HP-UX Reference

Release 11.0

System Calls and File Formats

Sections 2 and 4

Volume 3 of 5

Edition 1

HEWLETT® PACKARD

B2355-90166
E1097

Printed in: United States

Legal Notices

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

Hewlett-Packard 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 shall not be held 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 is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DOD agencies, and subparagraphs (c) (1) and (c) (2) of the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

HEWLETT-PACKARD COMPANY
3000 Hanover Street
Palo Alto, California 94304 U.S.A.

Use of this manual and flexible disk(s) or tape cartridge(s) supplied for this pack is restricted to this product only. Additional copies of the programs may be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

This software 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. Minor changes may be made at reprint without changing the printing date. the manual part number will change when extensive changes are made.

Manual updates may be issued between editions to correct errors or document product changes. To ensure that you receive the updated or new editions, you should subscribe to the appropriate product support service. See your HP sales representative for details.

First Edition: October 1997 (HP-UX Release 11.0)

Volume Three
Table of Contents

Volume Three
Table of Contents

Table of Contents
Volume Three

Section 2: System Calls

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

Table of Contents
Volume Three

Table of Contents
Volume Three

Section 4: File Formats

Table of Contents
Volume Three

Section 2

System Calls

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. Therefore, it should be tested only after an error has been indicated.

SEE ALSO

intro(3), errno(2), hier(5), Introduction(9).

a

NAME

accept - accept a connection on a socket

SYNOPSIS

#include <sys/socket.h>

AF_CCITT only

#include <x25/x25addrstr.h>

int accept(int s, void *addr, int *addrlen);

_XOPEN_SOURCE_EXTENDED only (UNIX 98)

int accept(int s, struct sockaddr *addr, socklen_t *addrlen);

Obsolescent _XOPEN_SOURCE_EXTENDED only (UNIX 95)

int accept(int s, struct sockaddr *addr, size_t *addrlen);

DESCRIPTION

The **accept()** system call is used with connection-based socket types, such as **SOCK_STREAM**. The argument, *s*, is a socket descriptor created with **socket()**, bound to a local address by **bind()**, and listening for connections after a **listen()**. **accept()** extracts the first connection on the queue of pending connections, creates a new socket with the same properties as *s*, and returns a new file descriptor, *ns*, for the socket.

If no pending connections are present on the queue and nonblocking mode has not been enabled with the **fcntl() O_NONBLOCK** or **O_NDELAY** flags or the **ioctl() FIOSNBIO** request, **accept()** blocks the caller until a connection is present. **O_NONBLOCK** and **O_NDELAY** are defined in **<sys/fcntl.h>** (see *fcntl*(2), *fcntl*(5), and *socket*(7)). **FIOSNBIO** and the equivalent request **FIONBIO** are defined in **<sys/ioctl.h>**, although use of **FIONBIO** is not recommended (see *ioctl*(2), *ioctl*(5), and *socket*(7)).

If the socket has nonblocking mode enabled and no pending connections are present on the queue, **accept()** returns an error as described below. The accepted socket, *ns*, cannot be used to accept more connections. The original socket *s* remains open for incoming connection requests. To determine whether a listening socket has pending connection requests ready for an **accept()** call, use **select()** for reading.

The argument *addr* should point to a socket address structure. The **accept()** call fills in this structure with the address of the connecting entity, as known to the underlying protocol. In the case of AF_UNIX sockets, the peer's address is filled in only if the peer had done an explicit **bind()** before doing a **connect()**. Therefore, for AF_UNIX sockets, in the common case, when the peer had not done an explicit **bind()** before doing a **connect()**, the structure is filled with a string of nulls for the address. The format of the address depends upon the protocol and the address-family of the socket *s*.

