in the output buffer and **outbytesleft** points to the number of the bytes from the next available space to the end of the buffer. While conversions are done from the input buffer to the output buffer, the contents of **inchar**, **inbytesleft**, **outchar** and **outbytesleft** are incremented or decremented to reflect the current status of the input and output buffers.

ICONV1 and *ICONV2* are used where it is more efficient to handle single- and multi-byte characters separately. These routines do not check for lock-shift information. *ICONV1* converts single-byte characters in **from** according to the conversion identified by **cd** and returns the converted value in **to**. *ICONV1* assumes **from** contains only single-byte characters. Similarly, *ICONV2* converts double-byte characters in **from** according to the conversion identified by **cd** and returns the converted value in **to**. *ICONV2* assumes **from** contains only double-byte characters. The **buflen** argument in both *ICONV1* and *ICONV2* specifies the number of byes that will be converted.

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUES

Iconvsize returns the size of the conversion table in bytes if a table is needed and it exists. The function returns a -1 if a table is needed and it does not exist. The function returns a 0 if a table is not needed.

Iconvopen returns a conversion descriptor if successful; otherwise a (iconvd) -1 is returned.

Iconvclose returns a non-negative number if successful; otherwise a -1 is returned.

ICONV returns 0 if all characters from the input buffer are successfully converted and placed into the output buffer. *ICONV* returns 1 if a multi-byte input character or a lock-shift sequence spans the input buffer boundary. No conversion is attempted on the character and the contents of **inchar** points to the start of the truncated character sequence. *ICONV* returns 2 if an input character does not belong to the "converted from" character set. No conversion is attempted on the character and the contents of **inchar** points to the start of the unidentified input character. *ICONV* returns 3 if there is no room in the output buffer to place the converted character. The converted characters is not placed in the output buffer and the contents of **inchar** points to the start of the character sequence that caused the output buffer overflow.

ICONV1 and *ICONV2* return the number of bytes converted if successful; otherwise a -1 is returned.

EXAMPLE

```
int
convert( tocode, fromcode, d1, d2)
char *tocode;                      /* tocode name */
char *fromcode;                         /* fromcode name */
int d1;                            /* one-byte default character */
int d2;                            /* two-byte default character */
{
        extern void error();               /* local error message */

        iconvd cd;                         /* conversion descriptor */
        int size;                          /* size of translation table */
        unsigned char *table;              /* ptr to translation table */
        int bytesread;                     /* num bytes read into input buffer */
        unsigned char inbuf[BUFSIZ];       /* input buffer */
        unsigned char *inchar;             /* ptr to input character */
        int inbytesleft;                   /* num bytes left in input buffer */
        unsigned char outbuf[BUFSIZ];      /* output buffer */
        unsigned char *outchar;                 /* ptr to output character */
        int outbytesleft;                  /* num bytes left in output buffer */

        /* create conversion table */
        if ((size = iconvsize( tocode, fromcode)) == BAD) {
                error( FATAL, BAD_SIZE);
        }
        else if (size == 0) {
                table = (unsigned char *) NULL;
        }
        else if ((table = (unsigned char *) malloc ( (unsigned int) size)) == (unsigned char *) NULL) {
                error( FATAL, BAD_CREATE);
        }

        /* start up a conversion */
        if ((cd = iconvopen( tocode, fromcode, table, d1, d2)) == (iconvd) BAD) {
                error( FATAL, BAD_OPEN);
        }

        inchar = inbuf;
        inbytesleft = 0;
```

```
        outchar = outbuf;
        outbytesleft = BUFSIZ;

        /* translate the characters */
        for ( ;; ) {
                switch (ICONV( cd, &inchar, &inbytesleft, &outchar, &outbytesleft)) {
                case 0:
                case 1:
                        /*
                        ** Done with buffer, empty buffer or character spans
                        ** input buffer boundary.  Move any remaining stuff
                        ** to start of buffer, get more characters and
                        ** reinitialize input variables.  If at EOF, flush
                        ** output buffer and leave; otherwise, continue to
                        ** convert the characters.
                        */
                        strncpy( inbuf, inchar, inbytesleft);
                        if ((bytesread = read( Input, inbuf+inbytesleft, BUFSIZ-inbytesleft)) < 0) {
                                perror( "prog");
                                return BAD;
                        }
                        if (! (inbytesleft += bytesread)) {
                                if (write( 1, outbuf, BUFSIZ - outbytesleft) < 0) {
                                        perror( "prog");
                                        return BAD;
                                }
                                goto END_CONVERSION;
                        }
                        inchar = inbuf;
                        break;
                case 2:
                        error( FATAL, BAD_CONVERSION);
                case 3:
                        /*
                        ** Full buffer or output character spans output buffer
                        ** boundary.  Send the output buffer to stdout,
                        ** reinitialize the output variables.
                        */
                        if (write( 1, outbuf, BUFSIZ - outbytesleft) < 0) {
                                perror( "prog");
                                return BAD;
                        }
                        outchar = outbuf;
                        outbytesleft = BUFSIZ;
                }
        }
END_CONVERSION:

        /* end conversion & get rid of the conversion table */
        if (iconvclose( cd) == BAD) {
                error( FATAL, BAD_CLOSE);
        }
        if (size) {
```

```
                free( (char *) table);
            }
            return GOOD;
        }
```

AUTHOR

Iconv was developed by HP.

SEE ALSO

iconv(1)

NAME

initgroups – initialize group access list

SYNOPSIS

initgroups(name, basegid)
char *name;
int basegid;

DESCRIPTION

Initgroups reads the login group file, **/etc/logingroup**, and sets up the group access list for the user specified by *name*, using the *setgroups*(2) system call. If the value of *basegid* is zero or positive, it is automatically included in the groups list. Typically this value is given as the group number from the password file. If the login group file does not exist or is empty, *basegid* is the only member of the list.

DIAGNOSTICS

Initgroups returns **–1** if it was not invoked by the super-user.

WARNINGS

Initgroups uses the routines based on *getgrent*(3C). If the invoking program uses any of these routines, the group structure is overwritten by the call to *initgroups*.

On many systems, no one seems to keep **/etc/logingroup** up to date.

NETWORKING FEATURES

NFS:

If **/etc/logingroup** is linked to **/etc/group**, *initgroups* tries to use the Yellow Pages network database for entries beginning with a plus sign (+). See *group*(4) for proper syntax and operation.

AUTHOR

Initgroups was developed by the University of California, Berkeley.

FILES

/etc/logingroup	login group file

SEE ALSO

login(1), su(1), setgroups(2), group(4).

NAME

io_burst – perform low-overhead I/O on an HP-IB/GPIO channel

SYNOPSIS

#include <dvio.h>

io_burst (eid, flag)

DESCRIPTION

Io_burst enables you to perform low-overhead burst transfers on the specified HP-IB or GPIO channel. **Eid** is the entity identifier for an open HP-IB/GPIO device file returned by a previous call to *open*(2), *dup*(2), *creat*(2), or *fcntl*(2) with an FDUPD command option. **Flag** is an integer which, if non-zero, enables burst mode or, if zero, disables it.

In burst mode, memory-mapped I/O address space assigned to the interface card select code is mapped directly into user space such that data can be transferred directly between user memory and the interface card, eliminating the need for kernel calls and the associated overhead. Burst mode affects only *read*(2), *write*(2), *gpio_get_status*(3I), *gpio_set_ctl*(3I), *hpib_io*(3I), and *hpib_send_cmd*(3I) calls. All other operations are unaffected. When burst mode is enabled, the interface is locked so that no other process can access it until burst mode is disabled. When burst mode is disabled, the interface is reset (see *io_reset*(3I)).

RETURN VALUE

Io_burst returns zero if successful or –1 if an error is detected.

DIAGNOSTICS

Io_burst fails under any of the following circumstances and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF] **eid** does not refer to an open file.

[ENOTTY] **eid** does not refer to an HP-IB or GPIO device special file.

[EIO] a timeout occurred during the call to *ioburst*.

WARNINGS

Enabling burst mode locks the interface from all other processes, so it should never be used with any interface that supports a system disk or swap device.

Timeouts for *read*(2), *write*(2), *gpio_get_status*(3I), *gpio_set_ctl*(3I), *hpib_io*(3I), and *hpib_send_cmd*(3I) do not work while in burst mode, but these commands can be interrupted by signals.

SEE ALSO

read(2), write(2), gpio_get_status(3I), gpio_set_ctl(3I), hpib_io(3I), hpib_send_cmd(3I), io_reset(3I).

NAME

io_dma_ctl – control DMA allocation for an interface

SYNOPSIS

#include <sys/dil.h>
io_dma_ctl (eid, mode)
int eid, mode;

DESCRIPTION

Io_dma_ctl enables you to control system DMA allocation for a specific interface. **Eid** is the entity identifier for an open HP-IB/GPIO device file returned by a previous call to *open*(2), *dup*(2), *creat*(2), or *fcntl*(2) with an FDUPD command option.

The *mode* parameter describes what type of DMA allocation the system should use for the interface associated with **EID.** *Mode* is determined by selecting one of flags from the following list in <sys/dil.h>:

Only one of the following flags *must* be specified:

DMA_ACTIVE
: Inform the DMA subsystem that this interface intends to use DMA and requires higher priority than slow devices. This is the level of DMA allocation used by CS80, Amigo and SCSI devices.

DMA_UNACTIVE
: Remove the effect of a previous DMA_ACTIVE.

DMA_RESERVE
: Guarantee that a DMA channel will remain unlocked for future requests for DMA by all devices on this interface.

DMA_UNRESERVE
: Remove the effect of a previous DMA_RESERVE.

DMA_LOCK
: Lock a DMA channel for exclusive use by all devices on this interface.

DMA_UNLOCK
: Unlock a DMA channel locked by this interface.

RETURN VALUES

Io_dma_ctl returns 0 (zero) if successful, or −1 if an error was encountered.

ERRORS

io_dma_ctl fails under the following circumstances, and sets *errno* (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an DIL bus device file.
[EIO]	a timeout occurred.
[EINTR]	the request was interrupted by a signal.
[EINVAL]	the interface was unable to reserve or lock a DMA channel.

WARNING

There are only two DMA channels available on the Series 300. Use of DMA_LOCK could starve your system disks of DMA resources, resulting in lower system performance.

AUTHOR

Io_dma_ctl was developed by the Hewlett-Packard Company.

NAME

io_eol_ctl – set up read termination character on special file

SYNOPSIS

int io_eol_ctl (eid, flag, match);
int eid, flag, match;

DESCRIPTION

Io_eol_ctl enables you to specify a character to be used in terminating a read operation from the specified file id.

Eid is an entity identifier of an open HP-IB raw bus or GPIO device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Flag* is an integer which enables or disables character-match termination. A non-zero value enables character-match termination, while a zero value disables it. *Match* is an integer containing the numerical equivalent of the termination character. *Match* is ignored if *flag* is zero. When in 8-bit mode, the lower 8 bits of *match* are used as the termination character. In 16-bit mode, the lower 16 bits are used.

Upon opening a file, the default condition is character-match termination disabled. When enabled, the character specified by *match* is checked for during read operations. The read is terminated upon receipt of this character, or upon any of the other termination conditions normally in effect for this file. Examples of other conditions are satisfying the specified byte count, and receiving a character when the EOI line is asserted (HP-IB). When the read is terminated by a *match* character, this character is the last character returned in the buffer.

Entity ids for the same device file obtained by separate *open*(2) requests have their own termination characters associated with them. Entity ids for the same device file inherited by a *fork*(2) request share the same termination character. In the latter case, if one process changes the termination character, the new termination character is in effect for all such entity ids.

RETURN VALUE

Io_eol_ctl returns 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

Io_eol_ctl fails under the following circumstances, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF] *eid* does not refer to an open file [EBADF];

[ENOTTY] *eid* does not refer to a channel device file.

AUTHOR

Io_eol_ctl was developed by HP.

SEE ALSO

io_width_ctl(3I).

NAME

io_get_term_reason – determine how last read terminated

SYNOPSIS

int io_get_term_reason (eid);
int eid;

DESCRIPTION

Io_get_term_reason returns the termination reason for the last read made on this entity id. *Eid* is an entity identifier of an open HP-IB raw bus or GPIO device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call.

All entity ids descending from an *open*(2) request (such as from *dup*(2) or *fork*(2)) set this status. For example, if the calling process had opened this entity id, and later forked, the status returned would be from the last read done by either the calling process or its child.

RETURN VALUE

Io_get_term_reason returns a value indicating how the last read on the specified entity id was terminated. This value is interpreted as follows (note that combinations are possible):

Value	**Description**
−1	An error was encountered while making this function request.
0	Last read encountered some abnormal termination reason not covered by any of the other reasons.
1	Last read terminated by reading the number of bytes requested.
2	Last read terminated by detecting the specified termination character.
4	Last read terminated by detecting some device-imposed termination condition. Examples are: EOI for HP-IB, PSTS line on GPIO, or some other end-of-record condition, such as the physical end-of-record mark on a 9-track tape.

ERRORS

Io_get_term_reason fails under the following circumstances, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to a channel device file.

DEPENDENCIES

Series 300

For the GPIO interface, PSTS is checked only at the beginning of a transfer. An interrupt caused by an EIR will also terminate a transfer. The value of the termination reason in this case is also 4.

AUTHOR

Io_get_term_reason was developed by HP.

SEE ALSO

read(2), io_eol_ctl(3I).

NAME

io_interrupt_ctl – enable/disable interrupts for the associated eid

SYNOPSIS

int io_interrupt_ctl (eid, enable_flag)
int eid, enable_flag;

DESCRIPTION

Eid is an entity identifier of an open HP-IB raw bus or GPIO device file, obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Flag* is an integer which enables or disables interrupts for the associated *eid*. A non-zero value enables interrupts.

Interrupts may be disabled or enabled by the user as desired. When an interrupt occurs for a given *eid* the interrupts associated with this *eid* are automatically disabled from reoccurring. Interrupts for this *eid* may be re–enabled by the user with *io_interrupt_ctl*.

RETURN VALUE

io_interrupt_ctl returns 0 (zero) if successful, or -1 if an error was encountered.

ERRORS

Io_interrupt_ctl fails under the following situations, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to a device that supports interrupts.
[EINVAL]	no interrupt conditions were specified for this *eid*.

AUTHOR

Io_interrupt_ctl was developed by the Hewlett-Packard Company.

SEE ALSO

io_on_interrupt(3I)

NAME

io_lock, io_unlock – lock and unlock an interface

SYNOPSIS

int io_lock (eid)
int eid;
int io_unlock (eid)
int eid;

DESCRIPTION

Eid is an entity identifier of an open HP-IB or GPIO, device file, obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call.

This function attempts to lock the interface associated with an entity identifier for the requesting process. Locking an interface gives exclusive use of the interface associated with the *eid* to the requesting process, thus avoiding unintended interference from other processes during a series of separate I/O requests. All the locks for a process are removed when the process closes the file or terminates.

Other processes that attempt to access or lock a locked interface will either return an error or sleep until the interface becomes unlocked. The action taken is determined by the current setting of the *O_NDELAY* flag (see *open*(2). If the *O_NDELAY* flag is set, accesses to a locked interface will fail and set **errno** to indicate the error. If the *O_NDELAY* flag is not set, accesses to a locked interface will block until the interface is unlocked, the current timeout expires, or the request is interrupted by a signal.

A lock is associated with a process, not an *eid*. Locking an interface with a particular *eid* does not prevent the process that owns the lock from accessing the interface through another *eid*. A lock associated with an *eid* is not inherited by a child process during a *fork*(2).

Nested locking is fully supported. If a process owns a locked interface and calls a generic subroutine that does a lock and unlock, the calling process does not lose its lock on the interface. Locking requests produced by a given process for an interface already locked by the same process will increment the current lock count for that interface.

Io_unlock allows a process to remove a lock from the interface associated with the *eid*. A locked interface can be unlocked only by the process directly owning the lock. When an unlock operation is applied to an *eid* that is currently multiply locked, the unlock operation decrements the current lock counter for that interface, and the interface remains locked until the count is reduced to zero.

RETURN VALUE

Io_lock and *io_unlock* return the integer value of the current lock count if successful. A lock count greater than zero indicates that the interface is still locked. A lock count of zero indicates that the interface is no longer locked. A -1 indicates that an error has occured.

ERRORS

Io_lock and *io_unlock* fail in the following situations, and set **errno** (see *errno*(2)) to the value in square brackets:

[EACCES]	an attempt is made to lock an interface locked by another process with O_NDELAY set.
[EBADF]	an *eid* does not refer to an open file.
[EINTR]	a signal is caught while attempting to perform the lock with O_NDELAY clear.
[EINVAL]	an attempt is made to unlock when the interface is not locked.
[ETIMEDOUT]	a timeout occurs while attempting to perform the lock with O_NDELAY clear.

[ENOTTY] an *eid* does not refer to a channel device file.

[EPERM] an attempt is made to unlock when lock is not owned by this user.

WARNING

Io_lock provides a mandatory lock enforced by the system and should not be used with any interface supporting a system disk or swap device.

DEPENDENCIES

Series 300:

EIO is returned if a timeout occurs.

AUTHOR

Io_lock and *io_unlock* were developed by HP.

SEE ALSO

io_timeout_ctl(3I), open(2).

NAME

io_on_interrupt – device interrupt (fault) control

SYNOPSIS

#include <dvio.h>

```
int (*io_on_interrupt (eid, causevec, handler))()
int eid;
struct interrupt_struct *causevec;
int (*handler)();
```

```
handler (eid, causevec)
int eid;
struct interrupt_struct *causevec;
```

DESCRIPTION

Eid is an entity identifier of an open HP-IB raw bus, or GPIO device file, obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call.

Causevec is a pointer to a structure of the form:

```
struct interrupt_struct {
        integer cause;
        integer mask;
};
```

The *interrupt_struct* structure is defined in the file **dvio.h.**

The *cause* parameter is a bit vector specifying which of the interrupt or fault events will cause the handler routine to be invoked. The interrupt causes are often specific to the type of interface being considered. Also, certain exception (error) conditions can be handled using the *io_on_interrupt* capability. Specifying a zero valued *cause* vector effectively turns off the interrupt for that *eid*.

The *mask* parameter is used when an HP-IB parallel poll interrupt is being defined. *Mask* is an integer that specifies which parallel poll response lines are of interest. The value of *mask* is viewed as an 8-bit binary number where the least significant bit corresponds to line DIO1; the most significant bit to line DIO8. For example, to activate an interrupt handler when a response occurs on lines 2 or 6, the correct binary number is 00100010. Thus a hexadecimal value of 22 is the correct argument value for *mask*.

When an enabled interrupt condition on the specified *eid* occurs, the receiving process executes the interrupt-handler function pointed to by *handler.* The entity identifier *eid* and the interrupt condition *cause* are returned as the first and second parameters respectively.

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

Interrupt *handlers* are not inherited across a *fork*(2). *Eids* for the same device file produced by *dup*(2) share the same *handler*.

An interrupt for a given *eid* is implicitly disabled after the occurrence of the event. The interrupt condition may be re-enabled with *io_interrupt_ctl*(3I).

When an event specified by *cause* occurs, the receiving process executes the interrupt *handler* function pointed to by *handler*. When the *handler* returns, the user process resumes at the point of execution left when the event occurred.

Handler will be passed two parameters, the *eid* associated with the event and a pointer to a *causevec* structure. The cause of the interrupt can be determined by the value returned in the *cause* field of the *causevec* structure (more than 1 bit can be set, indicating that more than 1 interrupting condition has occurred). If the interrupt *handler* was invoked due to a parallel poll interrupt, then the *mask* field of the *causevec* structure will contain the parallel poll response byte.

HP-IB INTERRUPTS

This section describes interrupt causes specific to an HP-IB device. For an HP-IB device the cause is a bit vector which is used as follows. To enable a given event, the appropriate bit (in *cause*), shown below, must be set to **1**:

SRQ	SRQ and active controller
TLK	Talker addressed
LTN	Listener addressed
TCT	Controller in charge
IFC	IFC has been asserted
REN	Remote enable
DCL	Device clear
GET	Group execution trigger
PPOLL	Parallel poll

GPIO INTERRUPTS

This section describes interrupt causes specific to a GPIO device. For a GPIO device the cause is a bit vector which is used as follows. To enable a given event, the appropriate bit (in *cause*), shown below, must be set to **1**:

EIR	External interrupt
SIE0	Status line 0
SIE1	Status line 1

RETURN VALUE

Io_on_interrupt returns a pointer to the previous *handler* if the new *handler* is successfully installed; otherwise it returns a **−1** and **errno** is set.

ERRORS

Io_on_interrupt can fail for any of the following reasons:

[EACCES]	The interface associated with this *eid* is locked by another process and *O_NDELAY* is sed for this *eid* (see *iolock*(3I)).
[EBADF]	*Eid* does not refer to an open file.
[ENOTTY]	*Eid* does not refer to a GPIO or a raw HP-IB device file.
[EFAULT]	*Handler* points to an illegal address. The reliable detection of this error will be implementation dependent.
[EFAULT]	*Causevec* points to an illegal address. The reliable detection of this error will be implementation dependent.

DEPENDENCIES

Series 300

For the HP 98622 GPIO interface, only the EIR interrupt is available. For the HP 98265A/B HP-IB interface, the IFC and GET interrupts are not available.

Series 800

For the HP 27114 AFI interface, only the EIR interrupt is available.

AUTHOR

Io_on_interrupt was developed by HP.

SEE ALSO

pause(2), sigpause(2), io_interrupt_ctl(3I).

NAME

io_reset – reset an I/O interface

SYNOPSIS

int io_reset (eid);
int eid;

DESCRIPTION

Io_reset resets the interface associated with the device file that was opened. It also pulses the peripheral reset line on the GPIO interface, or the IFC line on the HP-IB. *Eid* is an entity identifier of an open DIL device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call.

Io_reset also causes an interface to go through its self-test, and returns a failure indication if the interface fails its test.

RETURN VALUE

Io_reset returns 0 (zero) if successful, or −1 if an error was encountered.

ERRORS

Io_reset fails under the following circumstances, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to a channel device file.
[EIO]	the interface could not be reset, or failed self-test.
[EACCES]	The interface associated with this *eid* is locked by another process and *O_NDELAY* is set for this *eid* (see io_lock(3I)).

DEPENDENCIES

Series 300

When an HP-IB interface is reset, the interrupt mask is set to 0, the parallel poll response is set to 0, the serial poll response is set to 0, the HP-IB address is assigned its powerup default value, the IFC line is pulsed (if system controller), the card is put on line, and REN is set (if system controller).

When a GPIO interface is reset, the peripheral reset line is pulled low, the PCTL line is placed in the clear state, and if the DOUT CLEAR jumper is installed, the data out lines are all cleared. The interrupt enable bit is also cleared.

Interface self-test is not supported.

AUTHOR

Io_reset was developed by HP.

NAME

io_speed_ctl – inform system of required transfer speed

SYNOPSIS

int io_speed_ctl (eid, speed);
int eid, speed;

DESCRIPTION

Io_speed_ctl enables you to select the data transfer speed for a data path used for a particular interface. The transfer method (i.e., DMA, fast-handshake) chosen by the system is determined by the speed requirements.

Eid is an entity identifier of an open HP-IB raw bus or GPIO device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Speed* is an integer specifying the data transfer speed in K-bytes per second (one K-byte equals 1024 bytes).

RETURN VALUE

Io_speed_ctl returns 0 if successful, and −1 otherwise.

ERRORS

Io_speed_ctl fails under the following condition, and sets **errno** to the value enclosed in square brackets:

[ENOTTY] *eid* does not refer to channel device file.

[EBADF] *eid* does not refer to an open file.

DEPENDENCIES

Series 300

For values of speed less than 7, the system will use an interrupt transfer. For larger values, DMA will be used if available; otherwise, the system will use an interrupt transfer. The default transfer method is DMA.

Series 800

DMA is the only supported transfer method.

AUTHOR

Io_speed_ctl was developed by HP.

NAME

io_timeout_ctl – establish a time limit for I/O operations

SYNOPSIS

int io_timeout_ctl (eid, time);
int eid;
long time;

DESCRIPTION

Io_timeout_ctl enables you to assign a timeout value to the specified entity id. *Eid* is an entity identifier of an open HP-IB raw bus or GPIO device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Time* is a long integer value specifying the length of the timeout in microseconds. A value of 0 for *time* specifies no timeout (infinity).

This timeout applies to future read and write requests on this entity id. If a read or write request does not complete within the specified time limit, the request is aborted and returns an error indication. If an operation is aborted due to a timeout, *errno*(2) is set to **ETIMEDOUT.**

Although the timeout value is specified in microseconds, the resolution of the timeout is system-dependent. For example, a particular system might have a resolution of 10 milliseconds, in which case the specified timeout value is rounded up to the next 10 msec boundary. A timeout value of zero means that the system never causes a timeout. When a file is opened, a zero timeout value is assigned by default.

Entity ids for the same device file obtained by separate *open*(2) requests have their own timeout values associated with them. Entity ids for the same device file obtained by *dup*(2) or inherited by a *fork*(2) request share the same timeout value. In the latter case, if one process changes the timeout, the new timeout is in effect for all such entity ids.

RETURN VALUE

Io_timeout_ctl returns 0 (zero) if successful, or −1 if an error was encountered.

ERRORS

Io_timeout_ctl fails under the following circumstances, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to a channel device file.

DEPENDENCIES

Series 300

System timeout resolution is 20 msec.

EIO is returned if an operation is aborted due to a timeout.

AUTHOR

Io_timeout_ctl was developed by HP.

NAME

io_width_ctl – set width of data path

SYNOPSIS

int io_width_ctl (eid, width)
int eid, width;

DESCRIPTION

Io_width_ctl enables you to select the width of the data path to be used for a particular interface. *Eid* is an entity identifier of an open device file obtained from an *open*(2), *dup*(2), *fcntl*(2), or *creat*(2) call. *Width* is an integer specifying the width of the data path in bits.

An error is given if an invalid width is specified. Specifying a width with this function sets the width for all users of the device file associated with the given entity id. When first opened, the default width is 8 bits.

For the GPIO interface only widths of 8 and 16 bits are currently supported. For the HP-IB interface only a width of 8 bits is supported.

RETURN VALUE

Io_width_ctl returns 0 if successful, and –1 if an error was encountered.

ERRORS

Io_width_ctl fails under the following circumstances, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to a channel device file.

[EINVAL] the specified *width* is not supported on this device file.

AUTHOR

Io_width_ctl was developed by HP.

NAME

is_68010_present, is_68881_present, is_98635A_present, is_98248A_present – check for presence of hardware capabilities

SYNOPSIS

int is_68010_present()

int is_68881_present()

int is_98635A_present()

int is_98248A_present()

DESCRIPTION

Each function checks for the presence of a specified hardware capability, returning **1** if it exists or **0** if it does not.

RETURN VALUE

The value **1** is returned by:

is_68010_present if the system has an MC68010 as its CPU.

is_68881_present if an MC68881 floating-point coprocessor is present.

is_98635A_present if an HP 98635A floating-point accelerator has been installed.

is_98248A_present if an HP 98248A floating-point accelerator has been installed.

AUTHOR

Is_hw_present was developed by HP.

NAME

isinf – test for INFINITY function

SYNOPSIS

#include <math.h>

int isinf (x)
double x;

DESCRIPTION

Isinf returns a positive integer if *x* is +INFINITY , or a negative integer if *x* is −INFINITY . Otherwise it returns zero.

DEPENDENCIES

Series 300
This function is not supported.

SEE ALSO

isnan(3M).

STANDARDS CONFORMANCE

isinf: XPG2, XPG3

NAME

isnan – test for NaN function

SYNOPSIS

#include <math.h>

int isnan (x)
double x;

DESCRIPTION

Isnan returns a nonzero integer if *x* is NaN (not-a-number). Otherwise it returns zero.

DEPENDENCIES

Series 300
This function is not supported.

SEE ALSO

isinf(3M).

STANDARDS CONFORMANCE

isnan: XPG2, XPG3

NAME

J_UD_open, J_UD_close, J_UD_search, J_UD_free, J_UD_store, J_UD_delete – manage user dictionaries

SYNOPSIS

```
#include <jlib.h>

UserDict *J_UD_open (filename, mode)
char *filename;
int mode;

int J_UD_close (dp)
UserDict *dp;

int J_UD_store (key, kouho, dp)
unsigned char *key;
UDKouho *kouho;
UserDict *dp;

int J_UD_delete (key, kouho, dp)
unsigned char *key;
UDKouho *kouho;
UserDict *dp;

UDKouhogun *J_UD_search (key, dp)
unsigned char *key;
UserDict *dp;

int J_UD_free (p)
UDKouhogun *p;
```

DESCRIPTION

J_UD_open opens the user dictionary named by *filename* and returns a dictionary pointer to the UserDict structure associated with the dictionary. The UserDict structure is declared in the **<jlib.h>** header file. Various operations to a user dictionary can be performed only by a dictionary pointer. The argument *mode* must be one of the following:

RDONLY	Open for reading only.
RDWR	Open for update (reading and writing).

If *J_UD_open*
tries to open the named dictionary and it does not exist, *J_UD_open* creates a new dictionary.

Dp is a dictionary pointer obtained from a *J_UD_open* call. *J_UD_close* closes the dictionary pointer indicated by *dp*.

J_UD_store is used to store a word. The arguments to *J_UD_store* are *key* and *kouho*. *Key* is a pointer to YOMI about a word to be stored and must be made of HIRAGANA characters. The permissible number of characters is 8 at most, counting DAKUTEN and HANDAKUTEN as one character.

The UDKouho structure includes the following fields:

```
unsigned char *hyouki;   /* HYOUKI (an array of charac-
                            ters terminated by a null charac-
                            ter)*/
int hinshi;              /* HINSHI */
```

It is necessary to give *hyouki* and *hinshi* in a UDKouho structure before calling *J_UD_store*. *Hyouki* points to HYOUKI about the word and must be made of 16-bit Japanese characters. The permissible number of characters is 10 at most. A permissible value for *Hinshi* is as follows:

MEISHI	noun

If a dictionary does not contain a word equal to that to be stored, *J_UD_store* stores the word into the dictionary.

J_UD_delete is used to delete a word from a user dictionary. The arguments are the same as for *J_UD_store*. If a dictionary does not contain a word equal to that to be deleted, *J_UD_delete* takes no action and no errors are encountered.

J_UD_search is used to search a word. *Key* is a pointer to YOMI about the word to be found. *J_UD_search* returns a pointer to a UDKouhogun structure. The UDKouhogun structure is declared in the <**jlib.h**> header file:

```
typedef struct {
    int nkouho:            /* number of KOUHOs */
    UDKouho **kouho;       /* KOUHO table */
} UDKouhogun;
```

Nkouho equal to 0 means there is no word in a dictionary equal to *key (the value pointed to by key). The first entry in KOUHO table is the last stored one for the *key*.

J_UD_search allocates a space to store a set of KOUHO itself. The argument to *J_UD_free* is a pointer obtained from a *J_UD_search* call. After *J_UD_free* is performed, this space is made available for further allocation.

DIAGNOSTICS

J_UD_open returns a dictionary pointer upon successful completion. Otherwise, a NULL pointer is returned and **jlib_errno** is set to indicate the error:

[JUDNOSPC]	The named dictionary cannot be created.
[JUDNOENT]	The named dictionary cannot be opened for reading because it does not exist.
[JUDBADENT]	The named file exists but it is not a user dictionary.
[JUDACCES]	The dictionary exists but permission is denied.
[JUDWRONG]	The format of the dictionary is wrong.
[JUDINVAL]	*Mode* specifies neither **RDONLY** nor **RDWR**.
	J_UD_close returns a value of 0 upon successful completion. Otherwise, a value of -1 is returned and **jlib_errno** is set to indicate the error.
[JUDBADDP]	*Dp* is not a valid dictionary pointer.
	J_UD_store returns a value of 0 upon successful completion. Otherwise, a value of -1 is returned and **jlib_errno** is set to indicate the error.
[JUDBADDP]	*Dp* is not a valid dictionary pointer.
[JUDACCES]	The dictionary exists and write permission is denied.
[JUDNOSPC]	The file system is full.
[JUDINVAL]	*Hinshi* is invalid. *Key* includes illegal characters, or *key* is too long.
	J_UD_delete returns a value of 0 upon successful completion. Otherwise, a value of -1 is returned and **jlib_errno** is set to indicate the error.
[JUDBADDP]	*Dp* is not a valid dictionary pointer.
[JUDACCES]	The dictionary exists and write permission is denied.
[JUDINVAL]	*Key* includes illegal characters, or *key* is too long.

J_UD_search returns a pointer to a UDKouhogun structure upon successful completion. Otherwise, a NULL pointer is returned and **jlib_errno** is set to indicate the error:

[JUDBADDP]	*Dp* is not a valid dictionary pointer open for reading.
[JUDNOSPC]	Not enough space on memory to return the result.
[JUDINVAL]	*Key* includes illegal characters, or *key* is too long.

J_UD_free returns a value of 0 upon successful completion. Otherwise, a value of −1 is returned.

WARNINGS

It is recommended to call *J_UD_free* before a *J_UD_search* call.

J_UD_store, *J_UD_delete*, and *J_UD_search* do not check a lock for a file access.

J_UD_open and *J_UD_store* invoke the command *wdutil*(1).

SEE ALSO

open_jlib(3X), SetUserDict(3X)

NAME

jistosj, jistouj, sjtojis, sjtouj, ujtojis, ujtosj, cjistosj, cjistouj, csjtojis, csjtouj, cujtojis, cujtosj – code set conversion routines for JIS, Shift JIS and UJIS

SYNOPSIS

#include <jcode.h>

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

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

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

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

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

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

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

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

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

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

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

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

DESCRIPTION

Functions, *jistosj*, *jistouj*, *sjtojis*, *sjtouj*, *ujtojis*, and *ujtosj* convert a string from one code set to another (using 8-bit process code). These routines convert the string pointed to by *s2*, store the converted string to the array pointed to by *s1*, and return *s1*. These functions do not check for overflow of *s1*. Validity of the string pointed to by *s2* is assumed, and no checks are made for invalid code in the string.
JIS encoded strings for *s2* are assumed to include proper control sequences (for character set designation). Also, strings converted to JIS by the routines include proper control sequences (for character set designation).

Jistosj converts JIS to SJIS.

Jistouj converts JIS to UJIS.

Sjtojis converts SJIS to JIS.

Sjtouj converts SJIS to UJIS.

Ujtojis converts UJIS to JIS.

Ujtosj converts UJIS to SJIS.

Each of the functions, *cjistosj*, *cjistouj*, *csjtojis*, *csjtouj*, *cujtojis*, and *cujtosj* converts one Kanji character from one code set to another (using 8-bit process code). These routines get one Kanji character from the string pointed to by *s2*, convert it, store the converted character in the array pointed to by *s1*, and return *s1*. The contents of the array pointed to by *s2* is not checked for validity. Also, conversion to JIS does not include addition of control sequences.

Cjistosj converts JIS to SJIS.

Cjistouj converts JIS to UJIS.

Csjtojis converts SJIS to JIS.

Csjtouj converts SJIS to UJIS

Cujtojis converts UJIS to JIS.

Cujtosj converts UJIS to SJIS.

SEE ALSO

iconv(3C)

NAME

KutenZenkaku – translate characters

SYNOPSIS

```
#include <jlib.h>

unsigned char *KutenZenkaku (c, s)
int c;
unsigned char *s;
```

DESCRIPTION

The argument *c* means KUTEN (section-point) code defined as follows:

c = **n** * 10000 + **x** * 100 + **y**;

where **n** is plane number, **x** is section number and **y** is point number.

KutenZenkaku copies the corresponding 16-bit Japanese character in string *s*, terminated by a null character.

DIAGNOSTICS

KutenZenkaku returns *s* upon successful completion. Otherwise, a NULL pointer is returned.

SEE ALSO

open_jlib(3X)

NAME

l3tol, ltol3 – convert between 3-byte integers and long integers

SYNOPSIS

```
void l3tol (lp, cp, n)
long *lp;
char *cp;
int n;

void ltol3 (cp, lp, n)
char *cp;
long *lp;
int n;
```

DESCRIPTION

L3tol converts a list of *n* three-byte integers packed into a character string pointed to by *cp* into a list of long integers pointed to by *lp*.

Ltol3 performs the reverse conversion from long integers (*lp*) to three-byte integers (*cp*).

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

SEE ALSO

fs(4).

WARNINGS

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

STANDARDS CONFORMANCE

l3tol: XPG2

ltol3: XPG2

NAME

langinfo, langtoid, idtolang, currlangid – NLS information about native languages

SYNOPSIS

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

char *langinfo (langid, item)
int langid;
nl_item item;

int langtoid (langname)
const char *langname;

char *idtolang (langid)
int langid;

int currlangid ( )
```

DESCRIPTION

Note. All functions defined on this page are obsolete. Use of *nl_langinfo*(3C) is recommended as a replacement for *langinfo*.

Langinfo returns a pointer to a null-terminated string containing information relevant to a particular language or cultural area defined in the program's locale (see *setlocale*(3C)). *Langinfo* effectively calls *langinit* (see *nl_init*(3C)) to load the program's locale according to the language specified by *langid*.

Currlangid looks for a LANG string in the user's environment. If it finds one, *currlangid* returns the corresponding integer listed in *lang*(5). Otherwise, it returns **0** to indicate a default to native-computer, the method used before NLS was available.

Idtolang takes the integer *langid* and attempts to return the corresponding character string defined in *lang*(5). If *langid* is not found, an empty string is returned.

Langtoid is the inverse of *idtolang*: it attempts to convert a string to a language ID, returning **0** to indicate native-computer if no match is found.

EXTERNAL INFLUENCES

Locale

The string returned by *langinfo* for a particular *item* is determined by the locale category specified for that item in *langinfo*(5).

International Code Set Support

Single- and multi-byte character code sets are supported.

WARNINGS

Langinfo returns a pointer to a static area that is overwritten on each call.

AUTHOR

Langinfo was developed by HP.

SEE ALSO

nl_init(3C), nl_langinfo(3C), setlocale(3C), hpnls(5), lang(5), langinfo(5).

STANDARDS CONFORMANCE

nl_langinfo: XPG2, XPG3

NAME

_ldecvt, _ldfcvt, _ldgcvt – convert long-double floating-point number to string

SYNOPSIS

#include <stdlib.h>

```
char *_ldecvt (value, ndigit, decpt, sign)
long_double value;
int ndigit, *decpt, *sign;

char *_ldfcvt (value, ndigit, decpt, sign)
long_double value;
int ndigit, *decpt, *sign;

char *_ldgcvt (value, ndigit, buf)
long_double value;
int ndigit;
char *buf;
```

DESCRIPTION

_ldecvt converts *value* to a null-terminated string of *ndigit* digits and returns a pointer to the string. The high-order digit is non-zero, unless the value is zero. The low-order digit is rounded. The position of the radix character relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). The radix character 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.

_ldfcvt is identical to *_ldecvt*, except that the correct digit has been rounded for printf "%Lf" (FORTRAN F-format) output of the number of digits specified by *ndigit*.

_ldgcvt converts the *value* to a null-terminated string in the array pointed to by *buf* and returns *buf*. It produces *ndigit* significant digits in FORTRAN F-format if possible, or E-format otherwise. A minus sign, if required, and a radix character will be included in the returned string. Trailing zeros are suppressed. The radix character is determined by the currently loaded NLS environment (see *setlocale*(3C)). If *setlocale* has not been called successfully, the default NLS environment, "C" (see *lang*(5)), is used. The default environment specifies a period (.) as the radix character.

DIAGNOSTICS

NaN is returned for Not-a-Number, and ±*INFINITY* is returned for Infinity.

WARNINGS

The values returned by *_ldecvt* and *_ldfcvt* point to a single static data array whose content is overwritten by each call.

AUTHOR

_ldecvt, _ldfcvt and *_ldgcvt* were developed by HP

SEE ALSO

setlocale(3C), printf(3S), hpnls(5), lang(5).

EXTERNAL INFLUENCES

Locale

The LC_NUMERIC category determines the radix character.

International Code Set Support

Single-byte character code sets are supported.

NAME

localeconv – query the numeric formatting conventions of the current locale

SYNOPSIS

#include <locale.h>

struct lconv *localeconv( );

DESCRIPTION

Localeconv sets the components of an object of type **struct lconv** (defined in **<locale.h>**) with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the program's current locale (see *setlocale(3C)*).

The members of the structure with type char * are strings, any of which (except decimal_point) can point to \`\`'' (the empty string), 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 <limits.h>) to indicate that the value is not available in the current locale. The members include the following:

char *decimal_point
: The decimal-point character used to format non-monetary quantities. This will be the same value as that returned by a call to *nl_langinfo(3C)* with RADIXCHAR as its argument.

char *thousands_sep
: The character used to separate groups of digits to the left of the decimal-point character in formatted non-monetary quantities. This will be the same value as that returned by a call to *nl_langinfo(3C)* with THOUSEP as its argument.

char *grouping
: A string whose elements indicate the size of each group of digits in formatted non-monetary quantities.

char *int_curr_symbol
: The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in *ISO 4217 Codes for the Representation of Currency and Funds.* The fourth character (immediately preceding the null character) is the character used to separate the international currency symbol from the monetary quantity.

char *currency_symbol
: The local currency symbol applicable to the current locale. This value along with positioning information is returned by a call to *nl_langinfo(3C)* with CRNCYSTR as its argument.

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 whose elements indicate the size of each group of digits in formatted monetary quantities.

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 locally formatted monetary quantity.

char p_cs_precedes
Set to 1 or 0 if the **currency_symbol** respectively preceeds 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 preceeds 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.

char n_sign_posn
Set to a value indicating the positioning of the **negative_sign** for a negative formatted monetary quantity.

The elements of **grouping** and **mon_grouping** are interpreted according to the following:

MAX_CHAR No further grouping is to be performed.

0 The previous element is to be repeatedly used for the remainder of the digits.

other The value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits to the left of the current group.

The value of **p_sign_posn** and **n_sign_posn** is interpreted according to the following:

0 Parentheses surround the quantity and **currency_symbol.**

1 The sign string preceeds the quantity and **currency_symbol.**

2 The sign string succeeds the quantity and **currency_symbol.**

3 The sign string immediately preceeds the **currency_symbol.**

4 The sign string immediately succeeds the **currency_symbol.**

The implementation shall behave as if no library function calls the *localeconv* function.

RETURN VALUE

The *localeconv* function returns a pointer to the filled-in **struct lconv.**

EXAMPLES

The following table illustrates the formatting used in five languages for monetary quantities.

Country	Positive format	Negative format	International format
american	$1,234.56	-$1,234.56	USD 1,234.56
italian	L.1.234	-L.1.234	ITL.1.234
dutch	F 1.234,56	F -1.234,56	NLG 1.234,56
norwegian	kr1.234,56	kr1.234,56-	NOK 1.234,56
portuguese	1,234$56	-1,234$56	PTE 1,234$56

For these five languages, the respective values for the monetary members of the structure returned by *localeconv* are:

	american	italian	dutch	norwegian	portuguese
int_curr_symbol	"USD "	"ITL."	"NLG "	"NOK "	"PTE "
currency_symbol	"$"	"L."	"F"	"kr"	"$"
mon_decimal_point	"."	``''	","	","	"$"
mon_thousands_sep	","	"."	"."	"."	","
mon_grouping	"\3"	"\3"	"\3"	"\3"	"\3"
positive_sign	``''	``''	``''	``''	``''
negative_sign	"-"	"-"	"-"	"-"	"-"
int_frac_digits	2	0	2	2	2
frac_digits	2	0	2	2	2
p_cs_precedes	1	1	1	1	0
p_sep_by_space	0	0	1	0	0
n_cs_precedes	1	1	1	1	0
n_sep_by_space	0	0	1	0	0
p_sign_posn	1	1	1	1	1
n_sign_posn	4	1	4	2	1

WARNINGS

The structure returned by *localeconv* should not be modified by the calling program. Calls to the *setlocale(3C)* function with categories LC_ALL, LC_MONETARY, or LC_NUMERIC may overwrite the contents of the structure that *localeconv* points to when it returns.

AUTHOR

Localeconv was developed by HP.

SEE ALSO

hpnls(5), setlocale(3C), langinfo(3C), buildlang(1M)

EXTERNAL INFLUENCES

Locale

The LC_NUMERIC category influences the **decimal_point, thousands_sep,** and **grouping** members of the structure referenced by the pointer returned from a call to *localeconv.*

The LC_MONETARY category influences all of the other members of this structure.

International Code Set Support

Single- and multi-byte character code sets are supported.

STANDARDS CONFORMANCE

localeconv: ANSI C

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.

WARNINGS

Logname returns a pointer to static data that is overwritten by each subsequent call.

This method of determining a login name is subject to forgery.

FILES

/etc/profile

SEE ALSO

env(1), login(1), profile(4), environ(5).

STANDARDS CONFORMANCE

logname: SVID2, XPG2

NAME

lsearch, lfind – linear search and update

SYNOPSIS

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

char *lsearch ((char *)key, (char *)base, nelp, sizeof(*key), compar)
unsigned *nelp;
int (*compar)( );

char *lfind ((char *)key, (char *)base, nelp, sizeof(*key), compar)
unsigned *nelp;
int (*compar)( );
```

DESCRIPTION

Lsearch is a linear search routine generalized from Knuth (6.1) Algorithm S. It returns a pointer into a table indicating where a datum may be found. If the datum does not occur, it is added at the end of the table.

Key points to the datum to be sought in the table.

Base points to the first element in the table.

Nelp points to an integer containing the current number of elements in the table. The integer is incremented if the datum is added to the table.

Compar is the name of the comparison function which the user must supply (*strcmp*, for example). It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal and non-zero otherwise.

Lfind is the same as *lsearch* except that if the datum is not found, it is not added to the table. Instead, a NULL pointer is returned.

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.
The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.
Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

EXAMPLE

This fragment will read in $\leq$ TABSIZE strings of length $\leq$ ELSIZE and store them in a table, eliminating duplicates.

```
#include <stdio.h>

#define TABSIZE 50
#define ELSIZE 120

        char line[ELSIZE], tab[TABSIZE][ELSIZE], *lsearch( );
        unsigned nel = 0;
        int strcmp( );
        . . .
        while (fgets(line, ELSIZE, stdin) != NULL &&
           nel < TABSIZE)
              (void) lsearch(line, (char *)tab, &nel,
```

```
                    ELSIZE, strcmp);
      . . .
```

SEE ALSO

bsearch(3C), hsearch(3C), tsearch(3C).

DIAGNOSTICS

If the searched for datum is found, both *lsearch* and *lfind* return a pointer to it. Otherwise, *lfind* returns NULL and *lsearch* returns a pointer to the newly added element.

BUGS

Undefined results can occur if there is not enough room in the table to add a new item.

STANDARDS CONFORMANCE

lsearch: SVID2, XPG2, XPG3

lfind: SVID2, XPG2, XPG3

NAME

ltostr, ultostr, ltoa, ultoa – convert long integers to strings

SYNOPSIS

char *ltostr (n, base)
long n;
int base;

char *ultostr (n, base)
unsigned long n;
int base;

char *ltoa (n)
long n;

char *ultoa (n)
unsigned long n;

DESCRIPTION

The functions *ltostr* and *ultostr* convert a signed or unsigned long integer to the corresponding string representation in the specified base. The argument *base* must be between 2 and 36, inclusive.

The functions *ltoa* and *ultoa* convert a signed or unsigned long integer to the corresponding base 10 string representation, returning a pointer to the result.

These functions are smaller and faster than using *sprintf(3C)* for simple conversions.

WARNINGS

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

ERRORS

If the value of *base* is not between 2 and 36, *ltostr* and *ultostr* return the value **NULL** and set the external variable *errno* to **ERANGE**.

AUTHOR

Ltostr, *ultostr*, *ltoa* and *ultoa* were developed by HP.

SEE ALSO

printf(3C), strtol(3C).

NAME

malloc, calloc, realloc, free – main memory allocator

SYNOPSIS

#include <stdlib.h>

void *malloc (size)
size_t size;

void *calloc (nelem, elsize)
size_t nelem, elsize;

void *realloc (ptr, size)
void *ptr;
size_t size;

void free (ptr)
void *ptr;

DESCRIPTION

The set of *malloc* functions provide a simple, general-purpose memory allocation package.

Malloc allocates space for a block of at least *size* bytes; the space is not initialized.

Calloc allocates space for an array of *nelem* elements, each of size *elsize* bytes; the space is initialized to zeros.

Realloc changes the size of the block pointed to by *ptr*, a pointer to a block previously allocated by *malloc*, *calloc*, or *realloc*, to *size* bytes. The contents will be unchanged up to the lesser of the new and old sizes. If no free block of *size* bytes is available, *realloc* will call *malloc* to allocate a block of *size* bytes and will then move the data to the new space. If *ptr* is a NULL pointer, *realloc* behaves as *malloc*(*size*). If *size* is zero and *ptr* is not a NULL pointer, *realloc* behaves as *free*(*ptr*).

Free deallocates the space pointed to by *ptr*, a pointer to a block previously allocated by *malloc*, *calloc*, or *realloc*; the space is made available for further allocation, but its contents are left undisturbed. If *ptr* is a NULL pointer, no action occurs.

RETURN VALUE

Malloc, *calloc*, and *realloc* return a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

DIAGNOSTICS

Any error condition listed for *brk*(2) is considered to mean no available memory. *Malloc*, *realloc* and *calloc* return a NULL pointer if there is no available memory or if the memory being managed by *malloc* has been detectably corrupted. If this happens, the block pointed to by *ptr* may be destroyed.

WARNINGS

Results are undefined if the space assigned by the allocation functions is overrun.

Sbrk (see *brk*(2)) is called, as needed, to get memory from the system.

Free and *realloc* do not check their pointer argument for validity.

Allocation time is proportional to the number of allocated but un-freed objects. If a program allocates but never frees, each successive allocation takes longer. For an alternate, more flexible implementation, see *malloc*(3X).

SEE ALSO

brk(2), malloc(3X).

NAME

malloc, free, realloc, calloc, mallopt, mallinfo – fast main memory allocator

SYNOPSIS

#include <malloc.h>

char *malloc (size)
unsigned size;

void free (ptr)
char *ptr;

char *realloc (ptr, size)
char *ptr;
unsigned size;

char *calloc (nelem, elsize)
unsigned nelem, elsize;

int mallopt (cmd, value)
int cmd, value;

struct mallinfo mallinfo ()

DESCRIPTION

Malloc and *free* provide a simple general-purpose memory allocation package, which runs considerably faster than the *malloc*(3C) package. It is found in the library "malloc", and is loaded if the option "–lmalloc" is used with *cc*(1) or *ld*(1).

Malloc returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to *free* is a pointer to a block previously allocated by *malloc;* after *free* is performed this space is made available for further allocation, and its contents will usually have been destroyed (but see *mallopt* below for a way to change this behavior).

Undefined results will occur if the space assigned by *malloc* is overrun or if some random number is handed to *free*.

Realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If *ptr* is a null pointer, the *realloc* function behaves like the *malloc* function for the specified size. If *size* is zero and *ptr* is not a null pointer the object it points to is freed and NULL is returned.

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 and other options in the *malloc*(3X) package. 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 48.

M_NLBLKS Set *numlblks* to *value*. The above mentioned "large groups" each contain *numlblks* blocks. *Numlblks* must be greater than 1. The default value for *numlblks* is 100.

M_GRAIN Set *grain* to *value*. The sizes of all blocks smaller than *maxfast* are considered to be rounded up to the nearest multiple of *grain*. *Grain* must be greater than 0. The default value of *grain* is the smallest number of bytes which will allow alignment of any data type. Value will be rounded up to a multiple of the default when *grain* is set.

M_KEEP Preserve data in a freed block until the next *malloc*, *realloc*, or *calloc*. This option is provided only for compatibility with the old version of *malloc* and is not recommended.

M_BLOCK Block all blockable signals in *malloc*, *realloc*, *calloc*, and *free*. This option is provided for those who need to write signal handlers that allocate memory. When set, the *malloc*(3X) package becomes completely re-entrant. The default action is to NOT block all blockable signals.

M_UBLOCK Don't block all blockable signals in *malloc*, *realloc*, *calloc*, and *free*. This option cancels signal blocking initiated by the M_BLOCK option.

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 (unless *cmd* is set to M_BLOCK or M_UBLOCK).

Mallinfo provides instrumentation describing space usage, but may not be called until the first small block is allocated. 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.

DIAGNOSTICS

Malloc, *realloc* and *calloc* return a NULL pointer if there is not enough available memory. Any error condition listed for *brk*(2) is considered to mean no available memory. When *realloc* returns NULL, the block pointed to by *ptr* is left intact. If *mallopt* is called after any allocation of a small block and *cmd* is not set to M_BLOCK or M_UBLOCK or if *cmd* or *value* are invalid, non-zero is returned. Otherwise, it returns zero.

WARNINGS

This package usually uses more data space than *malloc*(3C).
The code size is also bigger than *malloc*(3C).
Note that unlike *malloc*(3C), this package does not preserve the contents of a block when it is freed, unless the M_KEEP option of *mallopt* is used.
Undocumented features of *malloc*(3C) have not been duplicated.

SEE ALSO

brk(2), malloc(3C).

STANDARDS CONFORMANCE

malloc: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C *calloc*: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

free: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

mallinfo: SVID2, XPG2

mallopt: SVID2, XPG2

realloc: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

matherr – error-handling function

SYNOPSIS

#include <math.h>

int matherr (x);
struct exception *x;

DESCRIPTION

Matherr is invoked by functions in the Math Library when errors are detected. Users can define their own procedures for handling errors, by including a function named *matherr* in their programs. *Matherr* must be of the form described above. When an error occurs, a pointer to the exception structure *x* is passed to the user-supplied *matherr* function. This structure, which is defined in the *<math.h>* header file, is as follows:

```
struct exception {
        int type;
        char *name;
        double arg1, arg2, retval;
};
```

The element *type* is an integer describing the type of error that has occurred, from the following list of constants (defined in the header file):

DOMAIN	argument domain error
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 nonzero, 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.

DEPENDENCIES

Series 800 (ANSI C /lib/libM.a)

In the ANSI C **/lib/libM.a**, *matherr*() has been renamed to *_matherr*() and no error messages are printed to the standard error output.

EXAMPLES

```
#include <math.h>

int
matherr(x)
register struct exception *x;
{
        switch (x->type) {
```

```
        case DOMAIN:
                /* change sqrt to return sqrt(-arg1), not 0 */
                if (!strcmp(x->name, "sqrt")) {
                        x->retval = sqrt(-x->arg1);
                        return (0); /* print message and set errno */
                }
        case SING:
                /* all other domain or sing errors, print message and abort */
                fprintf(stderr, "domain error in %s\n", x->name);
                abort( );
        case PLOSS:
                /* print detailed error message */
                fprintf(stderr, "loss of significance in %s(%g) = %g\n",
                        x->name, x->arg1, x->retval);
                return (1); /* take no other action */
        }
        return (0); /* all other errors, execute default procedure */
}
```

DEFAULT ERROR HANDLING PROCEDURES						
	Types of Errors					
type	DOMAIN	SING	OVERFLOW	UNDERFLOW	TLOSS	PLOSS
errno	EDOM	EDOM	ERANGE	ERANGE	ERANGE	ERANGE
BESSEL:	–	–	–	–	M, 0	*
y0, y1, yn (arg ≤ 0)	M, –H	–	–	–	–	–
EXP:	–	–	H	0	–	–
LOG, LOG10:						
(arg < 0)	M, –H	–	–	–	–	–
(arg = 0)	–	M, –H	–	–	–	–
POW:						
neg ** non-int	–	–	±H	0	–	–
0 ** non-pos	M, 0	–	–	–	–	–
SQRT:	M, 0	–	–	–	–	–
GAMMA:	–	M, H	H	–	–	–
HYPOT:	–	–	H	–	–	–
SINH:	–	–	±H	–	–	–
COSH:	–	–	H	–	–	–
SIN, COS, TAN:	–	–	–	–	M, 0	*
ASIN, ACOS, ATAN2:	M, 0	–	–	–	–	–

ABBREVIATIONS	
*	As much as possible of the value is returned.
M	Message is printed (EDOM error) (except for s800 libM.a).
H	HUGE is returned.
–H	–HUGE is returned.
±H	HUGE or –HUGE is returned.
0	0 is returned.

STANDARDS CONFORMANCE

matherr: SVID2, XPG2

NAME

memccpy, memchr, memcmp, memcpy, memmove, memset – memory operations

SYNOPSIS

#include <string.h>

```
void *memccpy (s1, s2, c, n)
void *s1;
const void *s2;
int c;
size_t n;

void *memchr (s, c, n)
const void *s;
int c;
size_t n;

int memcmp (s1, s2, n)
const void *s1, *s2;
size_t n;

void *memcpy (s1, s2, n)
void *s1;
const void *s2;
size_t n;

void *memmove (s1, s2, n)
void *s1;
const void *s2;
size_t n;

void *memset (s, c, n)
void *s;
int c;
size_t n;
```

DESCRIPTION

These functions operate as efficiently as possible on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

Definitions for all these functions, the type **size_t**, and the constant **NULL** are provided in the **<string.h>** header.

Memccpy copies characters from the object pointed to by *s2* into the object pointed to by *s1*, stopping after the first occurrence of character *c* has been copied, or after *n* characters have been copied, whichever comes first. If copying takes place between objects that overlap, the behavior is undefined. It returns a pointer to the character after the copy of *c* in *s1*, or a **NULL** pointer if *c* was not found in the first *n* characters of *s2*.

Memchr locates the first occurrence of *c* (converted to an **unsigned char**) in the initial *n* characters (each interpreted as **unsigned char**) of the object pointed to by *s*. It returns a pointer to the located character, or a **NULL** pointer if the character does not occur in the object.

Memcmp compares the first *n* characters of the object pointed to by *s1* to the first *n* characters of the object pointed to by *s2*. It returns an integer greater than, equal to, or less than zero, according as the object pointed to by *s1* is greater than, equal to, or less than the object pointed to by *s2*. The sign of a nonzero return value is determined by the sign of the difference between the values of the first pair of characters (both interpreted as **unsigned char**) that differ in the objects being compared.

Memcpy copies *n* characters from the object pointed to by *s2* into the object pointed to by *s1*. If copying takes place between objects that overlap, the behavior is undefined. It returns the value of *s1*.

Memmove copies *n* characters from the object pointed to by *s2* into the object pointed to by *s1*. Copying takes place as if the *n* characters from the object pointed to by *s2* are first copied into a temporary array of *n* characters that does not overlap the objects pointed to by *s1* and *s2*, and then the *n* characters from the temporary array are copied into the object pointed to by *s1*. It returns the value of *s1*.

Memset copies the value of *c* (converted to an **unsigned char**) into each of the first *n* characters of the object pointed to by *s*. It returns the value of *s*.

International Code Set Support

These functions support only single-byte character code sets.

WARNING

These functions were previously defined in <**memory.h**>.

SEE ALSO

string(3C)

STANDARDS CONFORMANCE

memccpy: SVID2, XPG2, XPG3

memchr: SVID2, XPG2, XPG3, ANSI C

memcmp: SVID2, XPG2, XPG3, ANSI C

memcpy: SVID2, XPG2, XPG3, ANSI C

memmove: ANSI C

memset: SVID2, XPG2, XPG3, ANSI C

NAME

mkfifo – make a FIFO special file

SYNOPSIS

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

int mkfifo (path, mode)
char *path;
mode_t mode;

DESCRIPTION

Mkfifo creates a new named pipe, a FIFO (first-in-first-out) file named by the path name pointed to by *path*. The file permission bits of the new FIFO are initialized from *mode*. The file permission bits of the *mode* argument are modified by the process's file creation mask: for each bit set in the process's file mode creation mask, the corresponding bit in the new file's mode is cleared (see *umask*(2)). Bits in *mode* other than the file permission bits are ignored.

The FIFO's owner ID is set to the process's effective-user-ID. The FIFO's group ID is set to the group ID of the parent directory if the set-group-ID bit is set on that directory. Otherwise the FIFO's group ID is set to the process's effective group ID.

For details of the I/O behavior of pipes see *read*(2) and *write*(2).

The following symbolic constants are defined in the **<sys/stat.h>** header, and should be used to construct the value of the *mode* argument. The value passed should be the bitwise inclusive OR of the desired permissions:

S_IRUSR	Read by owner.
S_IWUSR	Write by owner.
S_IRGRP	Read by group.
S_IWGRP	Write by group.
S_IROTH	Read by other users.
S_IWOTH	Write by other users.

RETURN VALUE

Upon successful completion a value of zero is returned. Otherwise, a value of **−1** is returned, no FIFO is created, and **errno** is set to indicate the error.

ERRORS

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

[ENOSPC] Not enough space on the file system.

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] A component of the path prefix does not exist.

[EROFS] The directory in which the file is being created is located in a read-only file system.

[EACCES] A component of the path prefix denies search permission.

[EEXIST] The named file exists.

[EFAULT] *Path* points outside the process's allocated address space. The reliable detection of this error is implementation dependent.

[ENOENT] *Path* is null.

[ENAMETOOLONG]
The length of the specified path name exceeds PATH_MAX bytes, or the length of a component of the path name exceeds NAME_MAX bytes while

_POSIX_NO_TRUNC is in effect.

[ELOOP] Too many symbolic links are encountered in translating the path name.

SEE ALSO

mknod(1M), chmod(2), exec(2), mknod(2), pipe(2), stat(2), umask(2), cdf(4), fs(4), mknod(4), stat(5).

AUTHOR

Mkfifo was developed by HP and conforms to the IEEE Standard POSIX 1003.1-1988.

STANDARDS CONFORMANCE

mkfifo: XPG3, POSIX.1, FIPS 151-1

NAME

mktemp – make a unique file name

SYNOPSIS

char *mktemp (template)
char *template;

DESCRIPTION

Mktemp replaces the contents of the string pointed to by *template* by a unique file name, and returns the address of *template*. The string in *template* should look like a file name with six trailing **X**s; *mktemp* will replace the **X**s with a letter and the current process ID. The letter will be chosen so that the resulting name does not duplicate the name of an existing file. If there are less than 6 **X**s, the letter will be dropped first, and then high order digits of the process ID will be dropped.

RETURN VALUE

Mktemp returns its argument except when it runs out of letters, in which case the result is a pointer to the empty string "".

SEE ALSO

getpid(2).

SEE ALSO

getpid(2), tmpfile(3S), tmpnam(3S).

BUGS

It is possible to run out of letters.

Mktemp does not check to see if the file name part of *template* exceeds the maximum length of a file name.

STANDARDS CONFORMANCE

mktemp: SVID2, XPG2

NAME

monitor – prepare execution profile

SYNOPSIS

#include <mon.h>

void monitor (lowpc, highpc, buffer, bufsize, nfunc)
int (*lowpc)( ), (*highpc)( );
WORD *buffer;
int bufsize, nfunc;

DESCRIPTION

An executable program created by **cc −p** automatically includes calls for *monitor* with default parameters; *monitor* need not be called explicitly except to gain fine control over profiling.

Monitor is an interface to *profil*(2). *Lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a (user supplied) array of *bufsize* WORDs (defined in the *<mon.h>* header file). *Monitor* arranges to record a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. The lowest address sampled is that of *lowpc* and the highest is just below *highpc*. *Lowpc* may not equal 0 for this use of *monitor*. At most *nfunc* call counts can be kept; only calls of functions compiled with the profiling option **−p** of *cc*(1) are recorded. (The C Library and Math Library supplied when **cc −p** is used also have call counts recorded.)

For results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use

```
extern etext;
...
monitor ((int (*)())2, ((int(*)())& etext, buf, bufsize, nfunc);
```

Etext lies just above all the program text; see *end*(3C).

To stop execution monitoring and write the results on the file **mon.out**, use

```
monitor ((int (*)())0, (int(*)())0, 0, 0, 0);
```

Prof(1) can then be used to examine the results.

FILES

/lib/libp/libc.a
/lib/libp/libm.a
mon.out

SEE ALSO

cc(1), prof(1), profil(2), end(3C).

STANDARDS CONFORMANCE

monitor: SVID2, XPG2

NAME

mblen, mbtowc, mbstowcs, wctomb, wcstombs – multibyte characters and strings conversions

SYNOPSIS

```
#include <stdlib.h>

int mblen(s, n)
const char *s;
size_t n;

int mbtowc(pwc, s, n)
wchar_t *pwc;
const char *s;
size_t n;

int wctomb(s, wchar)
char *s;
wchar_t wchar;

size_t mbstowcs(pwcs, s, n)
wchar_t *pwcs;
const char *s;
size_t n;

size_t wcstombs(s, pwcs, n)
char *s;
const wchar_t *pwcs;
size_t n;
```

DESCRIPTION

A multibyte character is composed of one or more bytes that represent a "whole" character in a character encoding. A wide character (type of *wchar_t*) is composed of a fixed number of bytes whose code value can represent any character in a character encoding.

The *mblen* function determines the number of bytes in the multibyte character pointed to by *s*. It is equivalent to

```
mbtowc((wchar_t *)0, s, n);
```

If *s* is a null pointer, the *mblen* function returns a nonzero or zero value, depending on whether the multibyte character encodings do or do not have state-dependent encodings, respectively. Since no character encodings currently supported by HP-UX are state-dependent, zero is always returned in this case. However, for maximum portability to other systems, application programs should not depend on this.

If *s* is not a null pointer, the *mblen* function returns the number of bytes in the multibyte character if the next *n* or fewer bytes form a valid multibyte character or return −1 if they do not form a valid multibyte character. If *s* points to the null character, the *mblen* function returns 0.

The *mbtowc* function determines the number of bytes in the multibyte character pointed to by *s*, determines the code for the value of type *wchar_t* corresponding to that multibyte character, and then stores the code in the object pointed to by *pwc*. The value of the code corresponding to the null character is zero. At most *n* characters are examined, starting at the character pointed to by *s*.

If *s* is a null pointer, the *mbtowc* function returns a nonzero or zero value, depending on whether the multibyte character encodings do or do not have state-dependent encodings, respectively. Since no character encodings currently supported by HP-UX are state-dependent, zero is always returned in this case. However, for maximum portability to other systems, application programs should not depend on this.

If *s* is not a null pointer, the *mbtowc* function returns the number of bytes in the converted multibyte character if the next *n* or fewer bytes form a valid multibyte character, or −1 if they do not form a valid multibyte character. If *s* points to the null character, the *mbtowc* function returns 0. The value returned is never greater than *n* or the value of the MB_CUR_MAX macro.

The *wctomb* function determines the number of bytes needed to represent the multibyte character corresponding to the code whose value is *wchar* and stores the multibyte character representation in the array object pointed to by *s*. At most MB_CUR_MAX characters are stored.