The argument *addrlen* is a pointer to a variable. Initially, the variable should contain the size of the structure pointed to by *addr*. On return, it contains the actual length (in bytes) of the address returned. If the memory pointed to by *addr* is not large enough to contain the entire address, only the first *addrlen* bytes of the address are returned. If *addr* is NULL or *addrlen* contains 0, the connecting entity's address will not be returned.

The **fcntl() O_NONBLOCK** and **O_NDELAY** flags and **ioctl() FIOSNBIO** request are all supported. These features interact as follows:

- If the **O_NONBLOCK** or **O_NDELAY** flag has been set, **accept()** requests behave accordingly, regardless of any **FIOSNBIO** requests.
- If neither the **O_NONBLOCK** flag nor the **O_NDELAY** flag has been set, **FIOSNBIO** requests control the behavior of **accept()**.

AF_CCITT only

The *addr* parameter to **accept()** returns addressing information for the connecting entity, except for the **x25ifname[]** field of *addr* which contains the name of the local X.25 interface through which the connection request arrived. Call-acceptance can be controlled with the **ioctl() X25_CALL_ACPT_APPROVAL** request (see *socketx25*(7)).

RETURN VALUE

Upon successful completion, **accept()** returns a nonnegative integer which is a descriptor for the accepted socket.

If an error occurs, **accept()** returns **-1** and sets **errno** to indicate the cause.

ERRORS

If **accept()** fails, **errno** is set to one of the following values:

[EAGAIN]	Nonblocking I/O is enabled using **O_NONBLOCK** and no connections are present to be accepted.
[EBADF]	The argument, *s*, is not a valid file descriptor.
[EFAULT]	The *addr* parameter is not a valid pointer.
[EINTR]	The call was interrupted by a signal before a valid connection arrived.
[EINVAL]	The socket referenced by *s* is not currently a listen socket or has been shut down with **shutdown()**. A **listen()** must be done before an **accept()** is allowed.
[EMFILE]	The maximum number of file descriptors for this process are currently open.
[ENFILE]	The system's table of open files is full and no more **accept()** calls can be processed at this time.
[ENOBUFS]	No buffer space is available. The **accept()** cannot complete. The queued socket connect request is aborted.
[ENOMEM]	No memory is available. The **accept()** cannot complete. The queued socket connect request is aborted.
[ENOTSOCK]	The argument, *s*, is a valid file descriptor, but it is not a socket.
[EOPNOTSUPP]	The socket referenced by *s* does not support **accept()**.
[EWOULDBLOCK]	Nonblocking I/O is enabled using **O_NDELAY** or **FIOSNBIO** and no connections are present to be accepted.

OBSOLESCENCE

Currently, the **socklen_t** and **size_t** types are the same size. This is compatible with both the UNIX 95 and UNIX 98 profiles. However, in a future release, **socklen_t** might be a different size. In that case, passing a **size_t** pointer will evoke compile-time warnings, which must be corrected in order for the application to behave correctly. Applications that use **socklen_t** now, where appropriate, will avoid such migration problems. On the other hand, applications that need to be portable to the UNIX 95 profile should follow the X/Open specification (see *xopen_networking*(7)).

FUTURE DIRECTION

Currently, the default behavior is the **HP-UX BSD Sockets**; however, it might be changed to **X/Open Sockets** in a future release. At that time, any **HP-UX BSD Sockets** behavior that is incompatible with **X/Open Sockets** might be obsoleted. Applications that conform to the X/Open specification now will avoid migration problems (see *xopen_networking*(7)).

MULTITHREAD USAGE

The **accept()** system call is thread-safe. It has a cancellation point; and it is async-cancel safe, async-signal safe, and fork-safe.

AUTHOR

accept() was developed by HP and the University of California, Berkeley.

SEE ALSO

bind(2), connect(2), listen(2), select(2), socket(2), socketx25(7), xopen_networking(7).