If *s* is a null pointer, the *wctomb* function returns a nonzero or zero value, depending on whether the multibyte character encodings do or do not have state-dependent encodings, respectively. Since no character encodings currently supported by HP-UX are state-dependent, zero is always returned in this case. However, for maximum portability to other systems, application programs should not depend on this.

If *s* is not a null pointer, the *wctomb* function returns the number of bytes in the multibyte character corresponding to the value of *wchar*, or −1 if the value of *wchar* does not correspond to a valid multibyte character. The value returned is never greater than the value of the MB_CUR_MAX macro.

The *mbstowcs* function converts a sequence of multibyte characters from the array pointed to by *s* into a sequence of corresponding codes and stores these codes into the array pointed to by *pwcs*, stopping after either *n* codes or a code with value zero (a converted null character) is stored. Each multibyte character is converted as if by a call to the *mbtowc* function. No more than *n* elements are modified in the array pointed to by *pwcs*.

If an invalid multibyte character is encountered, the *mbstowcs* function returns (size_t)-1. Otherwise, the *mbstowcs* function returns the number of array elements modified, not including a terminating zero code, if any. The array is not null- or zero-terminated if the value returned is *n*.

The *wcstombs* function converts a sequence of codes corresponding to multibyte characters from the array pointed to by *pwcs* into a sequence of multibyte characters and stores them into the array pointed to by *s*, stopping if a multibyte character exceeds the limit of *n* total bytes or if a null character is stored. Each code is converted as if by a call to the *wctomb* function. No more than *n* bytes are modified in the array pointed to by *s*.

If a code is encountered that does not correspond to a valid multibyte character, the *wcstombs* function returns (size_t) − 1. Otherwise, the *wcstombs* function returns the number of bytes modified, not including a terminating null character, if any. The array is not null- or zero-terminated if the value returned is *n*.

Locale

The LC_CTYPE category determines the behavior of the multibyte character and string functions.

WARNINGS

With the exception of ASCII characters, the code values of wide characters (type of wchar_t) are specific to the effective locale specified by the LC_CTYPE environment variable. These values may not be compatible with values obtained by specifying other locales which are supported now, or which may be supported in the future. It is recommended that wide character constants and wide string literals (see the *C Reference Manual*) not be used and that wide character code values not be stored in files or devices because future standards may dictate changes in the code value assignments of the wide characters. However, wide character constants and wide string literals corresponding to the characters of the ASCII code set can be safely used since their values are guaranteed to be the same as their ASCII code set values.

AUTHOR

Multibyte was developed by HP.

SEE ALSO

setlocale(3C), nl_tools_16(3C).

STANDARDS CONFORMANCE

mblen: ANSI C

mbstowcs: ANSI C

mbtowc: ANSI C

wcstombs: ANSI C

wctomb: ANSI C

NAME

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 (block = 1024 bytes)) 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. The functions can be accessed by giving the **–lndbm** option to *ld*(1) or *cc*(1).

Key and *content* parameters are described by the **datum** type. 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 of keys 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 and success with zero. Routines that return a **datum** indicate errors with a null *dptr*. If *dbm_store* is called with a *flags* value of **DBM_INSERT** and finds an existing entry with the same key, a value of **1** is returned.

WARNINGS

The **.pag** file will contain holes so that its apparent size is about four times its actual content. Some older UNIX systems create real file blocks for these holes when touched. These files cannot be copied by normal means (such as *cp*(1), *cat*(1), *tar*(1), or *ar*(1)) without expansion.

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. *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.

AUTHOR

Ndbm(3X) was developed by the University of California, Berkeley.

SEE ALSO

dbm(3X).

NAME

nl_toupper, nl_tolower – translate characters for use with NLS

SYNOPSIS

int nl_toupper (c, langid)
int c, langid;

int nl_tolower (c, langid)
int c, langid;

DESCRIPTION

These routines are extensions of their counterparts in *conv*(3C). They function in the same way, but have a *langid* parameter (see *lang*(5)) whose value represents a supported language. If *langid* is not valid, or if the NLS environment corresponding to *langid* is not available, "n-computer", the default NLS environment associated with *langinit*(3C), is used (see *nl_init*(3C)).

WARNINGS

These routines are provided for historical reasons only. Use of the routines in *conv*(3C), which now provide for international support via *setlocale*(3C), is recommended.

Nl_toupper and *nl_tolower* effectively call *langinit* to load the NLS environment according to the language specified by *langid*.

AUTHOR

Nl_conv was developed by the Hewlett-Packard Company.

SEE ALSO

conv(3C), nl_init(3C), hpnls(5), lang(5).

EXTERNAL INFLUENCES

Locale

The LC_CTYPE category determines the translations to be done.

International Code Set Support

Single-byte character code sets are supported.

NAME

nl_isalpha, nl_isupper, nl_islower, nl_isdigit, nl_isxdigit, nl_isalnum, nl_isspace, nl_ispunct, nl_isprint, nl_isgraph, nl_iscntrl – classify characters for use with NLS

SYNOPSIS

#include <nl_ctype.h>

int nl_isalpha (c, langid)
int c; int langid;

. . .

DESCRIPTION

These routines classify character-coded integer values by table lookup. *Langid* corresponds to a particular NLS environment (see *lang*(5)). Each is a predicate returning nonzero for true, zero for false. All are defined for the range **–1** to **255**. If *langid* is not defined, or if the NLS environment corresponding to *langid* is not available, "n-computer", the default NLS environment associated with *langinit*(3C), is used (see *nl_init*(3C)).

nl_isalpha — *c* is a letter.

nl_isupper — *c* is an uppercase letter.

nl_islower — *c* is a lowercase letter.

nl_isdigit — *c* is a decimal digit (in ASCII: characters [0-9]).

nl_isxdigit — *c* is a hexadecimal digit (in ASCII: characters [0-9], [A-F] or [a-f]).

nl_isalnum — *c* is an alphanumeric (letters or digits).

nl_isspace — *c* is a character that creates "white space" in displayed text (in ASCII: space, tab, carriage return, new-line, vertical tab, and form-feed).

nl_ispunct — *c* is a punctuation character (in ASCII: any printing character except the space character (040), digits, letters.)

nl_isprint — *c* is a printing character.

nl_isgraph — *c* is a visible character (in ASCII: printing characters, excluding the space character (040)).

nl_iscntrl — *c* is a control character (in ASCII: character codes less than 040 and the delete character (0177)).

DIAGNOSTICS

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

WARNINGS

These macros are provided for historical reasons only. Use of the macros in *ctype*(3C), which now provide for international support via *setlocale*(3C), is recommended.

The *nl_ctype*(3C) macros call *langinit* to load the NLS environment according to the language specified by *langid*.

AUTHOR

Nl_ctype was developed by the Hewlett-Packard Company.

SEE ALSO

ctype(3C), nl_init(3C), hpnls(5), lang(5).

EXTERNAL INFLUENCES

Locale

The LC_CTYPE category determines the classification of character type.

International Code Set Support

Single-byte character code sets are supported.

NAME

nl_init, langinit – initialize the NLS environment of a program

SYNOPSIS

int nl_init(langname)
char *langname;

int langinit(langname)
char *langname;

DESCRIPTION

Nl_init initializes the NLS (Native Language Support) environment of a program to the language specified by *langname*. If *langname* is null or points to an empty string, the default-mode language, "n-computer" (see *lang*(5)), is initialized.

Nl_init affects the behavior of the macros and routines defined in *conv*(3C), *ctime*(3C), *ctype*(3C), *ecvt*(3C), *langinfo*(3C), *multibyte*(3C), *nl_langinfo*(3C), *nl_string*(3C), *nl_tools_16*(3C), *printf*(3S), *printmsg*(3C), *scanf*(3S), *strftime*(3C), *string*(3C), *strtod*(3C), and *vprintf*(3S).

Typically, *nl_init* is used to bind program operation to the end-user's specified language requirements. For example,

```
nl_init( getenv("LANG") );
```

Prior to successfully calling *nl_init*, functions supporting NLS operate as though the default-mode language "n-computer" had been initialized.

Langinit performs the same initialization of the environment control areas as does *nl_init*. However, *nl_init* and *langinit* differ in the action taken when the requested language environment cannot be initialized (see ERRORS below).

RETURN VALUE

0 (zero) will be returned if the environment is successfully initialized to the requested language. Otherwise, **−1** will be returned.

ERRORS

Nl_init will fail if the string specified by *langname* does not identify a valid language name (see *lang*(3C)), or the language is not available on the system.

If *nl_init* fails but had previously succeeded, operation will continue with the environment initialized by the last successful call. If *nl_init* fails and has never been called successfully, the environment will revert to the default-mode language "n-computer".

If *langinit* fails, the environment will revert to the default-mode language "n-computer".

WARNINGS

Nl_init and *langinit* are provided for historical reasons only. Use *setlocale* instead (see *setlocale*(3C)). While the default processing language for *setlocale* is "C", the default processing language for *nl_init* is "n-computer". This is maintained for backward portability.

Langinit is implicitly called by the macros and routines which use a *langid* parameter (see *ctime*(3C), *langinfo*(3C), *nl_conv*(3C), *nl_ctype*(3C), *nl_string*(3C), and *strtod*(3C)). Using any *langid* parameter routine or macro will initialize the environment of the associated language name, thus affecting the behavior of other routines that interact with the NLS environment. For maximum portability and performance, use of macros and routines without the *langid* parameter is recommended.

AUTHOR

Nl_init was developed by HP.

SEE ALSO

conv(3C), ctime(3C), ctype(3C), ecvt(3C), langinfo(3C), multibyte(3C), nl_conv(3C), nl_ctype(3C), nl_langinfo(3C), nl_string(3C), nl_tools_16(3C), printf(3S), printmsg(3C), scanf(3S), string(3C), strtod(3C), vprintf(3S), environ(5), hpnls(5), lang(5), nl_langinfo(5).

STANDARDS CONFORMANCE

nl_init: XPG2

NAME

nl_langinfo – language information

SYNOPSIS

#include <nl_types.h>
#include <langinfo.h>

char *nl_langinfo (item)
nl_item item;

DESCRIPTION

Nl_langinfo returns a pointer to a null-terminated string containing information relevant to a particular language or cultural area defined in the program's locale (see *setlocale*(3C)). The manifest constant names and values of *item* are defined in **<langinfo.h>**. For example:

nl_langinfo(ABDAY_1)

would return a pointer to the string "Dom" if the language identified by the current locale was Portuguese, and "Sun" if the identified language was Finnish.

If an invalid *item* is specified, a pointer to an empty string is returned. An empty string can also be returned for a valid *item* if that *item* is not applicable to the language or customs of the current locale. For example, a thousands separator is not used when writing numbers according to the customs associated with the Arabic language.

EXTERNAL INFLUENCES

Locale

The string returned for a particular *item* is determined by the locale category specified for that item in *langinfo*(5).

International Code Set Support

Single- and multi-byte character code sets are supported.

WARNINGS

Nl_langinfo returns a pointer to a static area that is overwritten on each call.

AUTHOR

Nl_langinfo was developed by HP.

SEE ALSO

localeconv(3C), setlocale(3C), hpnls(5), langinfo(5).

NAME

strcmp8, strncmp8, strcmp16, strncmp16 – non-ASCII string collation

SYNOPSIS

int strcmp8 (s1, s2, langid, status)
unsigned char *s1, *s2;
int langid,*status;

int strncmp8 (s1, s2, n, langid, status)
unsigned char *s1, *s2;
int n, langid, *status;

int strcmp16 (s1, s2, file_name, status)
unsigned char *s1, *s2, *file_name;
int *status;

int strncmp16 (s1, s2, n, file_name, status)
unsigned char *s1, *s2, *file_name;
int n, *status;

DESCRIPTION

Strcmp8 compares string *s1* and *s2* according to the collating sequence of the NLS environment specified by *langid* (see *lang*(5)). If *langid* is invalid, or if the NLS environment corresponding to *langid* is unavailable, "n-computer", the default NLS environment associated with *langinit*(3C) is used (see *nl_init*(3C)). An integer greater than, equal to, or less than 0 is returned, depending on whether *s1* is, respectively, greater than, equal to, or less than *s2*. Trailing blanks in strings *s1* and *s2* are ignored. *Strncmp8* makes the same comparison but looks at a maximum of *n* characters.

Strcmp16 compares strings *s1* and *s2* and returns an integer greater than, equal to, or less than 0 depending on whether *s1* is, respectively, greater than, equal to, or less than *s2*. Strings *s1* and *s2* can contain 16-bit characters mixed with 7-bit and 8-bit characters (see *hpnls*(5)). Strings *s1* and *s2* are compared with 8-bit characters collating before 16-bit characters. *Strncmp16* makes the same comparison, but looks at a maximum of *n* characters.

Nl_init (see *nl_init*(3C)) must be called before the first call to *strcmp16* or *strncmp16*.

ERRORS

If an error condition is encountered, the integer pointed to by *status* is set to one of the non-zero values (listed below) defined in **<langinfo.h>**. For ENOCFFILE and ENOLFILE, **errno** indicates that a file system call failed.

[ENOCFFILE] Access of the file **/usr/lib/nls/config** has failed.

[ENOCONV] The entry for the language sought is not in the file **/usr/lib/nls/config**.

[ENOLFILE] Access of the NLS environment corresponding to *langid* or *file_name* has failed.

WARNINGS

These routines are provided for historical reasons only. Use the *strcoll*(3C) routine instead (see *string*(3C)). However, note that all characters are significant to *strcoll*, whereas *strcmp8* and *strncmp8* ignore trailing blanks.

Strcmp16 and *Strncmp16* do not support a collation sequence table. (A null string must be passed as *file_name* to maintain the correct argument count.)

Strcmp8 and *strncmp8* call *langinit* (see *nl_init*(3C)) to load the NLS environment according to the language specified by *langid*.

AUTHOR

Nl_string was developed by HP.

SEE ALSO

nl_init(3C), string(3C), hpnls(5), lang(5).

EXTERNAL INFLUENCES

Locale

The LC_CTYPE category determines the interpretation of the bytes within the string arguments to the *strcmp8*, *strncmp8*, *strcmp16* , and *strncmp16* functions as single- and/or multi-byte characters.

The LC_COLLATE category determines the collation ordering used by the *strcmp8* and *strncmp8* functions. See *hpnls*(5) for a description of supported collation features. See *nlsinfo*(1) to view the collation used for a particular locale.

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

firstof2, secof2, byte_status, FIRSTof2, SECof2, BYTE_STATUS, CHARAT, ADVANCE, CHARADV, WCHAR, WCHARADV, PCHAR, PCHARADV – tools to process 16-bit characters

SYNOPSIS

```
int firstof2(c)
int c;

int secof2(c)
int c;

int byte_status(c, laststatus)
int c, laststatus;

#include <nl_ctype.h>

FIRSTof2(c)
int c;

SECof2(c)
int c;

BYTE_STATUS(c, laststatus)
int c, laststatus;

CHARAT(p)
char *p;

ADVANCE(p)
char *p;

CHARADV(p)
char *p;

WCHAR(c, p)
int c;
char *p;

WCHARADV(c, p)
int c;
char *p;

PCHAR(c, p)
int c;
char *p;

PCHARADV(c, p)
int c;
char *p;
```

DESCRIPTION

The following macros and routines perform their operations based upon the loaded NLS environment (see *setlocale*(3C)).

FIRSTof2 takes a byte and returns a non-zero value if it can be the first byte of a two-byte character according to the NLS environment loaded, and zero if it cannot.

SECof2 takes a byte and returns a non-zero value if it can be the second byte of a two-byte character according to the loaded NLS environment, and zero if it cannot.

BYTE_STATUS returns one of the following values based on the value of the current byte in *c* and the status of the previous byte interpreted in *laststatus* as returned by the last call to *BYTE_STATUS*. These are the status values as defined in <**nl_ctype.h**>:

 NL_TOOLS_16(3C)

ONEBYTE	single-byte character
SECOF2	second byte of two-byte character
FIRSTOF2	first byte of two-byte character

To validate a two-byte character, both the first and second bytes must be valid. If the value of *laststatus* is **FIRSTOF2** but *SECof2(c)* returns false, *BYTE_STATUS(c, laststatus)* will return **ONEBYTE**.

For the macros *FIRSTof2*, *SECof2*, and *BYTE_STATUS* results are undefined for values of c less than **−1 (EOF)** or greater than **255**.

CHARAT takes as an argument a pointer "p", which is assumed to be pointing at either a one-byte character or the first byte of a two-byte character. In either case it evaluates to the unsigned value of the character, and is analogous to "(*p)".

ADVANCE advances its pointer argument by the width of the character it is pointing at (either one or two bytes), and is analogous to "(p++)".

CHARADV combines the functions of *CHARAT* and *ADVANCE* in a single macro that evaluates to the unsigned value of a character and advances a pointer argument beyond the last byte of the character. It is analogous to "(*p++)".

WCHAR writes one (0<=c<=255) or two (256<=c<=65535) bytes of its integer argument, more significant byte first, at the location specified by "p". It is analogous to "(*p = c)" and evaluates to unsigned "c".

WCHARADV writes one (0<=c<=255) or two (256<=c<=65535) bytes of its integer argument, more significant byte first, at the location specified by "p", and advances "p" past the last byte. It is analogous to "(*p++ = c)" and evaluates to unsigned "c".

PCHAR places one (0<=c<=255) or two (0<=c<=65535) bytes of its integer argument, more significant byte first, at the byte location specified by the pointer argument. It is analogous to "{*p = c;}" and does not evaluate to "c". *PCHAR* is obsolete; use *WCHAR* instead.

PCHARADV places one (0<=c<=255) or two (256<=c<=65535) bytes of its integer argument, more significant byte first, at the byte location specified by the pointer argument, and advances the pointer past the last byte. It is analogous to "{*p++ = c;}" and does not evaluate to "c". *PCHARADV* is obsolete; use *WCHARADV* instead.

The functions *firstof2()*, *secof2()*, and *byte_status()*, are subroutine versions of the corresponding macros, and can be called from languages other than C.

WARNINGS

For maximum portibility, the use of the routines specified in *multibyte*(3C) is recommended for multibyte character processing.

Other *nl_tools_16*(3C) macros cannot be used as the first argument to *WCHAR* or *WCHARADV*. For example, *t++ = *f++ cannot be replaced by *WCHARADV(CHARADV*(f),t). Use instead, something such as int c; ... c = *CHARADV*(f), *WCHARADV*(c,t).

WCHAR and *WCHARADV* may produce a "null effect" warning from *lint*(1) if not used as part of another expression or as part of a statement. This will not affect the functionality of either macro.

Note that *WCHAR*, *WCHARADV*, *PCHAR* and *PCHARADV* are not "replace_char" macros. They do not prevent the second byte of a two-byte character from being left dangling if *WCHAR*, *WCHARADV*, *PCHAR* or *PCHARADV* overwrite the first byte of the two-byte character with a single-byte character.

CHARAT, *ADVANCE*, and *CHARADV* examine the byte following the location pointed to by the argument to verify its validity as a *SECof2* byte. If it is not a *SECof2* byte, the preceding byte will always be treated as a single-byte character.

EXTERNAL INFLUENCES

Locale

The LC_CTYPE category determines the interpretation of single and/or multi-byte characters.

AUTHOR

Nl_tools_16 was developed by HP.

SEE ALSO

setlocale(3C), hpnls(5).

NAME

nlappend – append the appropriate language identification to a valid MPE file name

SYNOPSIS

void nlappend(filename, langid, err)
char *filename;
short langid;
unsigned short err[2];

DESCRIPTION

This routine replaces the first three blanks found in *filename* with the language number. The purpose of *nlappend* is to identify the language of a file in an operating system-independent manner.

The arguments to *nlappend* are used as follows:

filename A string of up to eight ASCII characters terminated by three blanks.

langid A short integer specifying the language ID.

err The first element contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
4	*Filename* is not terminated by 3 blanks.

WARNINGS

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR

Nlappend was developed by HP.

SEE ALSO

portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlcollate – compare two character strings according to the MPE language-dependent collating sequence

SYNOPSIS

void nlcollate(string1, string2, length, result, langid, err, collseq)
char *string1, *string2, *collseq;
short length, *result, langid;
unsigned short err[2];

DESCRIPTION

Nlcollate collates two character strings according to the collating sequence of the specified language. This routine's purpose is to determine a lexical ordering. It is not intended to be used for searching or matching.

If the *collseq* parameter points to the null address, and *langid* is specified as (or defaults to) a language in which binary collation is appropriate, the binary collation is used to compare the two indicated strings. Otherwise, the *collseq* array will be used to determine the string compare operation (note that this may be a binary collation).

The arguments to *nlcollate* are used as follows:

string1 One of the character strings to be collated.

string2 The second character string to be collated.

length The length of the string segments to be collated.

result The result of the character collation is stored in the short integer variable to which *result* points.

0	If *string1* collates equal to *string2*.
−1	If *string1* collates before *string2*.
1	If *string1* collates after *string2*.

langid The language ID indicating the collating sequence to be used for the collation.

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid collating table entry.
4	Invalid length parameter.

collseq An array containing the collating sequence to be used, as returned from a call to *nlinfo*(3X)'s *itemnumber* 11.

WARNINGS

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR

Nlcollate was developed by HP.

SEE ALSO

nlinfo(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlconvclock – check and converts a time string to the MPE internal format

SYNOPSIS

unsigned int nlconvclock(instr, leninstr, langid, err)
char *instr;
short leninstr, langid;
unsigned short err[2];

DESCRIPTION

Nlconvclock converts *instr* to a general time format as returned by *nlinfo*(3X) *itemnumber* 3. This routine is the inverse of *nlfmtclock*(3X). Note that the seconds and tenths of seconds are always set to zero.

The arguments to *nlconvclock* are used as follows:

instr A character buffer containing the time to be converted.

leninstr An unsigned short specifying the length of the buffer.

langid A short containing the language ID.

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid time format.
4	Invalid length.

RETURN VALUE

Nlconvclock returns the time in the format:

```
Bits  0              7   8            15
     ----------------------------------
     |               |                |
     | Hour of Day   | Minute of Hour |
     |_______________|________________|

Bits  16          23   24            31
     ----------------------------------
     |             |                  |
     | Seconds     | Tenths of Seconds|
     |_____________|__________________|
```

WARNINGS

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR

Nlconvclock was developed by HP.

SEE ALSO

clock(3X), nlfmtclock(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlconvcustda – convert a date string to the MPE packed date format

SYNOPSIS

unsigned short nlconvcustdate(instr, leninstr, langid, err)
char *instr;
short leninstr, langid;
unsigned short err[2];

DESCRIPTION

Nlconvcustda converts *instr* to a packed date format. This routine is the inverse of *nlfmtcustdate*(3X).

The arguments to *nlconvcustda* are used as follows:

instr A character buffer containing the date to be converted.

leninstr A positive integer specifying the length of the string (in bytes).

langid A short containing the language ID number.

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid date format.
4	Invalid string length.

RETURN VALUE

The routine returns the date as an unsigned integer in the format:

```
Bits  0                  6   7        15
     ---------------------------------
     | Year of Century    | Day of Year |
     |____________________|_____________|
```

WARNINGS

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR

Nlconvcustda was developed by HP.

SEE ALSO

calendar(3X), nlfmtcustdate(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlconvnum – convert an MPE native language formatted number to an ASCII number

SYNOPSIS

void nlconvnum(langid, instr, leninstr, outstr, plenoutstr, err, numspec, fmtmask, pdecimals)
unsigned short err[2];
short langid, leninstr, *plenoutstr, fmtmask, *pdecimals;
char *instr, *outstr, *numspec;

DESCRIPTION

Nlconvnum converts a native language formatted number to an ASCII number, with an n-computer decimal separator (.) and thousands separator (,), to use for further conversion to INTEGER, REAL, etc.

This routine converts the decimal separator and the thousands separators to the n-computer equivalent, or strips them, according to the value of *fmtmask*. If *fmtmask* **and** M_NUMBERSONLY is not zero , *instr* is validated as a number. If it is null, no validation will take place.

For languages using an alternate set of digits (currently only **arabic**, which uses HINDI digits), *nlconvnum* also converts these digits to ASCII digits so they can be recognized and used as numeric characters.

The arguments to *nlconvnum* are used as follows:

langid A language ID number.

instr A character buffer containing the native language formatted number to convert. Leading and trailing spaces are ignored.

leninstr Length, in bytes, of *instr*.

outstr Output buffer; an array containing the converted output. The output is left-justified in the buffer, and *plenoutstr* contains the actual length of the converted number. *Outstring* may refer to the same address as *instr*.

plenoutstr A pointer to the length, in bytes, of *outstr*. After a successful call to *nlconvnum*, the short integer to which *plenoutstr* points contains the actual length of the converted number.

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid length specified (*leninstr* or *plenoutstr*).
4	Invalid number specified (*instr*).
7	Truncation has occurred (*outstr* is left partially formatted).
8	Invalid *numspec* parameter.
9	Invalid *fmtmask* parameter.

numspec A character buffer, as returned from *nlnumspec*, containing information about correct formatting. If this parameter is not null, *langid* is ignored and performance is improved (see the description of *nlnumspec*).

fmtmask An unsigned short specifying how to format the number. The default value is zero, which means substitution only, convert thousands separators, convert decimal separators, and that *instr* can contain any character.

Value	Description
M_STRIPTHOU	– Strip thousands separators.
M_STRIPDEC	– Strip decimal separators.
M_NUMBERSONLY	– *instr* contains a number.

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR

Nlconvnum was developed by HP.

SEE ALSO

nlfmtnum(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlfindstr – search for a string in another string using the MPE character set definition

SYNOPSIS

short nlfindstr(langid, string1, length1, string2, length2, err, charset)
short langid, length1, length2;
char *string1, *string2, *charset;
unsigned short err[2];

DESCRIPTION

Nlfindstr searches for the first occurrence of a given string of characters in another character string.

The arguments to *nlfindstr* are:

langid	The ID number of the desired language.
string1	A pointer to the character buffer to be searched. It can contain one-byte and two-byte characters.
length1	Length (in bytes) of *string1*.
string2	The character buffer for which to search.
length2	Length (in bytes) of *string2*. The *length2* must be less than or equal to *length1*.
err	The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid *length1* parameter.
4	Invalid *length2* parameter.

charset	A byte buffer containing the character set definition for the language to be used, as returned by *nlinfo*(3X)'s *itemnumber* 12.

RETURN VALUE

Offset is a short integer that holds the number of bytes into *string1* where *string2* was found. A **–1** is returned if the string is not found.

WARNINGS

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR

Nlfindstr was developed by HP.

SEE ALSO

nlinfo(3X), mpnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlfmtcalendar – format an MPE packed date using a localized format

SYNOPSIS

void nlfmtcalendar(date, outstr, langid, err)
unsigned short date, err[2];
char *outstr;
short langid;

DESCRIPTION

Nlfmtcal formats the specified date in the localized custom version of the date format, but with no time information (see *nlfmtclock*(3X)). For example:

FRI, OCT 2, 1987

The arguments to *nlfmtcal* are used as follows:

date An unsigned short indicating the date in the packed date format:

```
Bits  0                    6   7          15
     --------------------------------------
     |                      |              |
     | Year of Century      | Day of Year  |
     --------------------------------------
```

outstr A character buffer in which the formatted date is returned. This buffer is 18 bytes long, and padded with blanks if necessary.

langid A short integer specifying the language whose custom is to be used.

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid date format.

WARNINGS

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR

Nlfmtcal was developed by HP.

SEE ALSO

calendar(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlfmtclock – format an MPE time of day using a localized format

SYNOPSIS

void nlfmtclock(time, outstr, langid, err)
unsigned int time;
char *outstr;
short langid;
unsigned short err[2];

DESCRIPTION

Nlfmtclock formats the time of day obtained with the clock routine, according to the clock format defined for the specified language.

The arguments to *nlfmtclock* are used as follows:

time An unsigned int obtained from the clock routine:

```
Bits  0            7   8           15
      -------------------------------
      | Hour of Day    | Minute of Hour |
      -------------------------------

Bits  16         23   24          31
      -------------------------------
      | Seconds      | Tenths of Seconds |
      -------------------------------
```

outstr An 8-byte buffer in which the formatted time of day is returned.

langid A short integer specifying the language whose clock format is to be used.

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid time format.

WARNINGS

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR

Nlfmtclock was developed by HP.

SEE ALSO

clock(3X), nlconvclock(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlfmtcustdate – format an MPE packed date using a custom date

SYNOPSIS

void nlfmtcustdate(date, outstr, langid, err)
unsigned short date, err[2];
char *outstr;
short langid;

DESCRIPTION

Nlfmtcustdate converts the packed date format to the language-dependent custom date as specified in the language definition file. A custom date has an abbreviated format, such as "10/21/87" or "87.10.21".

The arguments to *nlfmtcustdate* are used as follows:

date An unsigned short containing the date in the packed date format:

Bits	0 6	7 15
	Year of Century	Day of Year

outstr A 13-byte buffer in which the formatted date is returned.

langid A short integer of the language whose custom date specification is to be used for the format.

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid date format.

WARNINGS

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR

Nlfmtcustdate was developed by HP.

SEE ALSO

calendar(3X), nlconvcustdate(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlfmtdate – format MPE date and time in a localized format

SYNOPSIS

void nlfmtdate(date, time, outstr, langid, err)
unsigned short date, err[2];
unsigned long time;
char *outstr;
short langid;

DESCRIPTION

Nlfmtdate formats the specified date and time in a localized custom version. For example:

SUN, FEB 7, 1988 9:00 AM

The arguments to *nlfmtdate* are used as follows:

date An unsigned short indicating the date to be formatted in the packed date format:

```
Bits  0                 6   7        15
     ----------------------------------
     | Year of Century    | Day of Year |
     ----------------------------------
```

time An unsigned int indicating the time to be formatted. The double word is in the clock format:

```
Bits  0              7   8            15
     -----------------------------------
     | Hour of Day     | Minute of Hour |
     -----------------------------------

Bits  16         23  24              31
     -----------------------------------
     | Seconds      | Tenths of Seconds |
     -----------------------------------
```

outstr A 28-byte buffer in which the formatted date is returned.

langid A short containing the language ID indicating the custom to be used.

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid date format.
4	Invalid time format.

WARNINGS

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR

Nlfmtdate was developed by HP.

SEE ALSO

calendar(3X), clock(3X), nlfmtcal(3X), nlfmtclock(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlfmtlongcal – format an MPE packed date using a long calendar format

SYNOPSIS

void nlfmtlongcal(date, outstr, langid, err)
unsigned short date, err[2];
char *outstr;
short langid;

DESCRIPTION

Nlfmtlongcal formats the supplied date according to the long calendar format. The formatting is done according to the template returned by *nlinfo*(3X), *itemnumber* 30.

The arguments to *nlfmtlongcal* are used as follows:

date A short integer value containing a date in the packed date format:

```
Bits  0                    6   7        15
     ------------------------------------
     |  Year of Century    |  Day of Year |
     |_____________________|______________|
```

outstr A 36-byte buffer to which the formatted long calendar date is returned, padded with blanks if necessary.

langid An ID number specifying which language-specific format is to be used.

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid date format.

WARNINGS

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR

Nlfmtlongcal was developed by HP.

SEE ALSO

calendar(3X), nlfmtcalendar(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlfmtnum – convert an ASCII number to an MPE language-specific formatted number

SYNOPSIS

void nlfmtnum(langid, instr, leninstr, outstr, plenoutstr, err, numspec, fmtmask, decimals)
short langid, leninstr, *plenoutstr, fmtmask, decimals;
unsigned short err[2];
char *instr, *outstr, *numspec;

DESCRIPTION

Nlfmtnum converts a string containing an ASCII number to a language-specific formatted number using the currency name/symbol, decimal separator and thousands separators defined for the language. The string may contain the n-computer decimal separator (.), thousands separator (,) and a dollar sign ($).

This routine operates in two modes, substitution mode and formatting mode. The substitution mode (if *fmtmask* is zero) substitutes the native equivalent for "." and "," and, for **arabic**, the alternate set of digits for ASCII digits. The input is not validated as a number, and can contain several individual numbers. No justification takes place, and the output is left-truncated if *outstr* is shorter than *instr* (for example, **1,234.56** becomes **234,56**).

If *fmtmask* is not zero, the formatting mode formats the input according to *fmtmask* in addition to performing the substitution. In this mode the input is validated as a number and only ASCII digits and "−", "+", "$", "." and "," are allowed. Only one sign and one "$" is allowed and must be the first character(s) in *instr*. Even if insertion (of thousands separators, etc.) is specified in *fmtmask*, thousands separators and a decimal separator are still valid characters in the input. In this case they are substituted. If no justification is specified, the output is right-justified with the same number of trailing spaces as the input. Note that for languages written right-to-left, trailing spaces in the input are preserved as leading spaces in the output. If the output is truncated, it is left-truncated (for example, **1,234.56** becomes **.234,56**).

The arguments to *nlfmtnum* are used as follows:

langid — A language ID number specifying which language's formatting specifications to use for the formatting.

instr — A byte array containing the n-computer formatted ASCII number to be converted, for example, 123,456.78. Leading and trailing spaces are allowed.

leninstr — Length, in bytes, of *instr*.

outstr — A byte buffer where the language-specific formatted number is returned. The decimal separator, thousands separator and currency symbol/name are replaced according to the language definition if present or inserted, or if specified by *fmtmask*. *Outstr* may reference the same address as *instr*.

plenoutstr — Length, in bytes, of *outstr*. After a successful call, if specified by *fmtmask* (the two bits starting with bit 12 (from highest to lowest) are equal to **3**), *plenoutstr* returns the actual length, in bytes, of the formatted number.

err — The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured,
3	Invalid length specified (*leninstr* or **plenoutstr*).
4	Invalid number specified (*instr*).
5	Invalid decimal point in number specified (*instr*).

6	Invalid thousand separators in number specified (*instr*).
7	Truncation has occurred (*outstr* is left partially formatted).
8	Invalid *numspec* parameter.
9	Invalid *fmtmask* parameter.
10	Invalid *decimals* parameter.

numspec A byte array, as returned from *nlnumspec*(3X), containing formatting specifications for the specified language (currency symbol/name, decimal separator, etc.). If this parameter is not null, *langid* is ignored, and performance is improved. (See the description of *nlnumspec*(3X)).

fmtmask A short integer value specifying any formatting to be done on the input. The default value is zero, which means a simple substitution.

Value	Description
NULL	Do not insert thousands separators. Do not insert decimal separator. No justification of the output.
M_INSTHOU	Insert thousands separators.
M_INSDEC	Insert decimal separator.
M_CURRENCY	Insert currency name/symbol.
M_LEFTJUST	The output is left-justified.
M_RIGHTJUST	The output is right-justified.
M_RETLENGTH	The output is left-justified and *plenoutstr* returns the actual length of the formatted number.

decimals An integer specifying where to insert the decimal separator. The value is ignored if *fmtmask* **and** M_INSDEC is zero, or a decimal separator is present in the number.

WARNINGS

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR

Nlfmtnum was developed by HP.

SEE ALSO

nlconvnum(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlgetlang – return the current user, data, or system default language

SYNOPSIS

short nlgetlang(function, err)
short function;
unsigned short err[2];

DESCRIPTION

Nlgetlang looks for a LANG string in the user's environment. If it finds it, it returns the corresponding integer listed in *langid (5).* Otherwise, or if the value of *function* is not valid, it returns *0* and sets the *err* parameter.

The arguments to *nlgetlang* are used as follows:

function A short integer that specifies which language is returned.

Value	Description
1	User language
2	Data language
3	System default language

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
1	Native Language Support file(s) not found
2	Specified language not configured
3	Invalid *function* value
4	No language specified for NLGETLANG to access

RETURN VALUE

Nlgetlang returns the language ID as a short integer. In case of error, zero is returned.

WARNINGS

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR

Nlgetlang was developed by HP.

SEE ALSO

getenv(3C), currlangid(3C), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlinfo – return MPE language-dependent information

SYNOPSIS

void nlinfo(itemnumber, itemvalue, langid, error)
short itemnumber;
int *itemvalue;
short *langid;
unsigned short error[2];

DESCRIPTION

Nlinfo returns such information as the format of the date, the radix character, the direction of the language, etc.

The *itemnumber* indicates the type of information the user has requested. The data is passed back in *itemvalue*.

The arguments to *nlinfo* are used as follows:

itemnumber A short integer of the item desired. This number specifies which item value is to be returned. See below for a list of item numbers.

itemvalue A pointer to an integer that contains the value of the item specified by the corresponding item number. The data type of the item value depends on the item itself.

langid A pointer to a short integer containing the language ID, or for *itemnumber* 22, the location in which the language ID is returned.

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
1	Native Language Support file(s) not found
2	Specified language is not configured.
3	Specified character set is not configured.
10	*Itemnumber* is out of range.

Item numbers

The following is a list of the currently defined item numbers and the information returned.

Itemnumber	Description
1	An 18-byte buffer in which the calendar format is returned.
2	A 13-byte buffer in which the custom date format is returned.
3	An 8-byte buffer in which the clock specification is returned.
4	A 48-byte buffer in which the month denotation abbreviation table is returned. The abbreviation of each month is 4 bytes long (with blank padding if necessary). The first 4 bytes are the abbreviation for January.
5	A 144-byte array in which the month denotation table is returned. Each month denotation is 12 bytes long. The table starts with January.
6	A 21-byte array in which the day of the week denotation abbreviation table is returned. Each weekday abbreviation is three bytes long. The first three bytes are the abbreviation for Sunday.
7	An 84-byte array in which the day of the week denotation table is returned. Each weekday denotation is 12 bytes long. The table starts with Sunday.

8	A 12-byte array in which the YES/NO responses are returned. The first 6 bytes contain the (upshifted) "YES" response; the second 6 bytes contain the (upshifted) "NO" response.
9	A 2-byte array in which the symbols for decimal point and thousands indicator are returned. The first byte contains the decimal point, the second contains the thousands indicator.
10	A 6-byte array in which the currency signs are returned. The first byte contains the currency sign used in the business formats, the second byte is either a numeric zero, which indicates that the currency symbol precedes the value, or a one, which indicates that a symbol follows the value. The next 4 bytes contain the fully qualified currency sign.
11	An array in which the collating sequence table is returned. To determine the size of this array, the length must be determined by a call to *nlinfo* with *itemnumber* 27.
12	A 256-byte array in which the character set definition is returned. Each byte has numeric identification of the character type: **0** numeric character **1** Alphabetic lowercase character **2** Alphabetic uppercase character **3** Undefined graphic character **4** Special character **5** Control code **6** First byte of a two-byte character
15	A 256-byte array in which the upshift table is returned.
16	A 256-byte array in which the downshift table is returned.
17	An array of unsigned shorts in which the language numbers of all configured languages are returned. The first element of this array contains the number of configured languages. The second word contains the language number of the first configured language, etc. The system default language is returned (the *langid* parameter, if specified, is insignificant).
18	A short int in which true (**−1**) is returned if the specified language is supported (configured) on the system. Otherwise, false (**0**) is returned.
21	A 16-byte array in which the (uppercase) name of the specified language is returned. If the name contains less than 16 bytes, it is padded with blanks.
22	The *itemvalue* contains a byte buffer containing a language name or language number (ASCII digits) terminated by a blank. The array must contain less than or equal to 16 bytes. The *langid* (third) parameter is assigned the associated language ID number.
26	A short integer in which the class number of the specified language is returned.
27	An integer in which the length (in two-byte units) of the collating sequence table corresponding to the specified language is returned.
28	A short integer in which the length (in two-byte units) of the national dependent information table is returned. If no national table exists for the specified language, an error is returned.
29	A byte buffer in which the national dependent information table is returned. To determine the size of this array, the length must be obtained via a prior call to *nlinfo* with *itemnumber* 28.

30 A 36-byte array in which the long calendar format is returned. It may contain arbitrary text, as well as the following descriptors:

D	1 through 3 of these are to be replaced by that many bytes from the day abbreviation.
W	1 through 12 of these are to be replaced by that many bytes from the day of the week.
M	1 through 4 of these are to be replaced by that many bytes from the month abbreviation.
O	1 through 12 of these are to be replaced by that many bytes from the month of the year.
mm	Numeric month of the year.
yy	Numeric year of the century.
yyyy	Numeric year of the century.
Nyy	National year.

In addition, a special literal character "~" (tilde) can be used to indicate that the following character should be taken literally in the format, even if it is one of the special characters above.

For example, a format could be:

"WWWWWWWWW, OOOOOOOOO dd, A.~D. yyyy "

This format in n-computer would result in the following:

"WEDNESDAY, NOVEMBER 21, A.D. 1984 "

31 A 16-byte array in which the currency name is returned.

32 An 8-byte array, containing information about an Alternate set of digits. (Currently only used by **arabic**).

Byte	Description
0–1	Alternate digit indicator 0 – No Alternate digits defined 1 – Alternate digits defined
2	The Alternate digit "0"
3	The Alternate digit "9"
4	The "+" used with Alternate digits
5	The "–" used with Alternate digits
6	The decimal separator used with Alternate digits
7	The thousands separator used with Alternate digits

33 A 4-byte array, containing information about the direction of the language.

Byte	Description
0–1	Language direction 0 – Direction is "left-to-right" 1 – Direction is "right-to-left"
2	The "right-to-left" space
3	Undefined

34 An unsigned short that returns the data ordering of the language.

0	Keyboard order
1	Left-to-Right screen order

	2	Right-to-Left screen order
35	An unsigned short that returns the size of the character used by the language.	
	0	One-byte characters (8 bits)
	1	Two-byte characters (16 bits)

WARNINGS

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR

Nlinfo was developed by HP.

SEE ALSO

hpnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlist – get entries from name list

SYNOPSIS

#include <nlist.h>

int nlist (file-name, nl)
char *file-name;
struct nlist *nl;

REMARKS

The use of symbol table type and value information is inherently non-portable. Use of *nlist* should reduce the effort required to port a program which uses such information, but complete portability across all implementations of HP–UX cannot be expected.

DESCRIPTION

Nlist examines the name list in the executable file whose name is pointed to by *file-name*, and selectively extracts a list of values and puts them in the array of *nlist* structures pointed to by *nl*. The array of *nlist* structures initially contains only the names of variables. Once *nlist* has been called, the variable names are augmented with types and values. The list is terminated by a null name, which consists of a null string in the variable name position of the structure. The name list of the file is searched for each variable name. If the name is found, type and value information from the file is inserted into the name list structure. If the name is not found, type and value fields are set to zero. The structure **nlist** is defined in the include file **<nlist.h>**. See *a.out*(4) and *nlist*(4) for further description of the symbol table structure.

The file must have the organization and symbol table described for an a.out file in *a.out*(4). The information is extracted from the symbol table used by the loader, *ld*(1).

On machines which have such a file, this subroutine is useful for examining the system name list kept in the file **/hp-ux**. In this way programs can obtain system addresses that are up to date.

RETURNS

All nlist structure fields are set to 0 if the file cannot be found or if it is not a valid object file containing a linker symbol table.

Nlist returns −1 upon error; otherwise it returns 0.

NOTES

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

SEE ALSO

a.out(4), nlist(4).

STANDARDS CONFORMANCE

nlist: SVID2

NAME

nljudge – judge whether a character is a one-byte or multi-byte Asian character using the MPE character definition table

SYNOPSIS

short nljudge(langid, instr, length, judgeflag, err, charset)
short langid, length;
char *instr, *judgeflag, *charset;
unsigned short err[];

DESCRIPTION

Nljudge judges whether or not a character is a one-byte or multi-byte Asian character. If it is a multi-byte character, *judgeflag* is set to **1** or **2**. If it is a one-byte character, *judgeflag* is set to **0**.

Any language number can be specified as the *langid* parameter. However, if the language specified uses only one-byte characters (see *nlinfo*(3X)'s *itemnumber* 35), the *judgeflag* returns all zeroes.

The arguments to *nljudge* are used as follows:

langid The ID number for the desired language.

instr The character buffer to be judged.

length A short integer value specifying the number of bytes in *instr*.

judgeflag A pointer to a char whose value is set to:

0	One-byte character
1	First byte of a two-byte character
2	Second byte of a two-byte character
3	Invalid two-byte character

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid string length.
7	Invalid characters found in *instr*.

charset A character buffer containing the character set definition for the language to be used, as returned by *nlinfo*(3X)'s *itemnumber* 12. If it doesn't point to a null address, the *langid* parameter is ignored, and this routine is more efficient.

RETURN VALUE

Nljudge returns the number of multi-byte Asian characters that could be used to check if a string of character contains any Asian characters.

WARNINGS

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR

Nljudge was developed by HP.

SEE ALSO

nlinfo(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlkeycompare – determine if a character array (key1) is almost equal to another (key2) using the MPE language-dependent collation table

SYNOPSIS

void nlkeycompare(key1, length1, key2, length2, presult, langid, err, collseq)
char *key1, *key2;
short length1, length2, langid, *presult;
unsigned short err[2], collseq[];

DESCRIPTION

Nlkeycompare determines if a character array (key1) is almost equal to another character array (key2). Two character arrays are considered almost equal when they differ only in case or accent priorities. For example, the arrays **ABC** and **aBc** are almost equal in English.

Nlkeycompare determines if a given character array can be collated before or after another character array of a different length. For example, *nlkeycompare* examines the records in a file sorted in a given language and determines if the character array **key1** can be found later on in the file as the leading substring of the sort key, if the value of the last record read is **key2.**

The arguments to *nlkeycompare* are used as follows:

key1 A byte array being compared to the key2.

length1 The length in bytes of *key1*. *Length1* must be less than *length2*.

key2 A byte array containing a character array to which to compare *key1*.

length2 The length in bytes of *key2*. *Length2* must be greater than *length1*.

presult A pointer to a short integer variable in which to return the result of the comparison.

- **0** The retrieved key2 matches the key1.
- **1** The retrieved key2 does not match the key1. It is different only in case or accent priority.
- **2** The retrieved key2 is less than the key1 (its collating order is before the desired one).
- **3** The retrieved key2 is greater than the key1 (it collates after the desired key).

langid The language ID number indicating the collating sequence to be used for the compare.

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid collating table entry.
4	Invalid length parameter.
7	*Length1* is greater than *length2*.

collseq An array containing the collating sequence table as returned by *nlinfo*(3X)'s *itemnumber* 11.

WARNINGS

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR

Nlkeycompare was developed by HP.

SEE ALSO

nlcollate(3X), nlinfo(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlnumspec – return information needed by MPE routines for formatting and converting numbers

SYNOPSIS

void nlnumspec(langid, numspec, err)
short langid;
char *numspec;
unsigned short err[2];

DESCRIPTION

Nlnumspec returns the information needed for formatting and converting numbers. It combines several calls to *nlinfo*(3X) in order to simplify the use of native language formatting. By calling *nlnumspec* once, and passing the obtained information to *nlfmtnum*(3X) and *nlconvnum*(3X), implicit calls to *nlnumspec*(3X) from *nlfmtnum*(3X) and *nlconvnum*(3X) are avoided and performance is improved.

Nlnumspec combines the functions of *itemnumber* 9, 10, 31, 32, and 33 on *nlinfo*(3X). The information is formatted where needed. For example, any spaces in the currency symbol/name are included. The currency symbol/name is the shortest non-blank descriptor, as returned from *nlinfo*(3X) *itemnumber* 10 and 31.

Nlnumspec does not, apart from the mentioned formatting, provide any information not obtainable with *nlinfo*(3X), but is included for the convenience of the user. For efficiency, the user of this routine calls it once, saves the result, and then calls *nlfmtnum*(3X) and/or *nlconvnum*(3X) multiple times.

The arguments to *nlnumspec* are used as follows:

langid The ID number of the desired language.

numspec A character buffer of at least 60 bytes in which the following information is returned:

Byte	Description
00–01	Language ID number.
02–03	Alternate Digit Indicator. 0 – No Alternate digits exist. 1 – Alternate digits exist.
04–05	Language Direction Indicator. 0 – The Language is "left-to-right." 1 – The Language is "right-to-left."
06–07	The Alternate digit range ("0", "9").
08	Decimal separator (ASCII-digits).
09	Decimal separator (Alternate-digits).
10	Thousands separator (ASCII-digits).
11	Thousands separator (Alternate-digits).
12	"+" Alternate-digits.
13	"–" Alternate-digits.
14	"Right-to-left" space.
15	Reserved.
16-17	Currency place. 0 – Currency symbol precedes the number. 1 – Currency symbol follows the number. 2 – Currency symbol replaces the decimal separator.
18-19	Length of Currency symbol (including any spaces).

20-37	Currency symbol (including any spaces).
38-39	Data ordering of the language.
40-41	Size of character used by the language.
42-59	Reserved.

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.

WARNINGS

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR

Nlnumspec was developed by HP.

SEE ALSO

nlinfo(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlrepchar – replace non-displayable characters of a string using the MPE character set table

SYNOPSIS

void nlrepchar(instr, outstr, length, repchar, langid, err, charset)
char *instr, *outstr, repchar, *charset;
short length, langid;
unsigned short err[2];

DESCRIPTION

Nlrepchar replaces all non-displayable characters in the input character buffer with the replacement character. Non-displayable characters are those of types 3 and 5, as returned by *nlinfo*(3X), *itemnumber* 12. Native language characters of the supported character set are not replaced.

The arguments to *nlrepchar* are used as follows:

instr A character buffer in which the non-displayable characters must be replaced.

outstr A character buffer to which the replaced character string is returned.

length A short integer specifying the length (in bytes) of *instr*.

repchar A byte specifying the replacement character to be used.

langid A short integer value specifying the language ID number of the language that determines the character set to be used.

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid replacement character.
4	Invalid length parameter.
8	The value of *outstr* would overwrite *instr*.
10	Invalid Asian character.

charset Contains the character set definition for the language to be used, as returned in *nlinfo*(3X)'s *itemnumber* 12. If this parameter is supplied (i.e., not NULL), *langid* is ignored and this routine is much more efficient.

AUTHOR

Nlrepchar was developed by HP.

SEE ALSO

nlinfo(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlscanmove – move, scan and case shift character strings using the MPE character set definition table

SYNOPSIS

short int nlscanmove(instr, outstr, flags, length, langid, err, pcharset, pshift)
char *instr, *outstr;
short flags;
int length;
short langid;
unsigned short err[2];
char *pcharset, *pshift;

DESCRIPTION

Nlscanmove moves, scans and/or case shifts character strings.

The arguments to *nlscanmove* are used as follows:

instr A character buffer that acts as the source string of the scan or move functions.

outstr A character buffer that acts as the target. Note that if *outstr* is equal to *instr*, this routine will act as scan. Otherwise, a move will be performed, see *err* below.

flags A flag defining the options for the routine invocation. This parameter defines the end condition for the scan or move.

Value	Description
M_L	Select lowercase alphabetic characters.
M_U	Select uppercase alphabetic characters.
M_N	Select numeric characters.
M_S	Select special characters.
M_WU	By default *nlscanmove* will scan or move characters while the character currently being scanned is one of those selected (i.e. upper, lower, numeric, special). If **M_WU** is used, then *nlscanmove* will scan or move characters until the character currently being scanned is one of those selected.
M_US	Shift scanned or moved characters to the uppercase.
M_DS	Shift scanned or moved characters to the lowercase.
M_OB	Select one-byte characters.
M_TB	Select two-byte (Asian) characters.
M_OB or M_TB	Select both one- and two-byte characters.

length A short integer indicating the maximum number of valid bytes to be acted upon during the indicated option.

langid A short integer containing the language ID number which implies the both the character set definitions of character attributes and the language specific shift.

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error # Meaning

2	Specified language is not configured.
3	Overlapping strings, *instr* overwrites *outstr*.
4	Invalid length parameter.
7	The reserved part of *flags* is not zero.
8	Both upshift and downshift request.
9	Invalid table element.
10	Invalid Asian character.

pcharset A pointer to a character buffer containing the character set definition for the language to be used, as returned *nlinfo*(3X)'s *itemnumber* 12. If not zero, the *langid* parameter is ignored, and this routine is much more efficient. This parameter is required for calls in which bits (12:4) of *flags* is neither **0** nor **15**.

pshift A pointer to a character buffer containing shift information for a desired upshift or downshift (e.g., as returned in *nlinfo*(3X)'s *itemnumber* 15 or 16). This parameter is used when bits (9:2) of *flags* is not **0**.

RETURN VALUE

A short containing the number of bytes acted upon in the scan or move operation.

WARNINGS

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR

Nlscanmove was developed by HP.

SEE ALSO

nlinfo(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlsubstr − extract a substring of a string using the MPE character set definition table

SYNOPSIS

void nlsubstr(instring, inlength, outstring, poutlength, start, movelength, langid, flags, err, charset)
char *instring, *outstring,;
short inlength, *poutlength, start, movelength, langid;
short flags;
unsigned short err[], charset[];

DESCRIPTION

Nlsubstr extracts a substring from *instring* and places the result in *outstring*.

The arguments to *nlsubstr* are used as follows:

instring — The byte buffer from which the substring is extracted. The string can contain both one-byte and two-byte (Asian) characters.

inlength — Length, in bytes, of *instring*

outstring — Where the sub-string is placed.

poutlength — Length, in bytes, of *outstring*. After a successful call, the variable to which *poutlength* points will contain the actual length of the sub-string moved to *outstring*.

start — The offset into *instring* where the sub-string starts. A value of zero is the beginning point.

movelength — Length, in bytes, of the sub-string.

langid — The ID number of the desired language.

flags — This flag word is used primarily with Asian languages. It is meaningless with one-byte oriented languages. *Flags* is used to indicate the treatment of the case when the first byte of the sub-string is the second byte of a two-byte Asian character and in the case where the last byte in the sub-string is the first byte of a two-byte Asian character.
Selection of *nlsubstr*'s behavior if the first character is the second byte of an Asian character:

Value	Description
F_RETURNERR	Return an error condition.
F_SPP1	Start from *start*+1.
F_SPM1	Start from *start*−1.
F_SPBL	Start from *start*, but replace the character with a blank in *outstring*.
F_SP	Start from *start*, regardless of the value of the first character.

Selection of *nlsubstr*'s behavior if the last character is the first byte of an Asian character:

Value	Description
F_LMP1	Move until *movelength*+1 is reached.
F_LMM1	Move until *movelength*−1 is reached.

F_LMBL Move until *movelength* is reached, but replace the character with a blank in *outstring*.

F_LM Move until *movelength* is reached, regardless of the value of the last byte.

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
7	Invalid *inlength*.
8	Invalid *start*.
9	Invalid *movelength*.
11	Invalid value in *flags* bits (8:4).
12	Invalid value for *flags* bits (12:4).
13	The start position is the second byte of an Asian character, or an underflow condition occurred because of *flags*.
14	The end position is the first byte of an Asian character, or an overflow condition occurred because of *flags*.

charset An array containing the character set definition for the language to be used, as returned *nlinfo*(3X)'s *itemnumber* 12.

WARNINGS

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR

Nlsubstr was developed by HP.

SEE ALSO

nlinfo(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlswitchbuf – convert a string of characters between phonetic order and screen order using the MPE character set definition table

SYNOPSIS

void nlswitchbuf(langid, instr, outstr, length, lefttoright, err)
char *instr, *outstr;
short length, langid;
unsigned short lefttoright, err[2];

DESCRIPTION

Nlswitchbuf is useful for handling data from languages written from right-to-left (e.g., Middle Eastern languages). It is used by a program to convert a buffer that is in phonetic order (i.e., the order in which the characters would be typed at a terminal or spoken by a person) to screen order (i.e., the order in which the characters are displayed on a terminal screen or piece of paper), or vice-versa. Screen order is defined as right-to-left if the primary mode of the terminal or printer is from right-to-left (as when it is used principally for entering or displaying data from a right-to-left language). Otherwise, screen order is defined as left-to-right.

Phonetic order and screen order are, in general, not the same if USASCII text is mixed with that from a right-to-left language. The relationship between phonetic order and screen order is further complicated by the Hindi digits in Arabic, which play a third role intermediate between ASCII characters and characters of the right-to-left language.

Note that this is a somewhat special purpose native language support routine. *Nlswitchbuf* is useful only for languages that are written from right-to-left, and which may occasionally mix left-to-right text (e.g., English) with right-to-left. Nonetheless, it can be used by a general-purpose (not specifically for handling right-to-left data) program. Such a program calls *nlswitchbuf* to convert data from phonetic order to screen order and back again. (For example, an editor that wants to track cursor movement on a terminal against a buffer of text in memory needs to do this.) If the data is not that of a right-to-left language, this routine simply returns the same text unchanged, since for all other languages phonetic order and screen order are the same.

langid The ID number for the desired language.

instr The character buffer in phonetic order to be converted to screen order.

outstr The buffer in which the result of the conversion to screen order is returned. *Outstr* and *instr* can reference the same address.

length The length, in characters, of the buffer to be converted.

lefttoright An unsigned short integer that specifies whether the implied primary mode of the data (i.e., the way it would be displayed on a terminal) is left-to-right (TRUE) or right-to-left (FALSE). This determines what the opposite language is and, therefore, strings of which characters get switched.

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid string length.

WARNINGS

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR

Nlswitchbuf was developed by HP.

SEE ALSO

nlinfo(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nltranslate – translate ASCII strings to EBCDIC using the MPE conversion table

SYNOPSIS

void nltranslate(code, instr, outstr, length, langid, err, table)
short code, length, langid;
char *instr, *outstr, *table;
unsigned short err[2];

DESCRIPTION

Nltranslate translates a string of bytes from EBCDIC to ASCII or ASCII to EBCDIC, using the appropriate native language table.

The arguments to *nltranslate* are used as follows:

code — **1** - Specifies EBCDIC to ASCII conversion.
2 - Specifies ASCII to EBCDIC conversion.

instr — The byte buffer to be translated.

outstr — A byte buffer to which is returned the translated string. The parameters *instr* and *outstr* can specify the same array.

length — A short integer specifying the number of bytes of *instr* to be translated.

langid — A short integer containing the ID number of the language whose translation tables are to be used.

err — The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid code specified.
4	Invalid length parameter.

table — A 256-byte array that holds a translation table. Each byte contains the translation of the byte whose value is its index. This table is provided by the user.

WARNINGS

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR

Nltranslate was developed by HP.

SEE ALSO

nlinfo(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

open_jlib, close_jlib – enable or disable Japanese specific facilities

SYNOPSIS

#include <jlib.h>

int open_jlib (langname)
char *langname;

int close_jlib ()

DESCRIPTION

The arguments to *open_jlib* are *langname,* which is used to bind operation to the end-user's specified language requirements. For example,

open_jlib (getenv ("LANG"));

Once *open_jlib* is invoked, the following facilities are available. Those marked with an asterisk are provided by a server process. Note that once *open_jlib* is invoked, another one must not be invoked until *close_jlib* is invoked.

RomajiHiragana (s1, s2)*	ROMAJI to HIRAGANA
RomajiKatakana (s1, s2)*	ROMAJI to KATAKANA
RomajiHankakuKatakana (s1, s2)*	ROMAJI to HANKAKU KATAKANA
HiraganaKatakana (s1, s2)	HIRAGANA to KATAKANA
KatakanaHiragana (s1, s2)	KATAKANA to HIRAGANA
HankakuZenkaku (s1, s2, mode)	HANKAKU to ZENKAKU
ZenkakuHankaku (s1, s2)	ZENKAKU to HANKAKU
KutenZenkaku (c, s)	KUTEN (section-point) code to ZENKAKU
J_UD_open (filename, mode)*	open a user dictionary
J_UD_close (dp)*	close a user dictionary
J_UD_store (key, kouho, dp)*	store a word into a user dictionary
J_UD_delete (key, kouho, dp)*	delete a word from a user dictionary
J_UD_search (key, dp)*	search a word in a user dictionary
J_UD_free (p)*	free a space allocated by *J_UD_search*
open_kana_kan (filename)*	initialize KANA to KANJI conversion
close_kana_kan (ed)*	terminate KANA to KANJI conversion
Henkan (ed, string, len, buf, size, mode)*	perform KANA to KANJI conversion
JiKouho (ed, pb, nb)*	get all KOUHOs
Kakutei (ed, pb, nb, nk)*	update HINDO information
HenkanOwari (ed, pb)*	free a space allocated by *Henkan*
SetUserDict (ed, dp, mode)*	enable or disable to consult a user dictionary

When these facilities are no longer needed, invoke *close_jlib* to close them.

DIAGNOSTICS

Open_jlib returns 0 upon successful completion. Otherwise, −1 is returned and **jlib_errno** is set to indicate the error:

[JUNAVAIL] Cannot connect to server. Above facilities marked with an asterisk are not available.

[JUNAVAIL] *Open_jlib* has been invoked.

Close_jlib returns 0 upon successful completion. Otherwise, −1 is returned.

GLOSSARY

Here is a glossary of terms used in the description of each manpage entry for Japanese-specific facilities shown above.

BUNSETSU	a small group of words
HINDO	the frequency of use
ROMAJI	a way of spelling Japanese by Roman character
KANA	a character to express a syllable developed in Japan based on KANJI. There are two kinds of KANA, HIRAGANA and KATAKANA.
HIRAGANA	characters from 04-01 to 04-83 in section-point code
KATAKANA	characters from 05-01 to 05-86 in section-point code
KUTEN	code a one of expression for KANJI characters
ZENKAKU	character a character two times as large as a HANKAKU character
HANKAKU	character a character in the KANA8 character set
YOMI	show how to pronounce a KANJI
DAKUON	sound to express KANA that is written preceding DAKUTEN
DAKUTEN	a symbol to express DAKUON
HANDAKUON	sound to express KANA that is written preceding HANDAKUTEN; i.e., sound of PA, PI, PU, PE, and PO.
HANDAKUTEN	a symbol to express HANDAKUON
HYOUKI	show how to spell a Japanese word
HINSHI	a part of speech

SEE ALSO

RomajiHiragana(3X), RomajiKatakana(3X), RomajiHankakuKatakana(3X), HiraganaKatakana(3X), KatakanaHiragana(3X), HankakuZenkaku(3X), ZenkakuHankaku(3X), KutenZenkaku(3X), J_UD_open(3X), J_UD_close(3X), J_UD_store(3X), J_UD_delete(3X), J_UD_search(3X), J_UD_free(3X), open_kana_kan(3X), close_kana_kan(3X), Henkan(3X), JiKouho(3X), Kakutei(3X), HenkanOwari(3X), SetUserDict(3X)

NAME

open_kana_kan, close_kana_kan – initialize KANA to KANJI conversion

SYNOPSIS

#include <jlib.h>

int open_kana_kan (filename)
char *filename;

int close_kana_kan (ed)
int ed;

DESCRIPTION

Open_kana_kan initializes and sets up the environment for KANA to KANJI conversion. The file named *filename* is used to update and store HINDO information. If the file does not exist, it is created. If a NULL pointer is specified, it is disabled to update and store HINDO information.

Open_kana_kan returns an environment descriptor which is used in calling the following function:

Henkan (ed, string, len, buf, size, mode)	perform KANA to KANJI conversion
JiKouho (ed, pb, nb)	get all KOUHOs
Kakutei (ed, pb, nb, nk)	update HINDO information
HenkanOwari (ed, pb)	free a space allocated by *Henkan*
SetUserDict (ed, dp, mode)	enable or disable to consult a user dictionary

Close_kana_kan closes the environment descriptor indicated by *ed*, which is obtained from an *open_kana_kan* call.

DIAGNOSTICS

Open_kana_kan returns an environment descriptor upon successful completion. Otherwise, -1 is returned and **jlib_errno** is set to indicate the error:

[JSDACCES]	The system dictionary exists but permission is denied.
[JSDWRONG]	The system dictionary has an incorrect format
[JSDNOENT]	The system dictionary does not exist.
[JSDBADENT]	The file having the same path name as the system dictionary exists.
[JHTACCES]	The file named *filename* exists but permission is denied.
[JHTWRONG]	The format of the file named *filename* is wrong.
[JHTBADENT]	The named file exists but it is not the file to update and store HINDO information.
[JMENV]	The maximum allowed number of environment descriptors are already open.

Close_kana_kan returns 0 upon successful completion. Otherwise, -1 is returned and **jlib_errno** is set to indicate the error.

[JBADED]	*Ed* is not a valid environment descriptor.

WARNINGS

The maximum number of environment descriptors allowed is 1.

NAME

perror, strerror, errno, sys_errlist, sys_nerr – system error messages

SYNOPSIS

#include <string.h>

extern int errno;

extern char *sys_errlist[ ];

extern int sys_nerr;

void perror (s)
const char *s;

char *strerror (errnum)
int errnum;

DESCRIPTION

Perror writes a language-dependent message to the standard error output, describing the last error encountered during a call to a system or library function. The argument string *s* is printed first, followed by a colon, a blank, the message, and a new-line. To be most useful, 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. The contents of the message is identical to those returned by the *strerror* function with **errno** as the argument. If given a NULL string, the *perror* function prints only the message and a new-line.

To simplify variant formatting of messages, the *strerror* function and the *sys_errlist* array of message strings are provided. The *strerror* function maps the error number in *errnum* to a language-dependent error message string and returns a pointer to the string. The message string is returned without a new-line. *Errno* can be used as an index into *sys_errlist* to get an untranslated message string without the new-line. *Sys_nerr* is the largest message number provided for in the table; it should be checked because new error codes might be added to the system before they are added to the table. The *strerror* function must be used to retrieve messages when translations are desired.

EXTERNAL INFLUENCES

Environment Variables

The language of the message returned by *strerror* and printed by *perror* is specified by the LANG environment variable. If the language-dependent message is not available, or if LANG is not set or is set to the empty string, the default version of the message associated with the **c** language is used.

International Code Set Support

Single and multi-byte character code sets are supported.

RETURN VALUE

The *perror* function returns no value.

If the *errnum* message number is valid, *strerror* returns a pointer to a language-dependent message string. The array pointed to should not be modified by the program, and might be overwritten by a subsequent call to the function. If a valid *errnum* message number does not have a corresponding language-dependent message, *strerror* uses *errnum* as an index into *sys_errlist* to get the message string. If the *errnum* message number is invalid, *strerror* returns a pointer to a NULL string.

SEE ALSO

errno(2), c(5), environ(5).

STANDARDS CONFORMANCE

perror: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

strerror: XPG3, ANSI C

sys_errlist: SVID2, XPG2

sys_nerr: SVID2, XPG2

NAME

popen, pclose – initiate pipe I/O to/from a process

SYNOPSIS

#include <stdio.h>

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

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

DESCRIPTION

Popen creates a pipe between the calling program and the command to be executed.

The arguments to *popen* are pointers to null-terminated strings containing, respectively, a shell command line and an I/O mode, either **r** for reading or **w** for writing.

Popen returns a stream pointer such that one can write to the standard input of the command, if the I/O mode is **w**, by writing to the file *stream;* and one can read from the standard output of the command, if the I/O mode is **r**, by reading from the file *stream*.

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

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

RETURN VALUE

Popen returns a NULL pointer if files or processes cannot be created. The success of the command execution can be checked by examining the return value of *pclose.*

Pclose returns **−1** if *stream* is not associated with a *"popened"* command.

WARNINGS

If the original and *"popened"* processes concurrently read or write a common file, neither should use buffered I/O, because the buffering will not work properly. Problems with an output filter may be forestalled by careful buffer flushing, e.g., with *fflush;* see *fclose*(3S).

SEE ALSO

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

STANDARDS CONFORMANCE

popen: SVID2, XPG2, XPG3

pclose: SVID2, XPG2, XPG3

NAME

printf, nl_printf, fprintf, nl_fprintf, sprintf, nl_sprintf – print formatted output

SYNOPSIS

#include <stdio.h>

int printf (format [, *arg*] ... **)**
const char *format;

int nl_printf (format [, *arg*] ... **)**
const char *format;

int fprintf (stream, format [, *arg*] ... **)**
FILE *stream;
const char *format;

int nl_fprintf (stream, format [, *arg*] ... **)**
FILE *stream;
const char *format;

int sprintf (s, format [, *arg*] ... **)**
char *s;
const char *format;

int nl_sprintf (s, format [, *arg*] ... **)**
char *s;
const char *format;

DESCRIPTION

Printf and *nl_printf* place output on the standard output stream *stdout*.

Fprintf and *nl_fprintf* place output on the named output *stream*.

Sprintf and *nl_sprintf* place "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 converts, formats, and prints its *arg*s under control of the *format*. The *format* is a character string containing two types of objects: plain characters that are copied to the output stream, and conversion specifications, each of which results in fetching zero or more *arg*s. The results are undefined if there are insufficient *arg*s for the format. If the format is exhausted while *arg*s remain, excess *arg*s are ignored.

Each conversion specification is introduced by the character **%** or **%*n*$**, where *n* is a decimal integer in the range (1-{NL_ARGMAX}) (NL_ARGMAX is defined in <**limits.h**>). The **%*n*$** construction indicates that this conversion should be applied to the *n*th argument, rather than to the next unused one.

An argument may be referenced by a **%*n*$** specification more than once. The two forms of introducing a conversion specification, **%** and **%*n*$**, may not be mixed within a single *format* string. Improper use of **%*n*$** in a format string will result in a negative return value.

After the **%** or **%*n*$**, the following appear in sequence:

> Zero or more *flags*, which modify the meaning of the conversion specification.
>
> An optional string of decimal digits to specify a minimum *field width* in bytes. 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. If the field width is preceded by a zero, the string is right adjusted with zero-padding on the left (see the leading-zero flag " " described below).
>
> A *precision* that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, or **X** conversions, the number of digits to appear after the radix character for the **e** and **f**

conversions, the maximum number of significant digits for the **g** conversion, or the maximum number of bytes to be printed from a string in the **s** conversion. The *precision* takes the form of a period (.) followed by a decimal digit string; a null digit string is treated as zero.

An optional **l** (the letter "ell"), specifying that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion character applies to a long integer *arg;* an optional **l** specifying that a following **n** conversion character applies to a pointer to a long integer *arg;* an optional **h**, specifying that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion character applies to a short integer *arg;* an optional **h** specifying that a following **n** conversion character applies to a pointer to a short integer *arg;* an optional **L** specifying that a following **e**, **E**, **f**, **g**, or **G** conversion character applies to a long double *arg*. An **l**, **h** or **L** before any other conversion character is ignored.

A conversion character 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 *arg* supplies the field width or precision. The *arg* that is actually converted is not fetched until the conversion letter is seen, so the *arg*s specifying field width or precision must appear in that order before the *arg* to be converted. Format strings containing **%***n***$** conversion specifications may also indicate a field width or precision by the sequence ***n***$**. The *n* indicates the position of an integer *arg*. With the ***n***$** sequence, the *arg*s specifying field width or precision can appear before or after the *arg* to be converted.

The flag characters and their meanings are:

–	The resulting conversion will be left-justified within the field.
+	The resulting signed conversion will always begin with a sign (+ or –).
blank	If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and + flags both appear, the blank flag will be ignored.
#	This flag specifies that the value is converted to an "alternate form." For **c**, **d**, **i**, **s**, **n**, and **u** conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x** or **X** conversion, a non-zero result will have **0x** or **0X** prefixed to it. For a **p** conversion, a non-zero result will have **0x** prefixed to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a radix character, even if no digits follow the radix (normally, a radix character appears in the resulting conversions only if followed by a digit). For **g** and **G** conversions, trailing zeroes will *not* be removed from the result (which they normally are).
0	Leading zeros (following any indication of sign or base) are used to pad to the field width for all conversion characters. No space padding is performed. If both the **0** and **-** appear, the **0** flag will be ignored. For **d**, **i**, **o**, **u**, **p**, **x**, and **X**, conversions, if a precision is specified, the **0** flag will be ignored.

The conversion characters and their meanings are:

d,i,o,u,x,X	The integer *arg* is converted to signed decimal (**d** and **i** are identical), unsigned octal (**o**), decimal (**u**), or hexadecimal notation (**x** and **X**), respectively; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. (For compatibility with older versions, padding with leading zeroes may alternatively be specified by prepending a zero to the field width. This does not imply an octal value for the field width.) The default

precision is **1**. The result of converting a zero value with a precision of zero is a null string.

f The double *arg* is converted to decimal notation in the style "[−]ddd**r**ddd", where **r** is the radix character. The number of digits after the radix character is equal to the precision specification. If the precision is missing, six digits are output. If the precision is explicitly zero, no radix character appears.

e,E The double *arg* is converted in the style "[−]d**r**ddde±ddd", where **r** is the radix character. There is one digit before the radix character and the number of digits after it is equal to the precision; when the precision is missing, six digits are produced; if the precision is zero, no radix character appears. The **E** format code will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits.

g,G The double *arg* is printed in style **f** or **e** (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style **e** will be used only if the exponent resulting from the conversion is less than **−4** or greater than or equal to the precision. Trailing zeroes are removed from the fractional part of the result; a radix character appears only if it is followed by a digit.

c The int *arg* is converted to an unsigned char, and the resulting character is printed.

s The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (**\0**) is encountered or the number of bytes indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed. A NULL value for *arg* will yield undefined results.

p The value of a pointer to void *arg* is printed as a sequence of unsigned hexadecimal numbers. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. The default precision is **1**. The result of converting a zero value with a precision of zero is a null string.

n A pointer to an integer *arg* is expected. This pointer is used to store the number of bytes printed on the output stream so far by this call to the function. No argument is converted.

% Print a **%**; no argument is converted.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

Characters generated by *printf*, *fprintf*, *nl_printf*, and *nl_fprintf* are printed as if *putc*(3S) had been called.

EXTERNAL INFLUENCES

Locale

The LC_CTYPE category affects the following features:

Plain characters within format strings are interpreted as single and/or multi-byte characters.

Field width is given in terms of bytes. As characters are placed on the output stream, they are interpreted as single or multi-byte characters and the field width is decremented by the length of the character.

Precision is given in terms of bytes. As characters are placed on the output stream, they are interpreted as single or multi-byte characters and the precision is decremented by the length of the character.

The return value is given in terms of bytes. As characters are placed on the output stream, they are interpreted as single or multi-byte characters and the byte count that makes up the return value is incremented by the length of the character.

The LC_NUMERIC category determines the radix character used to print floating-point numbers.

International Code Set Support

Single-byte character code sets are supported. Multi-byte character code sets are also supported as described in the LC_CTYPE category above.

RETURN VALUES

Each function returns the number of bytes transmitted (excluding the **\0** in the case of *sprintf* and *nl_sprintf*), or a negative value if an output error was encountered. Improper use of **%*n*$** in a format string will result in a negative return value.

EXAMPLES

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

printf("%s, %s %d, %d:%.2d", weekday, month, day, hour, min);

To print π to 5 decimal places:

printf("pi = %.5f", 4 * atan(1.0));

To create a language independent date and time printing routine write:

printf(format,weekday,month,day,hour,min,2,2);

For American usage, *format* would point to the string:

"%1$s, %2$s %3$d, %4$*6$.*7$d:%5$*6$.*7$d"

and result in the output:

Sunday, July 3, 10:02

For German usage, the string:

"%1$s, %3$s %2$d, %4$*6$.*7$d:%5$*6$.*7$d"

results in the output:

Sonntag, 3 Juli 10:02

WARNINGS

Nl_printf, *nl_fprintf* and *nl_sprintf* are provided for historical reasons only. Their use is not recommended. Use *printf*, *fprintf* and *sprintf* instead.

Notice that with the **c** conversion character, an int *arg* is converted to an unsigned char. Hence, whole multi-byte characters can not be printed using a single **c** conversion character.

A precision with the **s** conversion character might result in the truncation of a multi-byte character.

AUTHOR

Printf, *fprintf* and *sprintf* were developed by AT&T and HP. *Nl_printf*, *nl_fprintf* and *nl_sprintf* were developed by HP.

SEE ALSO

ecvt(3C), setlocale(3C), putc(3S), scanf(3S), stdio(3S).

STANDARDS CONFORMANCE

printf: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

fprintf: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

nl_fprintf: XPG2

nl_printf: XPG2

nl_sprintf: XPG2

sprintf: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

printmsg, fprintmsg, sprintmsg – print formatted output with numbered arguments

SYNOPSIS

#include <stdio.h>

int printmsg (format [, *arg*] ... **)**
char *format;

int fprintmsg (stream, format [, *arg*] ... **)**
FILE *stream;
char *format;

int sprintmsg (s, format [, *arg*] ... **)**
char *s, *format;

DESCRIPTION

Printmsg, fprintmsg, and *sprintmsg* are derived from their counterparts in *printf*(3S). The conversion character % can be replaced by %*digits*$. *Digits* are decimal digits representing a number *n* in the range (1-{NL_ARGMAX}) (NL_ARGMAX is defined in <**limits.h**>), and indicates that this conversion should be applied to the *n*th argument, rather than to the next unused one. All other aspects of formatting are unchanged. All conversion specifications must contain the %*digits*$ sequence and the user must ensure correct numbering. All parameters must be used exactly once.

EXAMPLES

To create a language-independent date and time printing routine, write

printmsg(format, weekday, month, day, hour, min);

For American usage *format* would point to the string:

"%1$s, %2$s %3$d, %4$d:%5$.2d"

resulting in the output:

Sunday, July 3, 10:02

For German usage, the string:

"%1$s, %3$d %2$s %4$d:%5$.2d"

results in the following output:

Sonntag, 3 Juli 10:02

provided that the proper strings have been read.

WARNINGS

These routines are provided for historical reasons only. Use of the *printf*(3S) equivalent routines *printf, fprintf* and *sprintf* is recommended.

AUTHOR

Printmsg was developed by HP.

SEE ALSO

catgetmsg(3C), setlocale(3C), printf(3S), hpnls(5).

EXTERNAL INFLUENCES

Locale

The LC_CTYPE category affects the following features:

- Plain characters within format strings are interpreted as single and/or multi-byte characters.

- Field width is given in terms of bytes. As characters are placed on the output stream, they are interpreted as single or multi-byte characters and the field width is decremented by the length of the character.
- Precision is given in terms of bytes. As characters are placed on the output stream, they are interpreted as single or multi-byte characters and the precision is decremented by the length of the character.
- The return value is given in terms of bytes. As characters are placed on the output stream, they are interpreted as single- or multi-byte characters and the byte count that makes up the return value is incremented by the length of the character.

The LC_NUMERIC category determines the radix character used to print floating-point numbers.

International Code Set Support

Single-byte character code sets are supported. Multi-byte character code sets are also supported as described in the LC_CTYPE category above.

NAME

putc, putchar, fputc, putw − put character or word on a stream

SYNOPSIS

#include <stdio.h>

int putc (c, stream)
int c;
FILE *stream;

int putchar (c)
int c;

int fputc (c, stream)
int c;
FILE *stream;

int putw (w, stream)
int w;
FILE *stream;

DESCRIPTION

Putc writes the character *c* onto the output *stream* at the position where the file pointer, if defined, is pointing. *Putchar*(*c*) is defined as **putc(***c***, stdout)**. *Putc* and *putchar* are defined as both macros and functions.

Fputc behaves like *putc*, but is a function rather than a macro; it may therefore be used as an argument. *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., **int** in C) *w* to the output *stream* (at the position at which the file pointer, if defined, is pointing). The size of a word is the size of an integer and varies from machine to machine. *Putw* neither assumes nor causes special alignment in the file.

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. *Setbuf*(3S) or *setvbuf* (see *setbuf*(3S)) may be used to change the stream's buffering strategy.

RETURN VALUE

On success, *putc*, *fputc*, and *putchar* each return the value they have written. On failure, they return the constant **EOF**. This will occur if the file *stream* is not open for writing or if the output file cannot be grown. The function *putw* returns non-zero when an error has occurred; otherwise the function returns **0**.

WARNINGS

The *putc* and *putchar* routines are implemented as both library functions and macros. The macro versions, which are used by default, are defined in <stdio.h>. To obtain the library function either use a #undef to remove the macro definition or, if compiling in ANSI-C mode, enclose the function name in parenthesis or use the function address. For following example illustrates each of these methods :

```
#include <stdio.h>
#undef putc
...
main()
{
        int (*put_char()) ();
        ...
```

```
        return_val=putc(c,fd);
        ...
        return_val=(putc)(c,fd1);
        ...
        put_char = putchar;
};
```

Line buffering may cause confusion or malfunctioning of programs that use standard I/O routines but use *read*(2) themselves to read from standard input. When a large amount of computation is done after printing part of a line on an output terminal, it is necessary to *fflush* (on *fclose*(3S)) the standard output before beginning the computation.

The macro version of *putc* incorrectly treats the argument *stream* with side effects. In particular, the followng call may not work as expected:

putc(c, *f++);

The function version of *putc* or *fputc* should be used instead.

Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent, and may not be read using *getw* on a different processor.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), getc(3S), fread(3S), printf(3S), puts(3S), setbuf(3S).

STANDARDS CONFORMANCE

putc: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

fputc: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

putchar: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

putw: SVID2, XPG2, XPG3

NAME

putenv – change or add value to environment

SYNOPSIS

int putenv (string)
char *string;

DESCRIPTION

String points to a string of the form *"name=value." Putenv* makes the value of the environment variable *name* equal to *value* by altering an existing variable or creating a new one. In either case, the string pointed to by *string* becomes part of the environment, so altering the string will change the environment. The space used by *string* is no longer used once a new string-defining *name* is passed to *putenv*.

DIAGNOSTICS

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

WARNINGS

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

SEE ALSO

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

EXTERNAL INFLUENCES

Locale

The LC_CTYPE category determines the interpretation of characters in *string* as single- and/or multi-byte characters.

International Code Set Support

Single- and multi-byte character code sets are supported.

STANDARDS CONFORMANCE

putenv: SVID2, XPG2, XPG3

NAME

putpwent – write password file entry

SYNOPSIS

#include <pwd.h>

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

DESCRIPTION

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

Putpwent ignores the audit ID and audit flag in the *passwd* structure; and *does not* create the corresponding entries used in the secure password file (**/.secure/etc/passwd**). *Putspwent*(LIBC) which produces entries that match the secure password file format, must be used to create these entries.

DIAGNOSTICS

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

SEE ALSO

getpwent(3C), putspwent(3C), passwd(4), spasswd(4).

STANDARDS CONFORMANCE

putpwent: SVID2, XPG2

NAME

puts, fputs – put a string on a stream

SYNOPSIS

#include <stdio.h>

int puts (s)
char *s;

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

DESCRIPTION

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

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

Neither function writes the terminating null character. Note that *puts* appends a new-line character, but *fputs* does not.

RETURN VALUE

Both routines return **EOF** on error. This will happen if the routines try to write on a file that has not been opened for writing. A non-negative number is returned on success.

SEE ALSO

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

NOTES

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

STANDARDS CONFORMANCE

puts: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

fputs: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

putspwent – write secure password file entry

SYNOPSIS

#include <pwd.h>

int putspwent (p, f)
struct s_passwd *p;
FILE *f;

DESCRIPTION

Putspwent is the inverse of *getspwent*(3C). Given a pointer to a **s_passwd** structure, as created by *getspwent*(3C), *putspwent* writes a line on the stream *f*, which matches the format of **/.secure/etc/passwd**.

RETURN VALUE

Putspwent returns non-zero if it detects an error during its operation; otherwise it returns a value of zero.

AUTHOR

Putspwent was developed by HP.

SEE ALSO

getpwent(3C), getspwent(3C), putpwent(3C), spasswd(4).

NAME

qsort − quicker sort

SYNOPSIS

#include <stdlib.h>

```
void qsort (base, nel, size, compar)
void *base;
size_t nel;
size_t size;
int (*compar)( );
```

DESCRIPTION

Qsort is an implementation of the quicker-sort algorithm. It sorts a table of data in place.

Base points to the element at the base of the table. *Nel* is the number of elements in the table. *Size* is the size of each element in the table. *Compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function passed as *compar* must return an integer less than, equal to, or greater than zero as a consequence of whether its first argument is to be considered less than, equal to, or greater than the second. This is the same return convention that *strcmp*(3C) uses.

NOTES

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

SEE ALSO

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

BUGS

If *size* is zero, a divide-by-zero error may be generated.

STANDARDS CONFORMANCE

qsort: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

rand, srand – simple random-number generator

SYNOPSIS

int rand ( )

void srand (seed)
unsigned seed;

DESCRIPTION

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

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

NOTE

The spectral properties of *rand* leave a great deal to be desired. *Drand48*(3C) provides a much better, though more elaborate, random-number generator.

SEE ALSO

drand48(3C).

STANDARDS CONFORMANCE

rand: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

srand: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

regcmp, regex – compile and execute regular expression

SYNOPSIS

char *regcmp (string1 [, string2, ...], (char *)0)
char *string1, *string2, ...;

char *regex (re, subject[, ret0, ...])
char *re, *subject, *ret0, ...;

extern char *__loc1;

DESCRIPTION

Regcmp compiles a regular expression and returns a pointer to the compiled form. *Malloc*(3C) is used to create space for the vector. It is the user's responsibility to free unneeded space so allocated. A NULL return from *regcmp* indicates an incorrect argument.

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 largely 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.

– Used within brackets the hyphen signifies a character range. For example, **[a–z]** is equivalent to **[abcd...xyz]**. The – can represent 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 can be applied. The value *m* is the minimum number and *u* is a maximum number, which must be no greater than 256. The syntax {*m*} indicates the exact number of times the regular expression can be applied. The syntax {*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 is stored in the *(n+1)*th argument following the subject argument. A maximum of ten enclosed regular expressions are allowed. *Regex* makes its assignments unconditionally.

(...) Parentheses are used for grouping. An operator, such as *, +, or { }, can work on a single character or a regular expression enclosed in parentheses. For example, (a*(cb+)*)$0.

Since all of the above defined symbols are special characters, they must be escaped to be used as themselves.

This routine is kept in **/lib/libPW.a**.

EXAMPLES

Example 1:

```
char *cursor, *newcursor, *ptr;
 ...
newcursor = regex((ptr = regcmp("^\n", 0)), cursor);
```

```
free(ptr);
```

This example matches a leading new-line in the subject string to which the *cursor* points.

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" will be copied to the character array *ret0*.

WARNINGS

The user program might run out of memory if *regcmp* is called iteratively without freeing the vectors that are no longer required.

SEE ALSO

ed(1), malloc(3C).

NAME

compile, step, advance – regular expression compile and match routines

SYNOPSIS

```
#define INIT <declarations>
#define GETC() <getc code>
#define PEEKC() <peekc code>
#define UNGETC(c) <ungetc code>
#define RETURN(pointer) <return code>
#define ERROR(val) <error code>

#include <regexp.h>

char *compile (instring, expbuf, endbuf, eof)
char *instring, *expbuf, *endbuf;
int eof;

int step (string, expbuf)
char *string, *expbuf;

int advance (string, expbuf)
char *string, *expbuf;

extern char *loc1, *loc2, *locs;

extern int circf, sed, nbra;
```

DESCRIPTION

These functions are general-purpose regular expression matching routines to be used in programs that perform Basic Regular Expression (see *regexp*(5)) matching. These functions are defined in **<regexp.h>**.

The functions *step* and *advance* do pattern matching given a character string and a compiled regular expression as input. The function *compile* takes as input a Basic Regular Expression and produces a compiled expression that can be used with *step* and *advance*.

The interface to this file is unpleasantly complex. Programs that include this file must have the following five macros declared before the **#include <regexp.h>** statement. These macros are used by the *compile* routine.

GETC() Return the value of the next byte in the regular expression pattern. Successive calls to GETC() should return successive bytes of the regular expression.

PEEKC() Return the next byte in the regular expression. Successive calls to PEEKC() should return the same byte (which should also be the next byte returned by GETC()).

UNGETC(*c*) Cause the argument *c* to be returned by the next call to GETC() (and PEEKC()). No more than one byte of pushback is ever needed and this byte is guaranteed to be the last byte read by GETC(). The value of the macro UNGETC(*c*) is always ignored.

RETURN(*pointer*)
This macro is used on normal exit of the *compile* routine. The value of the argument *pointer* is a pointer to the character after the last character of the compiled regular expression. This is useful to programs that have memory allocation to manage.

ERROR(*val*) This is the abnormal return from the *compile* routine. The argument *val* is an error number (see table below for meanings). This call should never return.

ERROR	MEANING
11	Range endpoint too large.
16	Bad number.
25	"\digit" out of range.
36	Illegal or missing delimiter.
41	No remembered search string.
42	\(\) imbalance.
43	Too many \(.
44	More than 2 numbers given in \{ \}.
45	} expected after \.
46	First number exceeds second in \{ \}.
49	[ ] imbalance.
50	Regular expression overflow.

The syntax of the *compile* routine is as follows:

```
compile(instring, expbuf, endbuf, eof)
```

The first parameter *instring* is never used explicitly by the *compile* routine but is useful for programs that pass down different pointers to input characters. It is sometimes used in the INIT declaration (see below). Programs which call functions to input characters or have characters in an external array can pass down a value of **((char *) 0)** for this parameter.

The next parameter *expbuf* is a character pointer. It points to the place where the compiled regular expression will be placed.

The parameter *endbuf* is one more than the highest address where the compiled regular expression can be placed. If the compiled expression cannot fit in (*endbuf−expbuf*) bytes, a call to ERROR(50) is made.

The parameter *eof* is the character which marks the end of the regular expression. For example, in *ed*(1), this character is usually a /.

Each program that includes this file must have a **#define** statement for INIT. This definition is placed right after the declaration for the function *compile* and the opening curly brace {. It is used for dependent declarations and initializations. Most often it is used to set a register variable to point to the beginning of the regular expression so that this register variable can be used in the declarations for GETC(), PEEKC() and UNGETC(). Otherwise it can be used to declare external variables that might be used by GETC(), PEEKC() and UNGETC(). See the example below of the declarations taken from *grep*(1).

The function *step* also performs actual regular expression matching in this file. The call to *step* is as follows:

```
step(string, expbuf)
```

The first parameter to *step* is a pointer to a string of characters to be checked for a match. This string should be null terminated.

The second parameter *expbuf* is the compiled regular expression that was obtained by a call of the function *compile*.

The function *step* returns non-zero if the given string matches the regular expression, and zero if the expressions do not match. If there is a match, two external character pointers are set as a side effect to the call to *step*. The variable set in *step* is *loc1*. This is a pointer to the first character that matched the regular expression. The variable *loc2*, which is set by the function *advance*, points to the character after the last character that matches the regular expression. Thus, if the regular expression matches the entire line, *loc1* points to the first character of *string* and *loc2* points to the null at the end of *string*.

Step uses the external variable *circf*, which is set by *compile* if the regular expression begins with ^. If this is set, *step* tries to match the regular expression to the beginning of the string only. If more than one regular expression is to be compiled before the first is executed, the value of *circf* should be saved for each compiled expression and *circf* should be set to that saved value before each call to *step*.

The function *advance* is called from *step* with the same arguments as *step*. The purpose of *step* is to step through the *string* argument and call *advance* until *advance* returns non-zero, which indicates a match, or until the end of *string* is reached. To constrain *string* to the beginning of the line in all cases, *step* need not be called; simply call *advance*.

When *advance* encounters a * or \{ \} sequence in the regular expression, it advances its pointer to the string to be matched as far as possible and recursively calls itself, trying to match the rest of the string to the rest of the regular expression. As long as there is no match, *advance* backs up along the string until it finds a match or reaches the point in the string that initially matched the * or \{ \}. It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external character pointer *locs* is equal to the point in the string at sometime during the backing up process, *advance* breaks out of the loop that backs up and returns zero. This is used by *ed*(1) and *sed*(1) for substitutions done globally (not just the first occurrence, but the whole line) so, for example, expressions such as **s/y*//g** do not loop forever.

The additional external variables *sed* and *nbra* are used for special purposes.

EXTERNAL INFLUENCES

Locale

The LC_COLLATE category determines the collating sequence used in compiling and executing regular expressions.

The LC_CTYPE category determines the interpretation of text as single and/or multi-byte characters, and the characters matched by character class expressions in regular expressions.

International Code Set Support

Single- and multi-byte character code sets are supported.

EXAMPLES

The following is an example of how the regular expression macros and calls look from *grep*(1):

```
#define INIT             register char *sp = instring;
#define GETC()           (*sp++)
#define PEEKC()          (*sp)
#define UNGETC(c)        (--sp)
#define RETURN(c)        return;
#define ERROR(c)         regerr()

#include <regexp.h>
...
        (void) compile(*argv, expbuf, &expbuf[ESIZE], '\0');
...
        if (step(linebuf, expbuf))
                succeed();
```

SEE ALSO

grep(1), setlocale(3C), regexp(5).

STANDARDS CONFORMANCE

regexp: SVID2, XPG2, XPG3

advance: SVID2, XPG2, XPG3

compile: SVID2, XPG2, XPG3
loc1: SVID2, XPG2, XPG3
loc2: SVID2, XPG2, XPG3
locs: SVID2, XPG2, XPG3
step: SVID2, XPG2, XPG3

NAME
remove – remove a file

SYNOPSIS
#include <stdio.h>

int remove (path)
const char *path;

DESCRIPTION
Remove removes the file named by *path*. If *path* does not name a directory, **remove(path)** is equivalent to **unlink(path)**. If *path* names a directory, **remove(path)** is equivalent to **rmdir(path)**.

SEE ALSO
rmdir(2), unlink(2).

STANDARDS CONFORMANCE
remove: XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

RomajiHiragana, RomajiKatakana, RomajiHankakuKatakana– translate characters

SYNOPSIS

```
#include <jlib.h>

unsigned char *RomajiHiragana (s1, s2)
char *s1;
unsigned char *s2;

unsigned char *RomajiKatakana (s1, s2)
char *s1;
unsigned char *s2;

unsigned char *RomajiHankakuKatakana (s1, s2)
char *s1;
unsigned char *s2;
```

DESCRIPTION

The arguments *s1* and *s2* point to strings (arrays of characters terminated by a null character). The string *s1* is ROMAJI, which is an alphabetic representation of Japanese characters. Each character included in *s1* must be an 8-bit alphabet.

RomajiHiragana translates *s1* to string *s2* spelled by HIRAGANA. *RomajiKatakana* translates *s1* to string *s2* spelled by KATAKANA. *RomajiHankakuKatakana* translates *s1* to string *s2* spelled by HANKAKU KATAKANA.

Translation is performed based on *romaji*(5) which shows how Japanese is spelled using Roman characters.

DIAGNOSTICS

Each function returns a NULL pointer upon successful completion.

If string *s1* contains illegal or undetermined ROMAJI spelling, each function returns a pointer to the first character of the ROMAJI spelling and **jlib_errno** is set to indicate translation result.

[JNEEDMORE]	The string *s1* is undetermined ROMAJI.
[JNOTFOUND]	The string *s1* is unacceptable ROMAJI.
[JNOTRESPOND]	A server does not respond.

WARNINGS

Each function cannot check for overflow of any receiving string. The length of the resultant string is twice the length of *s1* at most. NULL destinations cause errors. NULL sources are treated as zero-length strings.

SEE ALSO

open_jlib(3X), romaji(5)

NAME

scanf, fscanf, sscanf, nl_scanf, nl_fscanf, nl_sscanf – formatted input conversion, read from stream file

SYNOPSIS

#include <stdio.h>

int scanf (format [, *pointer*] ... **)**
const char *format;

int fscanf (stream, format [, *pointer*] ... **)**
FILE *stream;
const char *format;

int sscanf (s, format [, *pointer*] ... **)**
char *s;
const char *format;

int nl_scanf (format [, *pointer*] ... **)**
const char *format;

int nl_fscanf (stream, format [, *pointer*] ... **)**
FILE *stream;
const char *format;

int nl_sscanf (s, format [, *pointer*] ... **)**
char *s;
const char *format;

DESCRIPTION

Scanf and *nl_scanf* read from the standard input stream *stdin*.

Fscanf and *nl_fscanf* read from the named input *stream*.

Sscanf and *nl_sscanf* read from the character string *s*.

Each function reads characters, interprets them according to the control string *format* argument, and stores the results in its *pointer* arguments. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are ignored. The control string contains conversion specifications and other characters used to direct interpretation of input sequences. The control string contains:

- White-space characters (blanks, tabs, newlines, or formfeeds) that cause input to be read up to the next non-white-space character (except in two cases described below).
- An ordinary character (not %) that must match the next character of the input stream.
- Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum-field width, an optional **l** (ell), **h** or **L** indicating the size of the receiving variable, and a conversion code.
- The conversion specification may alternatively be prefixed by the character sequence %*n*$ instead of the character %, where *n* is a decimal integer in the range (1-{NL_ARGMAX}) (NL_ARGMAX is defined in <**limits.h**>). The %*n*$ construction indicates that the value of the next input field should be placed in the *n*th argument, rather than to the next unused one. The two forms of introducing a conversion specification, % and %*n*$, may not be mixed within a single *format* string with the following exception: Skip fields (see below) can be designated as %* or %*n*$*. In the latter case, *n* is ignored.

Unless the specification contains the **n** conversion character (described below), a conversion specification directs the conversion of the next input field. The result of a conversion specification is placed in the variable to which the corresponding argument points, unless * indicates assignment suppression. Assignment suppression provides a way to describe an input field to be skipped. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted. For all descriptors except "[" and "c", white space leading an input field is ignored.

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

% A single % is expected in the input at this point; no assignment is done.

d A decimal integer is expected; the corresponding argument should be an integer pointer.

u An unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.

o An octal integer is expected; the corresponding argument should be an unsigned integer pointer.

x,X A hexadecimal integer is expected; the corresponding argument should be an unsigned integer pointer. The **x** and **X** conversion characters behave the same.

i An integer is expected; the corresponding argument should be an integer pointer. The value of the next input item, interpreted according to C conventions, will be stored; a leading **0** implies octal, a leading **0x** implies hexadecimal; otherwise, decimal is assumed.

n Cause the total number of bytes (including white space) scanned since the function call to be stored; the corresponding argument should be an integer pointer. No input is consumed. The function return value does not include **%n** assignments in the count of successfully matched and assigned input items.

e,E,f,g,G A floating-point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a *float*. The input format for floating-point numbers is an optionally signed string of digits, possibly containing a radix character, followed by an optional exponent field consisting of an **E** or an **e**, followed by an optional +, −, or space, followed by an integer. The conversion characters **E** and **G** behave the same as, respectively, **e** and **g**.

s A character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating **\0**, which is added automatically. The input field is terminated by a white-space character. *Scanf* will not read a null string.

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

[Indicates string data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, called the *scanset*, and a right bracket; the input field is the maximal sequence of input characters consisting entirely of characters in the scanset. The circumflex (^), when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters *not* contained in the remainder of the scanset string. Construction of the *scanset* follows certain conventions. A range of characters may be represented by the construct *first–last*, enabling [0123456789] to be expressed [0–9]. Using this convention, *first* must be lexically less than or equal to *last*; otherwise, the dash stands for itself. The dash also stands for itself when it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset, in which case it will not be interpreted syntactically as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating **\0**, which are added automatically. At least one character must match for this conversion to succeed.

p A sequence of unsigned hexadecimal numbers is expected. This sequence may be produced by the **p** conversion character of *printf*. The corresponding argument shall be a pointer to a pointer to **void** into which the value represented by the hexadecimal sequence is stored. The behavior of this conversion is undefined for any input item other than a value converted earlier during the same program execution.

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

The *scanf* functions terminate their conversions at **EOF**, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

EXTERNAL INFLUENCES

Locale

The LC_CTYPE category determines the interpretation of ordinary characters within format strings as single and/or multi-byte characters. Field width is given in terms of bytes. Characters received from the input stream are interpreted as single or multi-byte characters as determined by the LC_TYPE category and the field width is decremented by the length of the character.

The LC_NUMERIC category determines the radix character expected within floating-point numbers.

International Code Set Support

Single and multi-byte character code sets are supported.

RETURN VALUES

If the input ends before the first conflict or conversion, **EOF** is returned. Otherwise, these functions return the number of successfully assigned input items. This number is a short count, or even zero, if a conflict ensues between an input character and the control string.

EXAMPLES

The call:

```
int i, n; float x; char name[50];
n = scanf("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E−1 thompson
```

will assign to *n* the value **3**, to *i* the value **25**, to *x* the value **5.432**, and *name* will contain **thompson\0**. Or:

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

with input:

```
56789 0123 56a72
```

will assign **56** to *i*, **789.0** to *x*, skip **0123**, and place the string **56\0** in *name*. The next call to *getchar* (see *getc*(3S)) will return **a**.

For another example, to create a language-independent date scanning routine, write:

```
char month[20]; int day, year;
(void) scanf(format, month, &day, &year);
```

For American usage, *format* would point to a string:

```
"%1$s %2$d %3$d"
```

The input:

```
July 3 1986
```

would assign **July** to *month*, **3** to *day* and **1986** to *year*.

For German usage, *format* would point to a string:

```
"%2$d %1$s %3$d"
```

The input:

```
3 Juli 1986
```

would assign **Juli** to *month*, **3** to *day* and **1986** to *year*.

The success of literal matches and suppressed assignments can be determined with the **%n** conversion specification. Here is an example that checks the success of literal matches:

```
int i, n1, n2, n3, n4;
n1 = n2 = n3 = n4 = -1;
scanf( "%nBEGIN%n %d %nEND%n", &n1, &n2, &i, &n3, &n4);
if (n2 - n1 == 5) puts( "matched BEGIN");
if (n4 - n3 == 3) puts( "matched END");
```

Here is an example that checks the success of suppressed assignments:

```
int i, n1, n2;
n1 = n2 = -1;
scanf( "%d %n%*s%n", &i, &n1, &n2);
if (n2 > n1)
    printf( "successful assignment suppression of %d chars\n", n2 - n1);
```

WARNINGS

Trailing white space (including a newline) is left unread unless matched in the control string.

Truncation of multi-byte characters may occur if a field width is used with the conversion character.

Nl_scanf, *nl_fscanf* and *nl_sscanf* are provided for historical reasons only. Their use is not recommended. Use *scanf*, *fscanf* and *sscanf* instead.

DEPENDENCIES

Series 300

The **−i** and **−n** conversion codes are not currently recognized.

AUTHOR

Scanf was developed by AT&T and HP.

SEE ALSO

getc(3S), setlocale(3C), printf(3S), strtod(3C), strtol(3C).

STANDARDS CONFORMANCE

scanf: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

fscanf: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

nl_fscanf: XPG2

nl_scanf: XPG2

nl_sscanf: XPG2

sscanf: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

setaclentry, fsetaclentry – add, modify, or delete one entry in file's access control list (ACL)

SYNOPSIS

#include <unistd.h>
#include <acllib.h>

int setaclentry (path, uid, gid, mode)
char *path;
int uid, gid;
int mode;

int fsetaclentry (fd, uid, gid, mode)
int fd;
int uid, gid;
int mode;

Remarks:

To ensure continued conformance with emerging industry standards, features described in this manual entry are likely to change in a future release.

DESCRIPTION

Both forms of this call add, modify, or delete one entry in a file's access control list (ACL). *Setaclentry* and *fsetaclentry* take a path name (*path*) or open file descriptor (*fd*) and an entry identifier (*uid, gid*). They change the indicated entry's access mode bits to the given value (*mode*), meanings of which are defined in <**unistd.h**>. *Modes* are represented as R_OK, W_OK, and X_OK. Irrelevant bits in *mode* values must be zero.

If the file's ACL does not have an entry for the given *uid* and *gid*, the entry is created and added to the ACL. If *mode* is MODE_DEL (defined in <**acllib.h**>), the matching entry is deleted from the file's ACL if it is an optional entry, or its mode bits are set to zero (no access) if it is a base entry.

Uid or *gid* can be ACL_NSUSER or ACL_NSGROUP (defined in <**sys/acl.h**>), respectively, to represent non-specific entries *u*.%, %.*g*, or %.%. The file's *u*.% or %.*g* base entries can be referred to using ACL_FILEOWNER or ACL_FILEGROUP (defined in <**acllib.h**>), for the file's owner or group ID, respectively.

Setaclentry and *fsetaclentry* read the file's ACL with *getacl*(2) or *fgetacl*(2) and modify it with *setacl*(2) or *fsetacl*(2), respectively.

RETURN VALUE

If successful, *setaclentry* and *fsetaclentry* return zero.

ERRORS

If an error occurs, *setaclentry* and *fsetaclentry* return the following negative values and set **errno**:

−1 Unable to perform *getacl* or *fgetacl* on the file. **Errno** indicates the cause.

−2 Unable to perform *stat* or *fstat* on the file. **Errno** indicates the cause.

−3 Cannot add a new entry because the ACL already has NACLENTRIES (defined in <**sys/acl.h**>) entries.

−4 Cannot delete a nonexisting entry.

−5 Unable to perform *setacl* or *fsetacl* on the file. **Errno** indicates the cause.

EXAMPLES

The following code fragment adds an entry to file "work/list" for user ID 115, group ID 32, or modifies the existing entry for that user and group, if any, with a new access mode of read only. It also changes the owner base entry to have all access rights, and deletes the entry, if any, for any user in group 109.

```
#include <unistd.h>
#include <acllib.h>

char *filename = "work/list";

setaclentry (filename, 115, 32, R_OK);
setaclentry (filename, ACL_FILEOWNER, ACL_NSGROUP, R_OK | W_OK | X_OK);
setaclentry (filename, ACL_NSUSER, 109, MODE_DEL);
```

DEPENDENCIES

RFA and NFS

Setaclentry and *fsetaclentry* are not supported on remote files.

AUTHOR

Setaclentry and *fsetaclentry* were developed by HP.

SEE ALSO

getacl(2), setacl(2), stat(2), acltostr(3C), cpacl(3C), chownacl(3C), strtoacl(3C), acl(5).

NAME

setbuf, setvbuf – assign buffering to a stream file

SYNOPSIS

#include <stdio.h>

void setbuf (stream, buf)
FILE *stream;
char *buf;

int setvbuf (stream, buf, type, size)
FILE *stream;
char *buf;
size_t type, size;

DESCRIPTION

Setbuf may be used after a stream has been opened but before it is read or written. It causes the array pointed to by *buf* to be used instead of an automatically allocated buffer. If *buf* is the NULL pointer input/output will be completely unbuffered.

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

char buf[BUFSIZ];

Setvbuf may be used after a stream has been opened but before it is read or written. *Type* determines how *stream* 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.

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). *Fflush* can also be used to explicitly write the buffer.

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

By default, output to a terminal is line buffered and all other input/output is fully buffered.

SEE ALSO

fopen(3S), getc(3S), malloc(3C), putc(3S), stdio(3S).

DIAGNOSTICS

If an illegal value for *type* or *size* is provided, *setvbuf* returns a non-zero value. Otherwise, the value returned will be zero.

NOTE

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.

STANDARDS CONFORMANCE

setbuf: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

setvbuf: SVID2, XPG2, XPG3, ANSI C

NAME

setjmp, longjmp, sigsetjmp, siglongjmp – non-local goto

SYNOPSIS

```
#include <setjmp.h>

int setjmp (env)
jmp_buf env;

void longjmp (env, val)
jmp_buf env;
int val;

int _setjmp(env)
jmp_buf env;

void _longjmp(env, val)
jmp_buf env;
int val;

int sigsetjmp (env, savemask)
sigjmp_buf env;
int savemask;

void siglongjmp (env, val)
sigjmp_buf env;
int val;
```

DESCRIPTION

These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setjmp saves its stack environment in *env* (whose type, *jmp_buf*, is defined in the **<setjmp.h>** header file) for later use by *longjmp*. It returns the value **0**.

Longjmp restores the environment saved by the last call of *setjmp* with the corresponding *env* argument. After *longjmp* is completed, program execution continues as if the corresponding call of *setjmp* (which must not itself have returned in the interim) had just returned the value *val*. *Longjmp* cannot cause *setjmp* to return the value **0**. If *longjmp* is invoked with a second argument of **0**, *setjmp* returns **1**. All accessible data have values as of the time *longjmp* is called.

Upon the return from a *setjmp* call caused by a *longjmp*, the values of any non-static local variables belonging to the routine from which *setjmp* was called are undefined. Code which depends on such values is not guaranteed to be portable.

The two pairs of functions, *_setjmp* and *_longjmp* and *sigsetjmp* and *siglongjmp* behave identically to *setjmp* and *longjmp* except in the handling of the process' signal mask (see *sigaction*(2) and *sigvector*(2)). This distinction is only significant for programs which use *sigaction*(2), *sigprocmask*(2), *sigvector*(2), *sigblock*(2), and/or *sigsetmask*(2). *Setjmp* and *longjmp* always save and restore the signal mask. *_setjmp* and *_longjmp* never manipulate the signal mask. *Sigsetjmp* saves the signal mask if and only if *savemask* is non-zero. *Siglongjmp* restores the signal mask if and only if it is saved by *sigsetjmp*. The names *setjmp* and *longjmp* are used in a generic sense to describe all three variants.

If a *longjmp* is executed and the environment in which the *setjmp* is executed no longer exists, errors can occur. The conditions under which the environment of the *setjmp* no longer exists include exiting the procedure that contains the *setjmp* call, and exiting an inner block with temporary storage (such as a block with declarations in C or a *with* statement in Pascal). This condition might not be detectable, in which case the *longjmp* occurs, and if the environment no longer exists, the contents of the temporary storage of an inner block are unpredictable. This condition might also cause unexpected process termination. If the procedure has been exited the results are unpredictable.

Passing *longjmp* a pointer to a buffer not created by *setjmp*, passing *_longjmp* a pointer to a buffer not created by either *setjmp* or *_setjmp*, passing *siglongjmp* a pointer to a buffer not created by *sigsetjmp* or passing any of these three functions a buffer that has been modified by the user, can cause all the problems listed above, and more.

Some implementations of Pascal support a "try/recover" mechanism, which also creates stack marker information. If a *longjmp* operation occurs in a scope which is nested inside a try/recover, and the corresponding *setjmp* is not inside the scope of the try/recover, the recover block will not be executed and the currently active recover block will become the one enclosing the *setjmp*, if one exists.

WARNINGS

A call to *longjmp* to leave the guaranteed stack space reserved by *sigspace*(2) might remove the guarantee that the ordinary execution of the program will not extend into the guaranteed space. It might also cause the program to forever lose its ability to automatically increase the stack size, and the program might then be limited to the guaranteed space.

The result of using *setjmp* within an expression can be unpredictable.

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.

AUTHOR

Setjmp was developed by AT&T and HP.

SEE ALSO

sigaction(2), sigblock(2), signal(5), sigprocmask(2), sigsetmask(2), sigspace(2), sigsuspend(2), sigvector(2).

STANDARDS CONFORMANCE

setjmp: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

longjmp: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

siglongjmp: XPG3, POSIX.1, FIPS 151-1

sigsetjmp: XPG3, POSIX.1, FIPS 151-1

NAME

setlocale, getlocale – set and get the locale of a program

SYNOPSIS

#include <locale.h>

char *setlocale(category, locale)
const int category;
const char *locale;

struct locale_data *getlocale(type)
int type;

DESCRIPTION

Setlocale will set, query or restore that aspect of a program's locale as specified by the *category* argument. A program's locale refers to those areas of the program's Native Language Support (NLS) environment for which the following values of *category* have been defined:

LC_ALL — affects the behavior of all categories below as well as all *nl_langinfo*(3C) items. Note that some *nl_langinfo* items are only affected by the setting of the LC_ALL category.

LC_COLLATE — affects the behavior of regular expressions and the NLS string collation functions (see *string*(3C), and *regexp*(5)).

LC_CTYPE — affects the behavior of regular expressions, character classification and conversion functions (see *ctype*(3C), *conv*(3C), and *regexp*(5)). The LC_CTYPE category also affects the behavior of all routines which process multibyte characters (see *multibyte*(3C) and *nl_tools_16*(3C)).

LC_MONETARY — affects the behavior of functions which handle monetary values (see *localeconv*(3C)).

LC_NUMERIC — affects the handling of the radix character in the formatted input/output functions (see *printf*(3C), *scanf*(3C) and *vprintf*(3C)) and the string conversion functions (see *ecvt*(3C) and *strtod*(3C)). LC_NUMERIC also affects the numeric values found in the *localeconv* structure.

LC_TIME — affects the behavior of time conversion functions (see *strftime(3C)).*

All *nl_langinfo*(3C) items are affected by the setting of one of the categories listed above. See *langinfo*(5) to determine which category affect each item.

The value of the *locale* argument will determine the action taken by *setlocale. Locale* is a pointer to a character string.

Setting the Locale of a Program

To set the program's locale for *category, setlocale* will accept one of the following values as the *locale* argument: *locale name* ,"C", or "" (the empty string). The actions prescribed by these values are as follows :

locale name — If *locale* is a valid locale name (see *lang(5)), setlocale* will set that part of the NLS environment associated with *category* as defined for that locale.

"C" — If the value of *locale* is set to "C", *setlocale* will set that part of the NLS environment associated with *category* as defined for the "C" locale (see *lang(5)).* The "C" locale is the default prior to successfully calling *setlocale.*

"" — If the value of *locale* is the empty string, the setting of that part of the NLS environment associated with *category* will depend on the setting of the following environment variables in the user's environment (see *environ(5)) :*

LANG

LC_COLLATE

LC_CTYPE

LC_MONETARY

LC_NUMERIC

LC_TIME

If *category* is any defined value other than LC_ALL, *setlocale* will set that category as specified by the value of the corresponding environment variable. If the environment variable is not set or set to the empty string, *setlocale* will set the category as specified by the value of the LANG environment variable. If LANG is not set or is set to the empty string , then *setlocale* will set the category to the "C" locale. For example, setlocale(LC_TIME,"") will set the program's NLS environment associated with the LC_TIME *category* to the value specified by the user's LC_TIME environment variable. All other aspects of the NLS environment will be unaffected.

If *category* is LC_ALL, then all categories will be set corresponding to the value of LANG, except for those categories in which the corresponding environment variable is set to a valid language name (see *lang(5)).* In this case the value of the environment variable will override the value of LANG for that category. If the value of LANG is not set or is set to the empty string, then the "C" locale is used.

The following usage of setlocale will result in the program's locale being set according to the the user's language requirements:

setlocale(LC_ALL,"");

Querying the Locale of a Program

Setlocale will query the current NLS environment pertaining to *category* if the value of *locale* is NULL. The query operation will not change the environment. The purpose of performing a query is to save that aspect of the user's current NLS environment associated with *category,* in the value returned by *setlocale,* such that it can be restored with a subsequent call to *setlocale.*

Restoring the Locale of a Program

To restore a category within the program locale, a *setlocale* call is made with the same *category* argument and the return string of the previous *setlocale* call given as the *locale* argument.

The *getlocale* function will return a pointer to a **locale_data** structure (see **/usr/include/locale.h**). The members of the **locale_data** structure contain information about the setting of each setlocale category. *Type* determines what information is contained in the **locale_data** structure. Defined values of *type* and their behaviour are :

LOCALE_STATUS
: The structure member corresponding to each category will contain a string with the name of the locale currently set for that category. The string will not include modifier information.

MODIFIER_STATUS
: The structure member corresponding to each category will contain a string with the name of the modifier currently set for that category. If no modifier is set then the entry will contain an empty string.

ERROR_STATUS
: The structure member will contain information about errors which occurred

during the previous call to *setlocale*. If *setlocale* could not satisfy a request corresponding to a particular *category*, the structure member for that category will contain a string with the name of the invalid locale. In all other cases the member for the category will contain an empty string.

RETURN VALUE

If the pointer to a string is given for *locale* and the selection can be honored, the *setlocale* function returns a pointer to the string associated with the specified *category for the new locale. The* is LC_BUFSIZ bytes (see <**locale.h**>). If the selection cannot be honored, the *setlocale* function returns a null pointer and the program's locale is not changed.

A null pointer for *locale* causes *setlocale* to return a string associated with the *category* for the program's current locale.

The string returned by *setlocale* is such that a subsequent call with that string as the *locale* argument and its associated *category* will restore that part of the program's locale.

ERRORS

If a language name given through the *locale* argument does not identify a valid language name or the language is not available on the system (see *lang(5))* a null pointer is returned and the program's locale is not changed. The same behavior will occur when the call :

```
setlocale(LC_ALL, getenv("LANG"));
```

is made and any category related environment variable in the user's environment identifies an invalid language name or a language that is not available on the system.

If the *category* argument is not a defined category value a null pointer is returned and the program's locale is not changed.

Setlocale returns a string which reflects the current setting of that aspect of the NLS environment corresponding to the *category* argument. If this return string is used in a subsequent *setlocale* call and the *category* arguments of the two calls do not match, the locale remains unchanged and a null pointer is returned.

WARNINGS

The use of the *getenv*() function as the *locale* argument is not recommended. An example of this usage is :

```
setlocale(LC_ALL, getenv("LANG"));
```

Getenv will return a character string which may be a language name, an empty string or a null pointer depending on the setting of the user's LANG environment variable. Each of these values as the *locale* argument define a specific action to be taken by *setlocale.* Therefore the action taken by *setlocale* will depend upon the value returned from the *getenv* call. To ensure *setlocale* will set the program's locale based upon the setting of the user's environment variables the following usage is recommended :

```
setlocale(LC_ALL, "");
```

The value returned by *setlocale* points to a static area that will be overwritten with the next call to *setlocale.* It is recommended that these values be copied to another area if they are to be used after a subsequent *setlocale* call.

The structure which is returned through a call to *getlocale* will be overwritten with the next call to *getlocale*. It is recommended that these values be saved if they are to be used after a subsequent *getlocale* call.

EXAMPLES

To set a program's entire locale based on the language requirements specified via the user's environment variables :

```
setlocale(LC_ALL,"");
```

If, in the previous example, the user's environment variables were set as follows :

```
LANG="german"
LC_COLLATE="spanish@nofold"
LC_MONETARY=""
LC_TIME="american"
```

the LC_ALL, LC_CTYPE, LC_MONETARY, and LC_NUMERIC category items would be set to correspond to the "german" language definition, the LC_COLLATE category items would be set to correspond to the "spanish" language definition for unfolded collation (see *hpnls*(5)) and the LC_TIME category items would be set corresponding to the "american" language definition.

Using the same example, if the following call was made :

```
struct locale_data *locale_info=getlocale(LOCALE_STATUS);
```

the contents of *locale_info would be :

```
locale_info->LC_ALL_D="german"
locale_info->LC_COLLATE_D="spanish"
locale_info->LC_CTYPE_D="german"
locale_info->LC_MONETARY_D="german"
locale_info->LC_NUMERIC_D="german"
locale_info->LC_TIME_D="american"
```

Continuing with the same example, if the following call was made :

```
struct locale_data *modifier_info=getlocale(MODIFIER_STATUS);
```

the contents of *modifier_info would now be :

```
modifier_info->LC_ALL_D=""
modifier_info->LC_COLLATE_D="nofold"
modifier_info->LC_CTYPE_D=""
modifier_info->LC_MONETARY_D=""
modifier_info->LC_NUMERIC_D=""
modifier_info->LC_TIME_D=""
```

The calls :

```
setlocale(LC_ALL,"");
struct locale_data *error_info=getlocale(ERROR_STATUS);
```

with the following settings in the users environment :

```
LANG=german
LC_COLLATE=junk
```

where "junk" is an invalid language, would result in the contents of *error_info being :

```
_error_info->LC_ALL_D=""
_error_info->LC_COLLATE_D="junk"
_error_info->LC_CTYPE_D=""
_error_info->LC_MONETARY_D=""
_error_info->LC_NUMERIC_D=""
_error_info->LC_TIME_D=""
```

To set the date/time formats to French :

setlocale(LC_TIME, "french");

To set the collating sequence to the "C" locale :

setlocale(LC_COLLATE, "C");

To set monetary handling to the value of the user's LC_MONETARY environment variable :

setlocale(LC_MONETARY, "");

(Note that if the LC_MONETARY environment variable is not set or empty the value of the user's LANG environment variable will be used.)

To query a user's locale :
char *ch = setlocale(LC_ALL, NULL);

To restore the locale saved in the above example :
setlocale(LC_ALL, ch);

To query just that part of the user's locale pertaining to the LC_NUMERIC category :
char *ch = setlocale(LC_NUMERIC, NULL);

To restore the LC_NUMERIC category of the user's locale saved in the above example :
setlocale(LC_NUMERIC, ch);

AUTHOR

Setlocale was developed by HP.

SEE ALSO

nlsinfo(1), buildlang(1M), conv(3C), ctype(3C), ecvt(3C), langinfo(3C), multibyte(3C), nl_tools_16(3C), printf(3S), scanf(3S), strcoll(3C), strftime(3C), string(3C), strtod(3C), vprintf(3S), hpnls(5), environ(5), langinfo(5).

STANDARDS CONFORMANCE

setlocale: XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

sigemptyset, sigfillset, sigaddset, sigdelset, sigismember – initialize, manipulate, and test signal sets

SYNOPSIS

#include <signal.h>

int sigemptyset (set)
sigset_t *set;

int sigfillset (set)
sigset_t *set;

int sigaddset (set, signo)
sigset_t *set;
int signo;

int sigdelset (set, signo)
sigset_t *set;
int signo;

int sigismember (set, signo)
sigset_t *set;
int signo;

DESCRIPTION

Sigemptyset initializes the signal set pointed to by *set*, to exclude all signals supported by HP-UX.

Sigfillset initializes the signal set pointed to by *set*, to include all signals supported by HP-UX.

Applications must call either *sigemptyset* or *sigfillset* at least once for each object of type **sigset_t** before using that object for anything else, including cases where the object is returned from a function (for example, the *oset* argument to *sigprocmask*(2)).

Sigaddset adds the signal specified by *signo* to the signal set pointed to by *set*.

Sigdelset deletes the signal specified by *signo* from the signal set pointed to by *set*.

Sigismember tests whether the signal specified by *signo* is a member of the signal set pointed to by *set*.

RETURN VALUE

Upon successful completion, *sigismember* returns a value of **1** if the specified signal is a member of the specified set, or a value of **0** if it is not. The other functions return a value of **0** upon successful completion. For all of the above functions, if an error is detected, a value of **−1** is returned and **errno** is set to indicate the error.

ERRORS

Sigaddset, *sigdelset*, and *sigismember* fail if the following is true:

[EINVAL] The value of the *signo* argument is out of range. The reliable detection of this error is not guaranteed.

WARNINGS

The above functions do not detect a bad address passed in for the *set* argument. A segmentation fault is the most likely result.

AUTHOR

Sigfillset, *sigemptyset*, *sigaddset*, *sigdelset*, and *sigismember* were derived from the IEEE Standard POSIX 1003.1-1988.

SEE ALSO

sigaction(2), sigsuspend(2), sigpending(2), sigprocmask(2), signal(5).

STANDARDS CONFORMANCE

sigaddset: XPG3, POSIX.1, FIPS 151-1

sigdelset: XPG3, POSIX.1, FIPS 151-1

sigemptyset: XPG3, POSIX.1, FIPS 151-1

sigfillset: XPG3, POSIX.1, FIPS 151-1

sigismember: XPG3, POSIX.1, FIPS 151-1

NAME

sinh, cosh, tanh – hyperbolic functions

SYNOPSIS

#include <math.h>

double sinh (x)
double x;

double cosh (x)
double x;

double tanh (x)
double x;

DESCRIPTION

Sinh, cosh, and *tanh* return respectively the hyberbolic sine, cosine and tangent of their argument.

DEPENDENCIES

Series 800 (/lib/libm.a and ANSI C /lib/libM.a)

When x is ±INFINITY , *sinh* returns ±INFINITY respectively.

When x is ±INFINITY , *cosh* returns +INFINITY .

When x is ±INFINITY , *tanh* returns ±1.0 respectively.

ERRORS

Series 300

Sinh and *cosh* return HUGE_VAL (and *sinh* may return –HUGE_VAL for negative x) and set **errno** to **ERANGE** when the correct value would overflow.

Series 800 (/lib/libm.a and ANSI C /lib/libM.a)

Sinh and *cosh* return HUGE_VAL (and *sinh* may return –HUGE_VAL for negative x) and set **errno** to **ERANGE** when the correct value would overflow.

Sinh, cosh and *tanh* return NaN and set **errno** to **EDOM** when x is NaN.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

isinf(3M), isnan(3M), matherr(3M).

STANDARDS CONFORMANCE

sinh: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

cosh: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

tanh: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

sleep – suspend execution for interval

SYNOPSIS

unsigned int sleep (seconds)
unsigned int seconds;

DESCRIPTION

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

The routine is implemented by setting an alarm signal and pausing until it (or some other signal) occurs. The previous state of the alarm signal is saved and restored. The calling program may have set up an alarm signal before calling *sleep*. If the *sleep* time exceeds the time until such an 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. 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*.

Seconds must be less than 2^{32}.

SEE ALSO

alarm(2), pause(2), signal(5).

STANDARDS CONFORMANCE

sleep: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

sputl, sgetl – access long integer data in a machine-independent fashion

SYNOPSIS

void sputl (value, buffer)
long value;
char *buffer;

long sgetl (buffer)
char *buffer;

DESCRIPTION

Sputl takes the four bytes of the long integer *value* and places them in memory starting at the address pointed to by *buffer*. The ordering of the bytes is the same across all machines.

Sgetl retrieves the four bytes in memory starting at the address pointed to by *buffer* and returns the long integer value in the byte ordering of the host machine.

The combination of *sputl* and *sgetl* provides a machine-independent way of storing long numeric data in a file in binary form without conversion to characters.

A program which uses these functions must be loaded with the object-file access routine library **libld.a**.

STANDARDS CONFORMANCE

sputl: SVID2

sgetl: SVID2

NAME

ssignal, gsignal – software signals

SYNOPSIS

#include <signal.h>

int (*ssignal (sig, action))( )
int sig, (*action)( );

int gsignal (sig)
int sig;

DESCRIPTION

Ssignal and *gsignal* implement a software facility similar to *signal*(5). This facility is used by the Standard C Library to enable users to indicate the disposition of error conditions, and is also made available to users for their own purposes.

Software signals made available to users are associated with integers in the inclusive range 1 through 15. 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(5).

NOTES

There are some additional signals with numbers outside the range 1 through 15 which are used by the Standard C Library to indicate error conditions. Thus, some signal numbers outside the range 1 through 15 are legal, although their use may interfere with the operation of the Standard C Library.

STANDARDS CONFORMANCE

ssignal: SVID2, XPG2

gsignal: SVID2, XPG2

NAME

statfsdev, fstatfsdev – get file system statistics

SYNOPSIS

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

int statfsdev(path, buf)
char *path;
struct statfs *buf;

int fstatfsdev(fildes, buf)
int fildes;
struct statfs *buf;
```

DESCRIPTION

Statfsdev returns information about the file system on the file specified by *path.*

Buf is a pointer to a **statfs** structure into which information is placed concerning the file system. The contents of the structure pointed to by *buf* include the following members:

```
long    f_bavail;   /* free blocks available to non-superuser */
long    f_bfree;    /* free blocks */
long    f_blocks;   /* total blocks in file system */
long    f_bsize;    /* fundamental file system block size in bytes */
long    f_ffree;    /* free file nodes in file system */
long    f_files;    /* total file nodes in file system */
long    f_type;     /* type of info, zero for now */
fsid_t  f_fsid;     /* file system ID */
```

Fields that are undefined for a particular file system are set to −**1**.

Fstatfsdev returns the same information as above, but about the open file referred to by file descriptor *fildes*.

RETURN VALUE

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

ERRORS

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

[EACCES] Search permission is denied for a component of the path prefix.

[EAGAIN] The file exists, enforcement mode file/record locking is set, and there are outstanding record locks on the file.

[EFAULT] *Path* points to an invalid address.

[ELOOP] Too many symbolic links are encountered in translating the path name.

[EMFILE] The maximum number of file descriptors allowed are currently open.

[ENAMETOOLONG] The path name is too long.

[ENFILE] The system file table is full.

[ENOENT] The named file does not exist.

[ENOTDIR] A component of the path prefix is not a directory.

[ENXIO] The device specified by the named special file does not exist.

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

[EBADF] *Fildes* is not a valid open file descriptor.

[ESPIPE] *filedes* points to an invalid address.

Both *fstatfsdev* and *statfsdev* fail if one or more of the following is true:

[EAGAIN] Enforcement-mode record locking was set, and there was a blocking write lock.

[EDEADLK] A resource deadlock would occur as a result of this operation.

[EINTR] A system call was interrupted by a signal.

[EINVAL] The file specified by *path* or *filedes* does not contain a file system of any known type.

[ENOLOCK] The system lock table was full, so the read could not go to sleep until the blocking write lock was removed.

AUTHOR

Statfsdev and *fstatfsdev* were developed by HP.

FILES

/usr/include/sys/mount.h

SEE ALSO

bdf(1M), df(1M), stat(2), statfs(2).

NAME

stdio – standard buffered input/output stream file package

SYNOPSIS

#include <stdio.h>

FILE *stdin, *stdout, *stderr;

DESCRIPTION

The functions described in the entries of sub-class (3S) of this manual constitute an efficient, user-level I/O buffering scheme. The routines *getc*(3S) and *putc*(3S) handle characters quickly. The routines *fgetc, fgets, fprintf, fputc, fputs, fread, fscanf, fwrite, getchar, gets, getw, printf, putchar, puts, putw*, and *scanf* all use or act as if they use *getc* and *putc*; they can be freely intermixed.

A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type **FILE**. *Fopen*(3S) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. The Section (3S) library routines operate on this stream.

At program startup, three streams, *standard input, standard output* and *standard error,* are predefined and do not need not be explicitly opened. When opened, the standard input and standard output streams are fully buffered if the output refers to a file and line-buffered if the output refers to a terminal. The standard error output stream is be default unbuffered. These three streams have the following constant pointers declard in the <stdio.h> header file :

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

A constant **NULL** (0) designates a nonexistent pointer.

An integer-constant **EOF** (−1) is returned upon end-of-file or error by most integer functions that deal with streams (see the individual descriptions for details).

An integer constant **BUFSIZ** specifies the size of the buffers used by the particular implementation (see *setbuf*(3S)).

Any program that uses this package must include the header file of pertinent macro definitions, as follows:

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

The functions and constants mentioned in the entries of sub-class (3S) of this manual are declared in that header file and need no further declaration.

A constant _NFILE defines the maximum number of open files allowed per process.

SEE ALSO

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

DIAGNOSTICS

Invalid *stream* pointers will usually cause grave disorder, possibly including program termination. Individual function descriptions describe the possible error conditions.

STANDARDS CONFORMANCE

stdio: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

stderr: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

stdin: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C
stdout: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

ftok – standard interprocess communication package

SYNOPSIS

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

key_t ftok(path, id)
char *path;
char id;

DESCRIPTION

All interprocess communication facilities require the user to supply a key to be used by the *msgget*(2), *semget*(2), and *shmget*(2) system calls to obtain interprocess communication identifiers. One suggested method for forming a key is to use the *ftok* subroutine described below. Another way to compose keys is to include the project ID in the most significant byte and to use the remaining portion as a sequence number. There are many other ways to form keys, but it is necessary for each system to define standards for forming them. If some standard is not adhered to, it will be possible for unrelated processes to unintentionally interfere with each other's operation. Therefore, it is strongly suggested that the most significant byte of a key in some sense refer to a project so that keys do not conflict across a given system.

Ftok returns a key based on *path* and *id* that is usable in subsequent *msgget*, *semget*, and *shmget* system calls. *Path* must be the path name of an existing file that is accessible to the process. *Id* is a character which uniquely identifies a project. Note 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*.

DIAGNOSTICS

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

EXAMPLES

The following call to ftok() returns a key associated with the file *myfile* and id 'A':

```
        key_t mykey;

                mykey = ftok ("myfile", 'A');
```

WARNINGS

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.

In the HP Clustered environment, *ftok* may return a different key (using the same file name) when executed on different members of the cluster if any component of the file path name is a CDF.

SEE ALSO

intro(2), msgget(2), semget(2), shmget(2), cdf(4).

NAME

strftime – convert date and time to string

SYNOPSIS

#include <time.h>

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

DESCRIPTION

The *strftime* function converts the contents of a **tm** structure (see *ctime*(3C)) to a formatted date and time string.

The *strftime* function places 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. A directive consists of a **%** character, an optional field width and precision specification, and a terminating character that determines the directive's behavior. All ordinary characters (including the terminating null character) are copied unchanged into the array. No more than *maxsize* characters are placed into the array. Each directive is replaced by the appropriate characters as described in the following list. The appropriate characters are determined by the program's locale, by the values contained in the structure pointed to by *timeptr*, and by the TZ environment variable (see External Influences below).

Directives

The following directives, shown without the optional field width and precision specification, are replaced by the indicated characters:

%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
%d	day of the month as a decimal number [01,31]
%E	locale's combined Emperor/Era name and year
%H	hour (24-hour clock) as a decimal number [00,23]
%I	hour (12-hour clock) as a decimal number [01,12]
%j	day of the year as a decimal number [001,366]
%m	month as a decimal number [01,12]
%M	minute as a decimal number [00,59]
%n	new-line character
%N	locale's Emperor/Era name
%o	locale's Emperor/Era year
%p	locale's equivalent of either AM or PM
%S	second as a decimal number [00,61]
%t	tab character
%U	week number of the year (the first Sunday as the first day of week 1) as a decimal number [00,53]
%w	weekday as a decimal number [0(Sunday),6]
%W	week number of the year (the first Monday as the first day of week 1) as a decimal number [00,53]

%x	locale's appropriate date representation
%X	locale's appropriate time representation
%y	year without century as a decimal number [00,99]
%Y	year with century as a decimal number
%Z	time zone name (or by no characters if no time zone exists)
%%	%

The following directives are provided for backward compatibility with the directives supported by *date*(1) and the *ctime*(3C) functions. It is recommended that the directives above be used in preference to those below.

%D	date in usual US format (%m/%d/%y) (use **%x** instead)
%F	locale's full month name (use **%B** instead)
%h	locale's abbreviated month name (use **%b** instead)
%r	time in 12-hour US format (%I:%M:%S [AM|PM]) (use **%X** instead)
%T	time in 24-hour US format (%H:%M:%S) (use **%X** instead)
%z	time zone name (or by no characters if no time zone exists) (use **%Z** instead)

If a directive is not one of the above, the behavior is undefined.

Field Width and Precision

An optional field width and precision specification can immediately follow the initial **%** of a directive in the following order:

[–|**0**]*w* the decimal digit string *w* specifies a minimum field width in which the result of the conversion is right- or left-justified. It is right-justified (with space padding) by default. If the optional flag '–' is specified, it is left-justified with space padding on the right. If the optional flag '**0**' is specified, it is right-justified and padded with zeros on the left.

.*p* the decimal digit string *p* specifies the minimum number of digits to appear for the **d, H, I, j, m, M, o, S, U, w, W, y** and **Y** directives, and the maximum number of characters to be used from the **a, A, b, B, c, D,** E, **F, h, n, N, p, r, t, T, x, X, z, Z** and **%** directives. In the first case, if a directive supplies fewer digits than specified by the precision, it will be expanded with leading zeros. In the second case, if a directive supplies more characters than specified by the precision, excess characters will truncated on the right.

If no field width or precision is specified for a **d, H, I, m, M, S, U, W, y** or **j** directive, a default of ".2" is used for all but **j** for which ".3" is used.

EXTERNAL INFLUENCES

Locale

The LC_TIME category determines the characters to be substituted for those directives described above as being from the locale.

The LC_CTYPE category determines the interpretation of the bytes within *format* as single and/or multi-byte characters.

The LC_NUMERIC category determines the characters used to form numbers for those directives that produce numbers in the output. If ALT_DIGITS (see *langinfo*(5)) is defined for the locale, the characters so specified are used in place of the default ASCII characters.

Environment Variables

TZ determines the time zone name substituted for the **%Z** and **%z** directives. The time zone name is determined by calling the function *tzset* which sets the external variable *tzname* (see *ctime*(3C)).

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

If the total number of resulting characters including the terminating null character is not more than *maxsize*, *strftime* returns 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.

EXAMPLES

If the *timeptr* argument contains the following values:

```
timeptr→tm_sec = 4;
timeptr→tm_min = 9;
timeptr→tm_hour = 15;
timeptr→tm_mday = 4;
timeptr→tm_mon = 6;
timeptr→tm_year = 88;
timeptr→tm_wday = 1;
timeptr→tm_yday = 185;
timeptr→tm_isdst = 1;
```

the following combinations of the LC_TIME category and format strings produce the indicated output:

LC_TIME	format string	output
american	%x	Mon, Jul 4, 1988
german	%x	Mo., 4. Juli 1988
american	%X	03:09:04 PM
french	%X	15h09 04
any†	%H:%M:%S	15:09:04
any†	%.1H:%.1M:%.1S	15:9:4
any†	%2.1H:%-3M:%03.1S	15:9 :004

† The directives used in these examples are not affected by the LC_TIME category of the locale.

WARNINGS

If the arguments *s* and *format* are defined such that they overlap, the behavior is undefined.

The function *tzset* is called upon every invocation of *strftime* (whether or not the time zone name is copied to the output array).

The range of values for %S ([0,61]) extends to 61 to allow for the occasional one or two leap seconds. However, the system does not accumulate leap seconds and the **tm** structure generated by the functions *localtime* and *gmtime* (see *ctime*(3C)) never reflects any leap seconds.

Results are undefined if values contained in the structure pointed to by *timeptr* exceed the ranges defined for the **tm** structure (see *ctime*(3C)) or are not consistent. For example, the **tm_yday** element set to 0, indicating the first day of January, while the **tm_mon** element is set to 11, indicating a day in December).

AUTHOR

Strftime was developed by HP.

SEE ALSO

date(1), ctime(3C), setlocale(3C), environ(5), langinfo(5), hpnls(5).

STANDARDS CONFORMANCE

strftime: XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strdup, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strstr, strtok, strcoll, strxfrm, nl_strcmp, nl_strncmp – character string operations

SYNOPSIS

```
#include <string.h>

char *strcat (s1, s2)
char *s1;
const char *s2;

char *strncat (s1, s2, n)
char *s1;
const char *s2;
size_t n;

int strcmp (s1, s2)
const char *s1, *s2;

int strncmp (s1, s2, n)
const char *s1, *s2;
size_t n;

char *strcpy (s1, s2)
char *s1;
const char *s2;

char *strncpy (s1, s2, n)
char *s1;
const char *s2;
size_t n;

char *strdup (s)
const char *s;

size_t strlen (s)
const char *s;

char *strchr (s, c)
const char *s;
int c;

char *strrchr (s, c)
const char *s;
int c;

char *strpbrk (s1, s2)
const char *s1, *s2;

size_t strspn (s1, s2)
const char *s1, *s2;

size_t strcspn (s1, s2)
const char *s1, *s2;

char *strstr (s1, s2)
const char *s1, *s2;

char *strtok (s1, s2)
char *s1;
const char *s2;
```

```
int strcoll (s1, s2)
const char *s1, *s2;

size_t strxfrm (s1, s2, n)
char *s1;
const char *s2;
size_t n;

int nl_strcmp (s1, s2)
const char *s1, *s2;

int nl_strncmp (s1, s2, n)
const char *s1, *s2;
size_t n;
```

DESCRIPTION

The arguments *s1*, *s2*, and *s* point to strings (arrays of characters terminated by a null byte).

Definitions for all these functions, the type **size_t**, and the constant **NULL** are provided in the **<string.h>** header.

Strcat appends a copy of string *s2* to the end of string *s1*. *Strncat* appends a maximum of *n* characters. It copies fewer if *s2* is shorter than *n* characters. Each returns a pointer to the null-terminated result (the value of *s1*).

Strcmp compares its arguments and returns an integer less than, equal to, or greater than zero, depending on whether *s1* is lexicographically less than, equal to, or greater than *s2*. The comparison of corresponding characters is done as if the type of the characters were **unsigned char**. Null pointer values for *s1* and *s2* are treated the same as pointers to empty strings. *Strncmp* makes the same comparison but examines a maximum of *n* characters (*n* less than or equal to zero yields equality).

Strcpy copies string *s2* to *s1*, stopping after the null byte has been copied. *Strncpy* copies exactly *n* characters, truncating *s2* or adding null bytes to *s1* if necessary, until *n* characters in all have been written. The result will not be null-terminated if the length of *s2* is *n* or more. Each function returns *s1*. Note that *strncpy* should not be used to copy *n* bytes of an arbitrary structure. If that structure contains a null byte anywhere, *strncpy* will copy fewer than *n* bytes from the source to the destination, and fill the remainder with null bytes. Use the *memcpy* function (described on *memory*(3C)) to copy arbitrary binary data.

Strdup returns a pointer to a new string which is a duplicate of the string to which *s1* points. The space for the new string is obtained using the *malloc*(3C) or *malloc*(3X) function (depending on which is linked with the program).

Strlen returns the number of characters in *s*, not including the terminating null byte.

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

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

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

Strstr returns a pointer to the first occurrence of string *s2* in string *s1*, or a NULL pointer if *s2* does not occur in the string. If *s2* points to a string of zero length, *strstr* returns *s1*.

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 a non-null pointer *s1* specified) returns a pointer to the first character of the first token, and will have written a null byte into *s1* immediately following the returned token. The function keeps track of its position in the string *s1* between separate calls, so that subsequent calls made with the first argument a null pointer will work through the string 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.

Strcoll returns an integer greater than, equal to, or less than zero, according as the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2*. The comparison is based on strings interpreted as appropriate to the program's locale (see Locale below). In the "C" locale *strcoll* works like *strcmp*. *Nl_strcmp* is provided for historical reasons only and is equivalent to *strcoll*. *Nl_strncmp*, also provided only for historical reasons, makes the same comparisons as *strcoll*, but looks at a maximum of *n* characters (*n* less than or equal to zero yields equality).

Strxfrm transforms the string pointed to by *s2* and places the resulting string into the array pointed to by *s1*. The transformation is such that if the *strcmp* function is applied to two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the *strcoll* function applied to the same two original strings. No more than *n* bytes are placed into the resulting string including the terminating null character. If the transformed string fits in no more than *n* bytes, the length of the resulting string is returned (not including the terminating null character). Otherwise the return value is the number of bytes that the *s1* string would occupy (not including the terminating null character), and the contents of the array are indeterminate.

Strcoll has better performance with respect to *strxfrm* in cases where a given string is compared to other strings only a few times, or where the strings to be compared are long but a difference in the strings that determines their relative ordering usually comes among the first few characters. *Strxfrm* offers better performance in, for example, a sorting routine where a number of strings are each transformed just once and the transformed versions are compared against each other many times.

EXTERNAL INFLUENCES

Locale

The LC_CTYPE category determines the interpretation of the bytes within the string arguments to the *strcoll*, *strxfrm*, *nl_strcmp* and *nl_strncmp* functions as single and/or multi-byte characters.

The LC_COLLATE category determines the collation ordering used by the *strcoll*, *strxfrm*, *nl_strcmp* and *nl_strncmp* functions. See *hpnls*(5) for a description of supported collation features. See *nlsinfo*(1) to view the collation used for a particular locale.

International Code Set Support

Single- and multi-byte character code sets are supported for the *strcoll*, *strxfrm*, *nl_strcmp* and *nl_strncmp* functions. All other functions support only single-byte character code sets.

WARNINGS

The functions *strcat*, *strncat*, *strcpy*, *strncpy*, and *strtok* alter the contents of the array to which *s1* points. They do not check for overflow of the array.

Null pointers for destination strings cause undefined behavior.

Character movement is performed differently in different implementations, so moves involving overlapping source and destination strings may yield surprises.

The transformed string produced by *strxfrm* for a language using an 8-bit code set will usually be at least twice as large as the original string and may be as much four times as large (ordinary characters occupy two bytes each in the transformed string, 1-to-2 characters four bytes, 2-to-1 characters two bytes per original pair, and don't-care characters no bytes). Each character of a multi-byte code set (Asian languages) will occupy three bytes in the transformed string.

For the *strcoll*, *strxfrm*, *nl_strcmp* and *nl_strncmp* functions, the results are undefined if the languages specified by the LC_COLLATE and LC_CTYPE categories use different code sets.

AUTHOR

String was developed by AT&T and HP.

SEE ALSO

nlsinfo(1), malloc(3C), malloc(3X), memory(3C), setlocale(3C), hpnls(5).

STANDARDS CONFORMANCE

nl_strcmp: XPG2

nl_strncmp: XPG2

strcat: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

strchr: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

strcmp: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

strcoll: XPG3, ANSI C

strcpy: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

strcspn: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

strdup: SVID2

strlen: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

strncat: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

strncmp: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

strncpy: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

strpbrk: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

strrchr: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

strspn: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

strstr: XPG3, POSIX.1, FIPS 151-1, ANSI C

strtok: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

strxfrm: XPG3, ANSI C

NAME

strord – convert string data order

SYNOPSIS

#include <nl_types.h>

char *strord (s1, s2, m)
char *s1, *s2;
nl_mode m;

DESCRIPTION

The text orientation (mode) of a file can be right-to-left (non-Latin) or left-to-right (Latin). This text orientation can affect the way data is arranged in the file. The data arrangements that result are called screen order and keyboard order (see *hpnls*(5) for more details).

The *strord* routine converts the order of characters in *s2* from screen to keyboard order or vice versa and places the result in *s1*. The arguments *s1* and *s2* point to strings (arrays of characters terminated by a null character). *Strord* returns *s1*.

Strord performs the conversion based on mode information indicated by the argument *m*. The argument *m* is of type *nl_mode* found in the header file **<nl_types.h>**. The mode argument can have two possible values: NL_LATIN and NL_NONLATIN.

If the mode argument is NL_LATIN, the text orientation is left-to-right and all non-Latin sub-strings are reversed. Non-Latin sub-strings are any number of contiguous right-to-left language characters. Non-Latin sub-strings are delimited by ASCII characters.

Similarly, if the mode argument is NL_NONLATIN, the text orientation is right-to-left and all Latin sub-strings are reversed. Latin sub-strings are any number of contiguous printable ASCII characters. Latin sub-strings are delimited by right-to-left language characters and ASCII control codes.

Some right-to-left languages have a duplicate set of digits called alternative numbers. Alternative numbers always have a left-to-right orientation.

WARNINGS

Strord does not check for overflow of the array pointed to by *s1*.

AUTHOR

Strord was developed by HP.

SEE ALSO

nl_init(3C), hpnls(5), environ(5), forder(1), nljust(1).

EXTERNAL INFLUENCES

Locale

The LC_NUMERIC category determines whether a right-to-left language has alternative numbers.

International Code Set Support

Single-byte character code sets are supported.

NAME

strtoacl, strtoaclpatt – convert exact or pattern string form to access control list (ACL) structure

SYNOPSIS

```
#include <acllib.h>

int strtoacl (string, nentries, maxentries, acl, fuid, fgid)
char *string;
int nentries;
int maxentries;
struct acl_entry acl[];
int fuid, fgid;

int strtoaclpatt (string, maxentries, acl)
char *string;
int maxentries;
struct acl_entry_patt acl[];

extern char *aclentrystart[];
```

Remarks:

To ensure continued conformance with emerging industry standards, features described in this manual entry are likely to change in a future release.

DESCRIPTION

Strtoacl converts an access control list from exact symbolic (string) representation to structure form. It parses the input string and verifies its validity. Optionally it applies the entries in the string as a series of changes to an existing ACL.

Strtoaclpatt converts an access control list pattern from symbolic (string) representation to structure form. It parses the input string and verifies its validity.

The external array *aclentrystart*[], only valid until the next call of either routine, is useful for error reporting. See ERRORS below.

The "operator" and "short" symbolic forms of ACLs and ACL patterns (described in *acl*(5)) are acceptable as input strings. If the first non-whitespace character in *string* is "(", the ACL or ACL pattern in *string* must be in short form. Otherwise operator form is assumed.

Strtoacl takes a pointer to the string to be converted, and a pointer to the first element of an array of ACL entries (*acl*[]) initially containing the indicated number (*nentries*) of valid entries (zero or more). This array can grow to the indicated number of entries (*maxentries*). *Strtoacl* also takes file user ID (*fuid*) and group ID (*fgid*) values to substitute for @ symbols in *string* and returns the resulting number of entries in *acl*[].

Redundant entries (identical user ID and group ID values after processing @ symbols) are combined, so that *acl*[] contains unique entries in the order encountered. If a new entry is mentioned, it is added to the end of the *acl* array.

Strtoaclpatt

Strtoaclpatt differs from *strtoacl* because it processes an ACL pattern instead of an ACL. Since modification of an existing initial ACL is not useful, it is not supported.

Entries with matching user and group ID values are not combined. Each entry input yields one entry in the returned array.

The @ symbol for user and group IDs (see *acl*(5)) is converted to special values (ACL_FILEOWNER or ACL_FILEGROUP, respectively, defined in **<acllib.h>**), not to specific user or group names provided by the caller. Thus, *strtoaclpatt* need not be called to reparse the ACL pattern for each file, but the caller must handle the special values when comparing an ACL pattern to an ACL.

Wildcard user names, group names, and mode values are supported, as are absent mode parts; see *acl*(5).

Strtoaclpatt returns a different structure than *strtoacl*. The *acl_entry_patt* structure contains *onmode* and *offmode* masks rather than a single *mode* value.

In operator form input, operators have a different effect on *strtoaclpatt*:

= Sets bits in both the *onmode* and *offmode* fields appropriately, replacing existing bits in the entry, including any set by earlier operators.

+ Sets bits in *onmode* and clears the same bits in *offmode*.

− Sets bits in *offmode* and clears the same bits in *onmode*.

In short form input, the mode is treated like the = operator in operator form.

For both routines, a non-specific user or group ID of % is converted to ACL_NSUSER or ACL_NSGROUP, respectively. For *strtoaclpatt* only, a wildcard user or group ID of * is converted to ACL_ANYUSER or ACL_ANYGROUP, respectively. The values are defined in **<acllib.h>**.

Entries can appear in *string* in any order. *String* can contain redundant entries, and in operator form only, redundant + and − operators for ACL entry mode modifications (in exact form) or mode bit inclusions/exclusions (in patterns). Entries or modifications are applied left to right.

Suggested Use

To build a new ACL (ACL pattern) array using *strtoacl* (*strtoaclpatt*), define *acl*[] with as many entries as desired. Pass it to *strtoacl* (*strtoaclpatt*) with *nentries* set to zero (*strtoacl* only) and *maxentries* set to the number of elements in *acl*[].

To have *strtoacl* modify a file's existing ACL, define *acl*[] with the maximum possible number of entries (NACLENTRIES; see **<sys/acl.h>**). Call *getacl*(2) to read the file's ACL and *stat*(2) to get the file's owner and group IDs. Then pass the current number of entries, the current ACL, and the ID values to *strtoacl* with *maxentries* set to NACLENTRIES.

If *strtoacl* succeeds, the resulting ACL can be passed safely to *setacl*(2) because all redundancies (if any) have been resolved. However, note that since neither *strtoacl* nor *strtoaclpatt* validate user and group ID values, if the values are not acceptable to the system, *setacl*(2) will fail.

Performance Trick

Normally *strtoacl* replaces user and group names of @ with specific user and group ID values, and also combines redundant entries. Therefore, calling *stat*(2) and *strtoacl* for each of a series of files to which an ACL is being applied is simplest, although time consuming.

If *string* contains no @ symbol, or if the caller merely wants to compare one ACL against another (and will handle the special case itself), it is sufficient to call *strtoacl* once, and pointless to call *stat* for each file. To determine this, call *strtoacl* the first time with *fuid* set to ACL_FILEOWNER and *fgid* set to ACL_FILEGROUP. Repeated calls with file-specific *fuid* and *fgid* values are needed only if the special values of *fuid* and *fgid* appear in *acl*[] and the caller needs an exact ACL to set on each file; see EXAMPLES below.

If @ appears in *string* and *acl*[] will be used later for a call to *setacl*(2), it is necessary to call *strtoacl* again to reparse the ACL string for each file. It is possible that not all redundant entries were combined the first time because the @ names were not resolved to specific IDs. This also complicates comparisons between two ACLs. Furthermore, the caller cannot do the combining later because operator information from operator form input might be lost.

RETURN VALUE

If *strtoacl* (*strtoaclpatt*) succeeds, it returns the number of entries in the resulting ACL (ACL pattern), always equal to or greater than *nentries* (zero).

Strtoaclpatt also sets values in global array *aclentrystart*[] to point to the start of each pattern entry it parsed in *string*, in some cases including leading or trailing whitespace. It only sets a number of pointers equal to its return value plus one (never more than NACLENTRIES + 1). The last valid element points to the null character at the end of *string*. After calling *strtoaclpatt*, an entry pattern's corresponding input string may be used by the caller for error reporting by (temporarily) putting a null at the start of the next entry pattern in *string*.

ERRORS

If an error occurs, both routines return a negative value and the content of *acl* is undefined (was probably altered). To help with error reporting in this case, *aclentrystart*[0] and *aclentrystart*[1] are set to point to the start of the current and next entries, respectively, being parsed when the error occurred. If the current entry does not start with "(", *aclentrystart*[1] points to the next null character or comma at or after *aclentrystart*[0]. Otherwise, it points to the next null, or to the character following the next ")".

The following values are returned in case of error:

−1 Syntax error: entry doesn't start with "(" as expected in short form.

−2 Syntax error: entry doesn't end with ")" as expected in short form.

−3 Syntax error: user name is not terminated by a dot.

−4 (*strtoacl* only) Syntax error: group name is not terminated by an operator in operator form input or a comma in short form input.

−5 Syntax error: user name is null.

−6 Syntax error: group name is null.

−7 Invalid user name (not found in **/etc/passwd** file and not a valid number).

−8 Invalid group name (not found in **/etc/group** file and not a valid number).

−9 Syntax error: invalid mode character, other than **0..7**, **r**, **w**, **x**, **−** (allowed in short form only), * (allowed in patterns only), **,** (to end an entry in operator form), or **)** (to end an entry in short form). Or, **0..7** or * is followed by other mode characters.

−10 The resulting ACL would have more than *maxentries* entries.

EXAMPLES

The following code fragment converts an ACL from a string to an array of entries using an *fuid* of 103 for the file's owner and *fgid* of 45 for the file's group.

```
#include <acllib.h>

int nentries;
struct acl_entry acl [NACLENTRIES];

if ((nentries = strtoacl (string, 0, NACLENTRIES, acl, 103, 45)) < 0)
        error (...);
```

The following code gets the ACL, *fuid*, and *fgid* for file "../myfile", modifies the ACL using a description string, and changes the ACL on file "../myfile2" to be the new version.

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

struct stat statbuf;
int nentries;
struct acl_entry acl [NACLENTRIES];

if (stat ("../myfile", & statbuf) < 0)
        error (...);
```

```
if ((nentries = getacl ("../myfile", NACLENTRIES, acl)) < 0)
        error (...);

if ((nentries = strtoacl (string, nentries, NACLENTRIES, acl,
        statbuf.st_uid, statbuf.st_gid)) < 0)
{
        error (...);
}

if (setacl ("../myfile2", nentries, acl) < 0)
        error (...);
```

The following code fragment calls *strtoacl* with special values of *fuid* and *fgid*, then checks to see if they show up in *acl*[].

```
#include <acllib.h>

int  perfile = 0;  /* need to stat() and reparse per file? */
int  entry;

if ((nentries = strtoacl (string, 0, NACLENTRIES, acl,
        ACL_FILEOWNER, ACL_FILEGROUP)) < 0)
{
        error (...);
}

for (entry = 0; entry < nentries; entry++)
{
        if ((acl [entry] . uid == ACL_FILEOWNER)
         || (acl [entry] . gid == ACL_FILEGROUP))
        {
                perfile = 1;
                break;
        }
}
```

The following code fragment converts an ACL pattern from a string to an array of pattern entries.

```
#include <acllib.h>

int nentries;
struct acl_entry_patt acl [NACLENTRIES];

if ((nentries = strtoaclpatt (string, NACLENTRIES, acl)) < 0)
        error (...);
```

The following code fragment inside a "for" loop checks an entry pattern (p*, *onmask*, and *offmask* variable names) against an entry in a file's ACL (a* variable names) using the file's user and group IDs (f* variable names).

```
include <unistd.h>
if (((puid == ACL_FILEOWNER) && (fuid != auid))
 || ((puid != ACL_ANYUSER)   && (puid != auid)))
{
        continue;
}

if (((pgid == ACL_FILEGROUP) && (fgid != agid))
 || ((pgid != ACL_ANYGROUP)  && (pgid != agid)))
{
```

```
                continue;
        }
        if (((( amode) & MODEMASK & onmask ) != onmask)
         || (((~ amode) & MODEMASK & offmask) != offmask))
        {
                continue;
        }
```

AUTHOR

Strtoacl and *strtoaclpatt* were developed by HP.

FILES

/etc/passwd
/etc/group

SEE ALSO

getacl(2), setacl(2), acltostr(3C), cpacl(3C), chownacl(3C), setaclentry(3C), acl(5).

NAME

strtod, atof, nl_strtod, nl_atof – convert string to double-precision number

SYNOPSIS

```
#include <stdlib.h>

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

double atof (str)
char *str;

double nl_strtod (str, ptr, langid)
char *str, **ptr;
int langid;

double nl_atof (str, langid)
char *str;
inl langid;
```

DESCRIPTION

Strtod returns, as a double-precision floating-point number, the value represented by the character string pointed to by *str*. The string is scanned (leading white-space characters as defined by *isspace* in *ctype*(3C) are ignored) up to the first unrecognized character. If no conversion can take place, zero is returned.

Strtod recognizes characters in the following sequence:

1. An optional string of "white-space" characters which are ignored,
2. An optional sign,
3. A string of digits optionally containing a radix character,
4. An optional **e** or **E** followed by an optional sign or space, followed by an integer.

The radix character is determined by the loaded NLS environment (see *setlocale*(3C)). If *setlocale* has not been called successfully, the default NLS environment, "C", is used (see *lang*(5)). The default environment specifies a period (.) as the radix character.

If the value of *ptr* is not (char **)NULL, the variable to which it points is set to point at the character after the last number, if any, that was recognized. If no number can be formed, **ptr* is set to *str*, and zero is returned.

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

Nl_strtod and *nl_atof* are similar to the above routines, but first call *langinit* (see *nl_init*(3C)) to load the NLS environment specified by *langid*.

DIAGNOSTICS

If the correct value would cause overflow, ±**HUGE_VAL** 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**.

WARNINGS

Nl_strtod and *nl_atof* are provided for historical reasons only. Their use is not recommended. Use *strtod* and *atof* instead.

EXTERNAL INFLUENCES

Locale

The LC_NUMERIC category determines the value of the radix character within the currently loaded NLS environment.

AUTHOR

Strtod was developed by AT&T and HP.

SEE ALSO

ctype(3C), setlocale(3C), scanf(3S), strtol(3C), hpnls(5), lang(5).

STANDARDS CONFORMANCE

strtod: SVID2, XPG2, XPG3, ANSI C

atof: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

strtol, atol, atoi, strtoul – convert string to integer

SYNOPSIS

#include <stdlib.h>

long strtol (str, ptr, base)
char *str, **ptr;
int base;

long atol (str)
char *str;

int atoi (str)
char *str;

unsigned long strtoul (str, ptr, base)
char *str, **ptr;
int base;

DESCRIPTION

Strtol(*strtoul*) converts the character string pointed to by *str* to **long int (unsigned long int)** representation. 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 no conversion can take place, zero is returned.

If *base* is greater than or equal to 2 and less than or equal to 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; a leading "0x" or "0X" hexadecimal conversion. Otherwise, decimal conversion is used.

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, the location pointed to by *ptr* is set to *str*, and zero is returned.

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

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

RETURN VALUE

Upon successful completion, all functions return the converted value, if any. If the correct value would cause overflow, *strtol* returns **LONG_MAX** or **LONG_MIN** (according to the sign of the value), and sets **errno** to **ERANGE**; *strtoul* returns **ULONG_MAX** and sets **errno** to **ERANGE**. Overflow conditions are ignored by *atol* and *atoi*.

For all other errors, zero is returned and *errno* is set to indicate the error.

ERRORS

Strtol and *strtoul* fail and **errno** is set if one of the following conditions is true:

[ERANGE] The value to be returned would have caused overflow.

[EINVAL] The value of *base* is not supported.

SEE ALSO

ctype(3C), strtod(3C), scanf(3S).

STANDARDS CONFORMANCE

strtol: SVID2, XPG2, XPG3, ANSI C

atoi: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C
atol: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C
strtoul: ANSI C

NAME

strtold – convert string to long double-precision number

SYNOPSIS

#include <stdlib.h>

long_double strtold (str, ptr)
char *str, **ptr;

DESCRIPTION

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

strtold 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 radix character, then an optional **e** or **E** followed by an optional sign or space, followed by an integer. The radix character is determined by the loaded NLS environment (see *nl_init*(3C)). If *nl_init* has not been called successfully, the default NLS environment, "C" (see *langid*(5)), is used. The default environment specifies a period (.) as the radix character.

If the value of *ptr* is not (char **)NULL, the variable to which it points is set to point at the character after the last number, if any, that was recognized. If no number can be formed, **ptr* is set to *str*, and zero is returned.

DIAGNOSTICS

If the correct value would cause overflow, ±**_MAXLDBL** 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**.

AUTHOR

strtold was developed by HP.

SEE ALSO

ctype(3C), nl_init(3C), scanf(3S), hpnls(5), langid(5).

EXTERNAL INFLUENCES

International Code Set Support

Single-byte character code sets are supported.

NAME

swab – swap bytes

SYNOPSIS

void swab (from, to, nbytes)
char *from, *to;
int nbytes;

DESCRIPTION

Swab copies *nbytes* bytes pointed to by *from* to the array pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between byte-swapped and non-byte-swapped machines. *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.

STANDARDS CONFORMANCE

swab: SVID2, XPG2, XPG3

NAME

syslog, openlog, closelog, setlogmask – control system log

SYNOPSIS

#include <syslog.h>

syslog(priority, message, parameters, ...)
int priority;
char *message;

openlog(ident, logopt, facility)
char *ident;
int logopt, facility;

closelog()

setlogmask(maskpri)

DESCRIPTION

Syslog writes a message onto the system log maintained by *syslogd*(1M). The message is tagged with *priority*. The *message* is similar to a *printf*(3S) format string except that **%m** is replaced by the error message associated with the current value of *errno*. A trailing newline is added if needed. This message is 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 *level* and a *facility*. 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.

The facility describes the part of the system generating the message:

LOG_KERN	Messages generated by the kernel. These cannot be generated by any user processes.
LOG_USER	Messages generated by random user processes. This is the default facility identifier if none is specified.
LOG_MAIL	The mail system.
LOG_DAEMON	System daemons, such as *ftpd*(1M), *rwhod*(1M), etc.
LOG_AUTH	The authorization system: *login*(1), *su*(1), *getty*(1M), etc.
LOG_LPR	The line printer spooling system: *lp*(1), *lpsched*(1M), etc.
LOG_LOCAL0	Reserved for local use. Similarly for LOG_LOCAL1 through LOG_LOCAL7.

If *syslog* cannot pass the message to *syslogd*(1M), it attempts 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 precedes every message. *Logopt* is a mask of bits indicating logging options. The values for *logopt* are:

LOG_PID
: Log the process ID with each message; useful for identifying instantiations of daemons.

LOG_CONS
: Force writing messages to the console if unable to send it to *syslogd*(1M). This option is safe to use in daemon processes that have no controlling terminal, because *syslog* forks before opening the console.

LOG_NDELAY
: Open the connection to *syslogd*(1M) immediately. Normally, the open is delayed until the first message is logged. This is useful for programs that need to manage the order in which file descriptors are allocated.

LOG_NOWAIT
: Do not wait for children forked to log messages on the console. This option should be used by processes that enable notification of child termination via SIGCLD, 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 written subsequently by *syslog* with no explicit facility encoded.

Closelog closes the log file.

Setlogmask sets the log priority mask to *maskpri* and returns the previous mask. Calls to *syslog* with a priority not set in *maskpri* are rejected. The mask for an individual priority *pri* is calculated by the macro LOG_MASK(*pri*); the mask for all priorities up to and including *toppri* is given by the macro LOG_UPTO(*toppri*). The default allows all priorities to be logged.

EXAMPLES

This call to *syslog* logs a message regarding a corrupted *who* database:

```
syslog(LOG_ALERT, "who: internal error 23");
```

This example shows the use of *openlog* to set up special formatting for the *ftp* daemon:

```
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");
```

WARNINGS

A call to *syslog*(3C) has no effect if the syslog daemon (*syslogd*(1M)) is not running on the system.

AUTHOR

Syslog was developed by the University of California, Berkeley.

SEE ALSO

logger(1), syslogd(1M).

NAME

system – issue a shell command

SYNOPSIS

#include <sys/wait.h>
#include <stdlib.h>

int system (string)
const char *string;

DESCRIPTION

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

If the *string* is a null pointer, *system* returns 1. Otherwise, *system* returns the termination status of *sh*(1) in the format specified by *waitpid*(2).

FILES

/bin/sh

SEE ALSO

sh(1), exec(2), waitpid(2).

DIAGNOSTICS

System forks to create a child process that in turn exec's **/bin/sh** in order to execute *string*. If the fork fails, *system* returns -1 and sets *errno*. If the exec fails, *system* returns the status value returned by *waitpid*(2) for a process that terminates with a call of *exit*(127).

STANDARDS CONFORMANCE

system: SVID2, XPG2, XPG3, ANSI C

NAME

tcgetattr, tcsetattr – control tty device

SYNOPSIS

```
#include <termios.h>

int tcgetattr (fildes, termios_p)
int fildes;
struct termios *termios_p;

int tcsetattr (fildes, optional_actions, termios_p)
int fildes;
int optional_actions;
struct termios *termios_p;
```

DESCRIPTION

Tcgetattr gets the parameters associated with *fildes* and stores them in the *termios* structure referenced by *termios_p*. If the terminal device does not support split baud rates, the input baud rate stored in the *termios* structure is zero. This function is allowed from a background process (See *termio*(7)). However, the terminal attributes may be subsequently changed by a foreground process.

Tcsetattr sets the parameters associated with *fildes* (unless support is required from underlying hardware that is not available) from the *termios* structure referenced by *termios_p* as follows:

1. If *optional_actions* is **TCSANOW**, the change is immediate.
2. If *optional_actions* is **TCSADRAIN**, the change occurs after all output written to *fildes* is transmitted.
3. If *optional_actions* is **TCSAFLUSH**, the change occurs after all output written to *fildes* is transmitted, and all input that has been received but not read is discarded.

RETURN VALUE

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

ERRORS

These functions will fail if one or more of the following is true:

[EBADF] *Fildes* is not a valid file descriptor.

[ENOTTY] The file associated with *fildes* is not a terminal.

[EINVAL] The *optional_actions* argument is not a proper value.

WARNINGS

A request to set a hardware parameter to a value that is not supported by the hardware being used will be ignored. The remaining parameter values of the request which are supported or which do not affect hardware will be set as requested. For any hardware that does not support separate input and output baud rates, the requested output baud rate will be used to set the actual hardware baud rate. *Tcgetattr* always returns the actual values set in hardware.

SEE ALSO

tccontrol(3C), cfspeed(3C), termio(7).

NAME

tcsendbreak, tcdrain, tcflush, tcflow – tty line control functions

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, *tcsendbreak* causes transmission of a continuous stream of zero-valued bits for at least 0.25 seconds, but not more than 0.5 seconds. For all HP-UX implementations, *duration* is ignored.

Tcdrain waits until all output written to *fildes* has been transmitted.

Tcflush discards data written to *fildes* but not transmitted or data received but not read, depending on the value of *queue_selector*:

(1) If *queue_selector* is **TCIFLUSH**, data received but not read is flushed.

(2) If *queue_selector* is **TCOFLUSH**, data written but not transmitted is flushed.

(3) If *queue_selector* is **TCIOFLUSH**, both data received but not read, and data written but not transmitted is flushed.

Tcflow suspends transmission of data to *fildes* or reception of data from *fildes*, depending on the value of *action*:

(1) If *action* is **TCOOFF**, output is suspended.

(2) If *action* is **TCOON**, suspended output is restarted.

(3) If *action* is **TCIOFF**, a *STOP* character is transmitted which is intended to cause the terminal to stop transmitting data to the system.

(4) If *action* is **TCION**, a *START* character is transmitted which is intended to cause the terminal to start transmitting data to the system.

RETURN VALUE

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

ERRORS

These functions will fail if one or more of the following is true:

[EBADF] *Fildes* is not a valid file descriptor.

[ENOTTY] The file associated with *fildes* is not a terminal.

[EINTR]	A signal was received during *tcdrain*.
[EINVAL]	The *queue_selector* or the *action* argument is not a proper value.

SEE ALSO

tcattribute(3C), tccontrol(3C), termio(7).

STANDARDS CONFORMANCE

tcdrain: XPG3, POSIX.1, FIPS 151-1

tcflow: XPG3, POSIX.1, FIPS 151-1

tcflush: XPG3, POSIX.1, FIPS 151-1

tcsendbreak: XPG3, POSIX.1, FIPS 151-1

NAME

tcgetpgrp – get foreground process group id

SYNOPSIS

#include <sys/types.h>

pid_t tcgetpgrp (fildes)
int fildes;

DESCRIPTION

Tcgetpgrp returns the value of the process group ID of the foreground process group associated with the terminal referenced by *fildes*. *Tcgetpgrp* is allowed from a process that is a member of a background process group (See **termio(7)**); however, the information may be subsequently changed by a process that is a member of a foreground process group.

RETURN VALUE

Upon successful completion, the value of the process group ID of the foreground process group associated with the terminal referenced by *fildes* is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

Tcgetpgrp will fail if one or more of the following is true:

[EBADF] *Fildes* is not a valid file descriptor.

[ENOTTY] The file associated with *fildes* is not the controlling terminal or the calling process does not have a controlling terminal.

[EACCES] The file associated with *fildes* is the controlling terminal of the calling process, however, there is no foreground process group defined for the controlling terminal.

WARNING

The error EACCES, which is returned if the controlling terminal has no foreground process group, may not be returned in future releases, depending on the course taken by the POSIX standard. Portable applications therefore should not rely on this error condition.

SEE ALSO

tcsetpgrp(3C), termio(7), setpgid(2), setsid(2).

STANDARDS CONFORMANCE

tcgetpgrp: XPG3, POSIX.1, FIPS 151-1

NAME

tcsetpgrp – set foreground process group id

SYNOPSIS

#include <sys/types.h>

int tcsetpgrp (fildes, pgrp_id)
int fildes;
pid_t pgrp_id;

DESCRIPTION

If the calling process has a controlling terminal, *tcsetpgrp* sets the foreground process group ID associated with the terminal referenced by *fildes* to *pgrp_id*. 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 *pgrp_id* must match a process group ID of a process in the same session as the calling process.

RETURN VALUE

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

ERRORS

Tcsetpgrp will fail if one or more of the following is true:

[EBADF]	*Fildes* is not a valid file descriptor.
[EINVAL]	The value of the *pgrp_id* argument is not supported.
[ENOTTY]	The calling process does not have a controlling terminal, or the *fildes* is not the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.
[EPERM]	The value of *pgrp_id* is a supported value but does not match the process group ID of a process in the same session as the calling process.

SEE ALSO

termio(7), tcgetpgrp(3C), setsid(2), setpgid(2).

STANDARDS CONFORMANCE

tcsetpgrp: XPG3, POSIX.1, FIPS 151-1

NAME

tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs – emulate /etc/termcap access routines

SYNOPSIS

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

tgetnum(id)
char *id;

tgetflag(id)
char *id;

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

char *tgoto(cm, destcol, destline)
char *cm;

tputs(cp, affcnt, outc)
register char *cp;
int affcnt:
int (*outc)();
```

DESCRIPTION

The *termcap*(3X) functions extract and use capabilities from the compiled terminal capability data bases (see *terminfo*(4)). They are emulation routines that are provided as a part of the *curses*(3X) library.

Tgetent extracts the compiled entry for terminal *name* into buffers accessible by the programmer. Unlike previous termcap routines, all capability strings (except cursor addressing and padding information) are already compiled and stored internally upon return from *tgetent.* The buffer pointer *bp* is redundant in the emulation, and is ignored. It should not be relied upon to point to meaningful information. *Tgetent* returns −1 if it cannot access the *terminfo* directory, 0 if there is no capability file for *name,* and 1 if all goes well. If a TERMINFO environment variable is set, *tgetent* first looks for **TERMINFO/?/name** (where ? is the first character of *name*), and if that file is not accessible, it looks for **/usr/lib/terminfo/?/name**.

Tgetnum gets the numeric value of capability *id,* returning −1 if it is not given for the terminal. *Tgetnum* is useful only with capabilities having numeric values.

Tgetflag returns 1 if the specified capability is present in the terminal's entry, and 0 if it is not. *Tgetflag* is useful only with capabilities that are boolean in nature (i.e. either present or missing in *terminfo*(4)).

Tgetstr returns a pointer to the string value of capability *id.* In addition, if *area* is not a NULL pointer, *tgetstr* will place the capability in the buffer at *area* and advance the area pointer. The returned string capability is compiled except for cursor addressing and padding information. *Tgetstr* is useful only with capabilities having string values.

Tgoto returns a cursor addressing string decoded from *cm* to go to column *destcol* in line *destline*. (Programs which call *tgoto* should be sure to turn off the TAB3 bit(s), since *tgoto* may now output a tab. See *termio*(7). Note that programs using *termcap* should in general turn off TAB3 anyway since some terminals use control–I for other functions, such as nondestructive space.) If a % sequence is given which is not understood, then *tgoto* returns OOPS.

Tputs decodes the padding information of the string *cp. Affcnt* gives the number of lines affected by the operation, or 1 if this is not applicable. *Outc* is a routine which is called with each character in turn. The *terminfo* variable **pad_char** should contain a pad character to be used (from the *pc* capability) if a null (^@) is inappropriate.

FILES

/usr/lib/libcurses.a	−lcurses library
/usr/lib/terminfo/?/*	data bases

SEE ALSO

ex(1), terminfo(4), termio(7).

NAME

tmpfile – create a temporary file

SYNOPSIS

#include <stdio.h>

FILE *tmpfile ()

DESCRIPTION

Tmpfile creates a temporary file by generating a name through *tmpnam*(3S), and returns a corresponding FILE pointer. If the file cannot be opened, an error message is printed using *perror*(3C), and a NULL pointer is returned. The file will automatically be deleted when the process using it terminates. The file is opened for update ("wb+").

NOTES

On HP-UX the "wb+" mode is equivalent to the "w+" mode.

SEE ALSO

creat(2), unlink(2), mktemp(3C), perror(3C), fopen(3S), tmpnam(3S).

STANDARDS CONFORMANCE

tmpfile: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

tmpnam, tempnam − create a name for a temporary file

SYNOPSIS

```
#include <stdio.h>

char *tmpnam (s)
char *s;

char *tempnam (dir, pfx)
char *dir, *pfx;
```

DESCRIPTION

These functions generate file names that can safely be used for a temporary file.

Tmpnam always generates a file name using the path-prefix defined as **P_tmpdir** in the *<stdio.h>* header file. If *s* is NULL, *tmpnam* leaves its result in an internal static area and returns a pointer to that area. The next call to *tmpnam* will destroy the contents of the area. If *s* is not NULL, it is assumed to be the address of an array of at least **L_tmpnam** bytes, where **L_tmpnam** is a constant defined in *<stdio.h>*; *tmpnam* places its result in that array and returns *s*.

Tempnam allows the user to control the choice of a directory. The argument *dir* points to the name of the directory in which the file is to be created. If *dir* is NULL or points to a string which is not a name for an appropriate directory, the path-prefix defined as **P_tmpdir** in the *<stdio.h>* header file is used. If that directory is not accessible, **/tmp** will be used as a last resort. This entire sequence can be up-staged by providing an environment variable **TMPDIR** in the user's environment, whose value is the name of the desired temporary-file directory.

Many applications prefer their temporary files to have certain favorite initial letter sequences in their names. Use the *pfx* argument for this. This argument may be NULL or point to a string of up to five characters to be used as the first few characters of the temporary-file name.

Tempnam uses *malloc*(3C) to get space for the constructed file name, and returns a pointer to this area. Thus, any pointer value returned from *tempnam* may serve as an argument to *free* (see *malloc*(3C)). If *tempnam* cannot return the expected result for any reason, i.e. *malloc*(3C) failed, or none of the above mentioned attempts to find an appropriate directory was successful, a NULL pointer will be returned.

NOTES

Tmpnam and *tempnam* generate a different file name each time they are called, but they will start recycling previously used names if called more than TMP_MAX times in a single process.

Files created using these functions and either *fopen*(3S) or *creat*(2) are temporary only in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to use *unlink*(2) to remove the file when its use is ended.

SEE ALSO

creat(2), unlink(2), malloc(3C), mktemp(3C), fopen(3S), tmpfile(3S).

BUGS

Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using these functions or *mktemp*, and the file names are chosen so as to render duplication by other means unlikely.

STANDARDS CONFORMANCE

tmpnam: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

tempnam: SVID2, XPG2, XPG3

NAME

sin, cos, tan, asin, acos, atan, atan2 – trigonometric functions

SYNOPSIS

```
#include <math.h>

double sin (x)
double x;

double cos (x)
double x;

double tan (x)
double x;

double asin (x)
double x;

double acos (x)
double x;

double atan (x)
double x;

double atan2 (y, x)
double y, x;
```

DESCRIPTION

Sin, *cos*, and *tan* return respectively, the sine, cosine and tangent of their argument, x, measured in radians.

Asin returns the arcsine of x, in the range $-\pi/2$ to $\pi/2$.

Acos returns the arccosine of x, in the range 0 to π.

Atan returns the arctangent of x, in the range $-\pi/2$ to $\pi/2$.

Atan2 returns the arctangent of y/x, in the range $-\pi$ to π, using the signs of both arguments to determine the quadrant of the return value.

DEPENDENCIES

Series 300

The approximate limit for the values returned by these functions is 1.49^8.

The algorithms used for all functions except *atan2* are from HP 9000 BASIC.

Series 800 (/lib/libm.a and ANSI C /lib/libM.a)

When x is ±INFINITY , *atan* returns $\pm\pi/2$ respectively.

Atan2 returns $\pi/4$ when y and x are +INFINITY.

Atan2 returns $-\pi/4$ when y is +INFINITY and x is –INFINITY.

Atan2 returns $3*\pi/4$ when y is –INFINITY and x is +INFINITY.

Atan2 returns $-3*\pi/4$ when y and x are –INFINITY.

Atan2 returns 0.0 when y is 0.0 and x is a positive number.

Atan2 returns π when y is 0.0 and x is a negative number, or $-\pi$ when y is -0.0 and x is a negative number.

Atan2 returns $\pi/2$ when y is a positive number and x is 0.0, or $-\pi/2$ when y is a negative number and x is 0.0.

Atan2 returns $\pm\pi/2$ based on the sign of y if y/x would overflow.

Atan2 returns $-\pi$ or 0.0 based on the sign of y if y/x would underflow.

ERRORS

Series 300

Sin, *cos*, and *tan* lose accuracy when their argument is far from zero. For arguments sufficiently large, these functions return 0.0 when there would otherwise be a complete loss of significance. In this case a message indicating TLOSS error is printed on the standard error output. For less extreme arguments causing partial loss of significance, a PLOSS error is generated but no message is printed. In both cases, **errno** is set to **ERANGE.**

If the magnitude of the argument of *asin* or *acos* is greater than one, or if both arguments of *atan2* are 0.0, 0.0 is returned and **errno** is set to **EDOM**. In addition, a message indicating DOMAIN error is printed on the standard error output.

Series 800 (/lib/libm.a)

Sin, *cos*, and *tan* lose accuracy when their argument is far from zero. For arguments sufficiently large, these functions return 0.0 when there would otherwise be a complete loss of significance. In this case a message indicating TLOSS error is printed on the standard error output. For less extreme arguments causing partial loss of significance, a PLOSS error is generated but no message is printed. In both cases, **errno** is set to **ERANGE.**

If the magnitude of the argument of *asin* or *acos* is greater than one, or if both arguments of *atan2* are 0.0, 0.0 is returned and **errno** is set to **EDOM**. In addition, a message indicating DOMAIN error is printed on the standard error output.

Sin, *cos*, *tan*, *acos*, and *asin* return NaN and set **errno** to **EDOM** when x is NaN or ±INFINITY. In addition, a message indicating DOMAIN error is printed on the standard error output.

Atan returns NaN and sets **errno** to **EDOM** when x is NaN. In addition, a message indicating DOMAIN error is printed on the standard error output.

Atan2 returns NaN and sets **errno** to **EDOM** when x or y is NaN. In addition, a message indicating DOMAIN error is printed on the standard error output.

Series 800 (ANSI C /lib/libM.a)

No error messages are printed on the standard error output.

Sin, *cos*, and *tan* lose accuracy when their argument is far from zero. For arguments sufficiently large, these functions return 0.0 when there would otherwise be a complete loss of significance. For less extreme arguments causing partial loss of significance, a PLOSS error is generated. In both cases, **errno** is set to **ERANGE.**

If the magnitude of the argument of *asin* or *acos* is greater than one, NaN is returned and **errno** is set to **EDOM.**

If both arguments of *atan2* are 0.0, 0.0 is returned and **errno** is set to **EDOM**.

Sin, *cos*, *tan*, *acos* , and *asin* return NaN and set **errno** to **EDOM** when x is NaN or ±INFINITY.

Atan returns NaN and sets **errno** to **EDOM** when x is NaN.

Atan2 returns NaN and sets **errno** to **EDOM** when x or y is NaN.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

isinf(3M), isnan(3M), matherr(3M).

STANDARDS CONFORMANCE

acos: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

asin: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C
atan: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C
atan2: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C
cos: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C
sin: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C
tan: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

tsearch, tfind, tdelete, twalk – manage binary search trees

SYNOPSIS

#include <search.h>

char *tsearch ((char *) key, (char **) rootp, compar)
int (*compar)( );

char *tfind ((char *) key, (char **) rootp, compar)
int (*compar)( );

char *tdelete ((char *) key, (char **) rootp, compar)
int (*compar)( );

void twalk ((char *) root, action)
void (*action)( );

DESCRIPTION

Tsearch, tfind, tdelete, and *twalk* are routines for manipulating binary search trees. They are generalized from Knuth (6.2.2) Algorithms T and D. All comparisons are done with a user-supplied routine, *compar*. This routine is called with two arguments, the pointers to the elements being compared. It returns an integer less than, equal to, or greater than 0, according to whether the first argument is to be considered less than, equal to or greater than the second argument. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Tsearch is used to build and access the tree. **Key** is a pointer to a datum to be accessed or stored. If there is a datum in the tree equal to *key (the value pointed to by key), a pointer to this found datum is returned. Otherwise, *key is inserted, and a pointer to it returned. Only pointers are copied, so the calling routine must store the data. **Rootp** points to a variable that points to the root of the tree. A NULL value for the variable pointed to by **rootp** denotes an empty tree; in this case, the variable will be set to point to the datum which will be at the root of the new tree.

Like *tsearch*, *tfind* will search for a datum in the tree, returning a pointer to it if found. However, if it is not found, *tfind* will return a NULL pointer. The arguments for *tfind* are the same as for *tsearch*.

Tdelete deletes a node from a binary search tree. The arguments are the same as for *tsearch*. The variable pointed to by **rootp** will be changed if the deleted node was the root of the tree. *Tdelete* returns a pointer to the parent of the deleted node, or a NULL pointer if the node is not found.

Twalk traverses a binary search tree. **Root** is the root of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) *Action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The second argument is a value from an enumeration data type *typedef enum { preorder, postorder, endorder, leaf } VISIT;* (defined in the *<search.h>* header file), depending on whether this is the first, second or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level zero.

The pointers to the key and the root of the tree should be of type pointer-to-element, and cast to type pointer-to-character. Similarly, although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

EXAMPLE

The following code reads in strings and stores structures containing a pointer to each string and a count of its length. It then walks the tree, printing out the stored strings and their lengths in alphabetical order.

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

struct node {           /* pointers to these are stored in the tree */
        char *string;
        int length;
};
char string_space[10000];       /* space to store strings */
struct node nodes[500];         /* nodes to store */
struct node *root = NULL;       /* this points to the root */

main( )
{
        char *strptr = string_space;
        struct node *nodeptr = nodes;
        void print_node( ), twalk( );
        int i = 0, node_compare( );

        while (gets(strptr) != NULL && i++ < 500)  {

                /* set node */
                nodeptr->string = strptr;
                nodeptr->length = strlen(strptr);

                /* put node into the tree */
                (void) tsearch((char *)nodeptr, &root,
                        node_compare);

                /* adjust pointers, so we don't overwrite tree */
                strptr += nodeptr->length + 1;
                nodeptr++;
        }
        twalk(root, print_node);
}
/* This routine compares two nodes, based on an
      alphabetical ordering of the string field.  */
int
node_compare(node1, node2)
struct node *node1, *node2;
{
        return strcmp(node1->string, node2->string);
}
/* This routine prints out a node, the first time
      twalk encounters it.  */
void
print_node(node, order, level)
struct node **node;
VISIT order;
int level;
{
```

```
            if (order == preorder || order == leaf)  {
                    (void)printf("string = %20s,  length = %d\n",
                            (*node)->string, (*node)->length);
            }
    }
```

SEE ALSO

bsearch(3C), hsearch(3C), lsearch(3C).

DIAGNOSTICS

A NULL pointer is returned by *tsearch* if there is not enough space available to create a new node.
A NULL pointer is returned by *tsearch*, *tfind* and *tdelete* if **rootp** is NULL on entry.
If the datum is found, both *tsearch* and *tfind* return a pointer to it. If not, *tfind* returns NULL, and *tsearch* returns a pointer to the inserted item.

WARNINGS

The **root** argument to *twalk* is one level of indirection less than the **rootp** arguments to *tsearch* and *tdelete*.
There are two nomenclatures used to refer to the order in which tree nodes are visited. *Tsearch* uses preorder, postorder and endorder to respectively refer to visting a node before any of its children, after its left child and before its right, and after both its children. The alternate nomenclature uses preorder, inorder and postorder to refer to the same visits, which could result in some confusion over the meaning of postorder.

BUGS

If the calling function alters the pointer to the root, results are unpredictable.

STANDARDS CONFORMANCE

tsearch: SVID2, XPG2, XPG3

tdelete: SVID2, XPG2, XPG3

tfind: SVID2, XPG2, XPG3

twalk: SVID2, XPG2, XPG3

NAME

ttyname, isatty – find name of a terminal

SYNOPSIS

char *ttyname (fildes)
int fildes;

int isatty (fildes)
int fildes;

DESCRIPTION

Ttyname returns a pointer to a string containing the null-terminated path name of the terminal device associated with file descriptor *fildes*.

Isatty returns 1 if *fildes* is associated with a terminal device, 0 otherwise.

ERRORS

Isatty or *ttyname* will fail if any of the following is true:

[EBADF] The *fildes* argument is invalid.

[ENOTTY] An inappropriate I/O control operation has been attempted.

FILES

/dev/* /dev/pty/*

DIAGNOSTICS

Ttyname returns a NULL pointer if *fildes* does not describe a terminal device in directory **/dev**.

WARNINGS

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

STANDARDS CONFORMANCE

ttyname: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

isatty: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME
ttyslot – find the slot in the utmp file of the current user

SYNOPSIS
int ttyslot ( )

DESCRIPTION
Ttyslot returns the index of the current user's entry in the **/etc/utmp** file. This is accomplished by actually scanning the file **/etc/inittab** for the name of the terminal associated with the standard input, the standard output, or the error output (0, 1 or 2).

FILES
/etc/inittab
/etc/utmp

SEE ALSO
getut(3C), ttyname(3C).

DIAGNOSTICS
A value of −1 is returned if an error was encountered while searching for the terminal name or if none of the above file descriptors is associated with a terminal device.

STANDARDS CONFORMANCE
ttyslot: XPG2

NAME

ungetc – push character back into input stream

SYNOPSIS

#include <stdio.h>

int ungetc (c, stream)
int c;
FILE *stream;

DESCRIPTION

Ungetc inserts the character *c* (converted to an unsigned char) into the buffer associated with an input *stream*. That character, *c*, will be returned by the next *getc*(3S) call on that *stream*. A successful intervening call to a file positioning function with *stream* (*fseek*, *fsetpos*, or *rewind*) erases all memory of the inserted characters.

Ungetc affects only the buffer associated with the input *stream*. It does not affect the contents of the file corresponding to *stream*.

One character of pushback is guaranteed.

If *c* equals **EOF**, *ungetc* does nothing to the buffer and returns **EOF**.

RETURN VALUE

If successful, *ungetc* returns *c* and clears the end-of-file indicator for the stream. *Ungetc* returns **EOF** if it cannot insert the character.

SEE ALSO

fseek(3S), fsetpos(3S), getc(3S), setbuf(3S).

STANDARDS CONFORMANCE

ungetc: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1, ANSI C

NAME

vprintf, vfprintf, vsprintf – print formatted output of a varargs argument list

SYNOPSIS

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

int vprintf (format, ap)
char *format;
va_iist ap;

int vfprintf (stream, format, ap)
FILE *stream;
char *format;
va_list ap;

int vsprintf (s, format, ap)
char *s, *format;
va_list ap;
```

DESCRIPTION

Vprintf, *vfprintf*, and *vsprintf* are the same as *printf*, *fprintf*, and *sprintf* respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by *varargs*(5).

EXAMPLE

The following demonstrates how *vfprintf* could be used to write an error routine:

```
#include <stdio.h>
#include <varargs.h>
        .
        .
        .
/*
 *      error should be called like
 *              error(function_name, format, arg1, arg2...);
 */
/*VARARGS0*/
void
error(va_alist)
/* Note that the function_name and format arguments cannot be
 *       separately declared because of the definition of varargs.
 */
va_dcl
{
        va_list args;
        char *fmt;

        va_start(args);

        /* print out name of function causing error */
        (void)fprintf(stderr, "ERROR in %s: ", va_arg(args, char *));
        fmt = va_arg(args, char *);

        /* print out remainder of message */
        (void)vfprintf(stderr, fmt, args);
        va_end(args);
```

```
        (void)abort( );
    }
```

SEE ALSO

setlocale(3C), printf(3S), varargs(5).

STANDARDS CONFORMANCE

vprintf: SVID2, XPG2, XPG3, ANSI C

vfprintf: SVID2, XPG2, XPG3, ANSI C

vsprintf: SVID2, XPG2, XPG3, ANSI C

NAME

vscanf, vfscanf, vsscanf – formatted input conversion to a varargs argument list, read from stream file

SYNOPSIS

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

int vscanf (format, ap)
const char *format;
va_list ap;

int vfscanf (stream, format, ap)
FILE *stream;
const char *format;
va_list ap;

int vsscanf (s, format, ap)
char *s;
const char *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), setlocale(3C), varargs(5).

Index to Volume 2

Index
Volume 2

Description	Entry Name(Section)
a64l – convert base-64 value to long integer ASCII string	A64L(3C)
abort – generate an IOT fault	ABORT(3C)
absolute value, floor, ceiling, remainder functions	FLOOR(3M)
absolute value, return integer	ABS(3C)
abs – return integer absolute value	ABS(3C)
accelerator, math, check for presence of	IS_HW_PRESENT(3C)
access and modification times, set or update file	UTIME(2)
access control list (ACL), change owner and/or group in	CHOWNACL(3C)
access control list (ACL), copy to another file	CPACL(3C)
access control list (ACL) information, get	GETACL(2)
access control list (ACL) information, set	SETACL(2)
access control list (ACL) structure, convert string form to	STRTOACL(3C)
access control list (ACL) structure, convert to string form	ACLTOSTR(3C)
access control list; add, modify, or delete entry	SETACLENTRY(3C)
access – determine accessibility of a file	ACCESS(2)
accessibility of a file, determine	ACCESS(2)
access list, get group	GETGROUPS(2)
access list, initialize group	INITGROUPS(3C)
access list, set group	SETGROUPS(2)
access long integer data in a machine-independent fashion	SPUTL(3X)
access mode (permissions) of file, change	CHMOD(2)
access, open, or close a directory and associated *directory stream*	DIRECTORY(3C)
access or build a binary search tree	TSEARCH(3C)
access rights to a file, get a user's effective	GETACCESS(2)
access *utmp* or *wtmp* file	GETUT(3C)
accounting: enable or disable process accounting	ACCT(2)
acct – enable or disable process accounting	ACCT(2)
acltostr – convert access control list (ACL) structure to string form	ACLTOSTR(3C)
acos – trigonometric arcosine function	TRIG(3M)
active controllers on HP-IB, change	HPIB_PASS_CTL(3I)
activity on specified HP-IB bus, stop	HPIB_ABORT(3I)
add a swap device for interleaved paging/swapping	SWAPON(2)
addmntent – add entry to open file system description file	GETMNTENT(3X)
add, modify, or delete access control list entry	SETACLENTRY(3C)
addresses – first locations beyond allocated program regions	END(3C)
add value to environment	PUTENV(3C)
ADVANCE – advance pointer to next 8- or 16-bit character	NL_TOOLS_16(3C)
advance – regular expression substring comparison routines	REGEXP(3X)
alarm clock, set a process's	ALARM(2)
alarm – set a process's alarm clock	ALARM(2)
ALLBASE/HP-UX HP IMAGE programmatic calls	HPIMAGE(3X)
allocate data and stack space then lock process into memory	DATALOCK(3C)
allocated program regions, first locations beyond	END(3C)
allocation, change data segment space	BRK(2)
allocator, fast main memory	MALLOC(3X)
allocator for main memory	MALLOC(3C)
allow interface to enable SRQ line on HP-IB	HPIB_RQST_SRVCE(3I)
almanac – return numeric date information in MPE format	ALMANAC(3X)
arcsine, arccosine, arctangent trigonometric functions	TRIG(3M)
argument list, print formatted output of a varargs	VPRINTF(3S)
argument, varargs, formatted input conversion to a	VSCANF(3S)

Description **Entry Name(Section)**

Description **Entry Name(Section)**

Description | **Entry Name(Section)**

Description **Entry Name(Section)**

Description	**Entry Name(Section)**
pclose – terminate pipe I/O to or from a process	POPEN(3S)
pending signals, examine	SIGPENDING(2)
perform I/O with an HP-IB channel from buffers	HPIB_IO(3I)
perform low-overhead I/O on an HP-IB/GPIO channel	IO_BURST(3I)
permissions mask for file creation, set and get	UMASK(2)
permissions (mode) of file, change access	CHMOD(2)
perror, errno, sys_errlist, sys_nerr – system error messages	PERROR(3C)
pipe – create an interprocess channel	PIPE(2)
pipe I/O to or from a process, open or close	POPEN(3S)
plock – lock process, text, or data in memory	PLOCK(2)
pointer, file, move read/write	LSEEK(2)
pointer for binary search tree, get data	TSEARCH(3C)
pointer for I/O operations on a stream file, get or reposition	FSEEK(3S)
pointer, stream, map to file descriptor	FILENO(3S)
pointer to login name in *utmp*, get	GETLOGIN(3C)
poll on HP-IB bus, conduct a serial	HPIB_SPOLL(3I)
poll on HP-IB bus, conduct parallel	HPIB_PPOLL(3I)
poll on HP-IB, control response to parallel	HPIB_CARD_PPOLL_RESP(3I)
poll, parallel, define interface response	HPIB_PPOLL_RESP_CTL(3I)
poll value occurs, wait until a particular parallel	HPIB_WAIT_ON_PPOLL(3I)
popen – initiate pipe I/O to or from a process	POPEN(3S)
power, logarithm, exponential, square root functions	EXP(3M)
pow, *log10, log, exp, sqrt* – exponential, logarithm, power, square root functions	EXP(3M)
preallocate fast disk storage	PREALLOC(2)
prealloc – preallocate fast disk storage	PREALLOC(2)
prepare execution profile	MONITOR(3C)
presence of hardware capabilities, check for	IS_HW_PRESENT(3C)
preset contents of memory area to specified character	MEMORY(3C)
printf, nl_printf – print formatted output to *standard output*	PRINTF(3S)
print formatted output of a varargs argument list	VPRINTF(3S)
print formatted output to *standard output*, file, or string	PRINTF(3S)
print formatted output with numbered arguments to a file or string	PRINTMSG(3C)
printmsg – print formatted output with numbered arguments to *standard output*	PRINTMSG(3C)
priority of a process, change	NICE(2)
privileges for group, set or get kernel access	GETPRIVGRP(2)
process 16-bit characters, tools to	NL_TOOLS_16(3C)
process accounting, enable or disable	ACCT(2)
process and child process times, get	TIMES(2)
process, calling, set or clear auditing on	SETAUDPROC(2)
process, change priority of a	NICE(2)
process context for context-dependent file search, return	GETCONTEXT(2)
process, create a new	FORK(2)
process, get audit ID (*aid*) for current	GETAUDID(2)
process, get audit process flag for calling	GETAUDPROC(2)
process group ID, create session and set	SETSID(2)
process group ID, foreground, get	TCGETPGRP(3C)
process group ID, foreground, set	TCSETPGRP(3C)
process group ID for job control, set	SETPGID(2)
process interval timer, set or get value of	GETITIMER(2)
process, lock into memory after allocating data and stack space	DATALOCK(3C)

Description	**Entry Name(Section)**
sigpause – atomically release blocked signals and wait for interrupt	SIGPAUSE(2)
sigpause, sigset, sighold, sigrelse, sigignore – signal management	SIGSET(2V)
sigpending – examine pending signals	SIGPENDING(2)
sigprocmask – examine and change blocked signals	SIGPROCMASK(2)
sigrelse, sigignore, sigpause, sigset, sighold – signal management	SIGSET(2V)
sigsetmask – set current ignorable signals mask	SIGSETMASK(2)
sigset, sighold, sigrelse, sigignore, sigpause – signal management	SIGSET(2V)
sigspace – define or delete additional signal stack space	SIGSPACE(2)
sigstack – set and/or get signal stack context	SIGSTACK(2)
sigsuspend – wait for a signal	SIGSUSPEND(2)
sigvec – 4.2 BSD-compatible *sigvec* system call	BSDPROC(2)
sigvec system call, 4.2 BSD-compatible	BSDPROC(2)
sigvector – software signal facilities	SIGVECTOR(2)
sine trigonometric function	TRIG(3M)
sinh – hyperbolic sin function	SINH(3M)
sin – trigonometric sin function	TRIG(3M)
sixteen-bit characters, tools to process	NL_TOOLS_16(3C)
sjtojis, sjtouj – JIS, Shift JIS and UJIS code conversion	JCODE(3X)
sleep – suspend execution for interval	SLEEP(3C)
slot in the *utmp* file of the current user, find	TTYSLOT(3C)
software signal facilities	SIGVECTOR(2)
software signal, raise a	SSIGNAL(3C)
sorted tables, binary search routine for	BSEARCH(3C)
sort, quicker	QSORT(3C)
space allocation, change data segment	BRK(2)
space for signal stack, define, delete, or get amount of	SIGSPACE(2)
space, stack and data, allocate then lock process into memory	DATALOCK(3C)
spawn new process in a virtual memory efficient way	VFORK(2)
special file, control character device	IOCTL(2)
special file, FIFO, make a	MKFIFO(3C)
special file, set up I/O read termination character on	IO_EOL_CTL(3I)
special or ordinary file, make a directory, or a	MKNOD(2)
specify I/O read termination character on special file	IO_EOL_CTL(3I)
specify what to do upon receipt of a signal	SIGNAL(2)
speed, inform system of required minimum I/O transfer	IO_SPEED_CTL(3I)
split floating-point into mantissa and exponent	FREXP(3C)
sprintf, nl_sprintf – print formatted output to a string	PRINTF(3S)
sprintmsg – print formatted output with numbered arguments to a string	PRINTMSG(3C)
sputl – place a 4-byte long integer in memory	SPUTL(3X)
sqrt, log, exp, log10, pow – exponential, logarithm, power, square root functions	EXP(3M)
square root, power, logarithm, exponential functions	EXP(3M)
srand48, seed48, lcong48 – initialize pseudo-random number generator	DRAND48(3C)
srand – reset random-number generator to random starting point	RAND(3C)
SRQ line on HP-IB, allow interface to enable	HPIB_RQST_SRVCE(3I)
sscanf, nl_sscanf – formatted read from character string	SCANF(3S)
ssignal – raise a software signal and perform an action	SSIGNAL(3C)
stack and data space, allocate then lock process into memory	DATALOCK(3C)
stack context, signal, set and/or get	SIGSTACK(2)
stack environment, save/restore for non-local goto	SETJMP(3C)
stack space for signals, define, delete, or get amount of	SIGSPACE(2)