STANDARDS CONFORMANCE

accept(): XPG4

a

NAME

access - determine accessibility of a file

SYNOPSIS

```
#include <unistd.h>

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

DESCRIPTION

The **access()** system call checks the file pointed to by *path* for accessibility according to the bit pattern contained in *amode*. **access()** uses the real user ID, not the effective user ID, and the real group ID, not the effective group ID.

The value of *amode* is either the bit-wise inclusive OR of the access permissions to be checked, or the existence test. You can use the following symbolic constants, defined in **<unistd.h>**, to test for permissions:

R_OK	Read access
W_OK	Write access
X_OK	Execute (search) access
F_OK	Check existence of file

The owner of a file has permission checked with respect to the "user" 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. All others have permissions checked with respect to the "other" mode bits.

If a file is currently open for execution, **access()** reports that it is not writable, regardless of the setting of its mode.

Access Control Lists - HFS File Systems Only

Read, write, and execute/search permissions are checked against the file's access control list (ACL). Each mode is checked separately since different ACL entries can 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 *user*.*group* or **%.***group* entry can match a user if that user has a nonnull supplementary groups list. If any matching entry has the appropriate permission bit set, access is permitted.

If a shared text file is currently open for execution, **access()** reports that it is not writable, regardless of its access control list. However, **access()** does not report that a shared text file open for writing is not executable, since the check is not easily done.

It also reports that a file on a read-only file system is not writable.

RETURN VALUE

access() returns the following values:

0 Successful completion. The requested access is permitted.

If the path is valid and the real user ID is superuser, **access()** always returns 0, except when *amode* includes **X_OK**, the path is not a directory, and none of the execute bits are set in the file's mode.

-1 Failure. **errno** is set to indicate the error.

ERRORS

If **access()** fails, **errno** is set to one of the following values.

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

[EACCES] The access control list does not permit the requested access and the real user ID is not a user with appropriate privileges.

[EFAULT] *path* points outside the allocated address space for the process. 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.

[ENOENT] Read, write, or execute (search) permission is requested for a null path name.

[ENOENT] The named file does not exist.

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

[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.

SEE ALSO

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

STANDARDS CONFORMANCE

access(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

a

NAME

acct() - enable or disable process accounting

SYNOPSIS

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

int acct(const char *path);
```

DESCRIPTION

The **acct()** system call enables or disables the system's process accounting routine. If the routine is enabled, an accounting record is 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 superuser to use this call.

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

The accounting routine is enabled if *path* is nonzero 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 reenabled when the space reaches a second configurable threshold.

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 with the **ckpacct** command. **ckpacct** and **turnacct** are described in *acctsh*(1M)).

RETURN VALUE

acct() returns the following values:

- **0** Successful completion.
- **-1** Failure. **errno** is set to indicate the error.

ERRORS

If **acct()** fails, **errno** is set to one of the following values.

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

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

[EFAULT] *path* points to an illegal address. The reliable detection of this error simplementation dependent.

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

[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.

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

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

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

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

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

SEE ALSO

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

STANDARDS CONFORMANCE

acct(): SVID2, SVID3, XPG2

NAME

adjtime() - correct the time to synchronize the system clock

SYNOPSIS

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

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

DESCRIPTION

The function **adjtime()** adjusts the current time of the system. The time is either advanced or retarded by the amount of time specified in the struct timeval pointed to by *delta*.

The adjustment is made by applying small correctional adjustments to the value of current time that the system keeps. The time is always increasing monotonically, but at a rate slightly slower or faster than normal.

A time correction for an earlier call to **adjtime()** may not be complete when **adjtime()** is called. The second call to **adjtime()** stops the first call to **adjtime()** if *delta* is non-NULL, but does not undo the effects of the previous call. If *delta* is NULL, then no time correction will be done.

If *olddelta* is not a NULL pointer, then the structure it points to will contain, upon return, the number of seconds and/or microseconds still to be corrected from the earlier call. If *olddelta* is a NULL pointer, the corresponding information will not be returned.

The call to **adjtime()** returns immediately, though its effect will continue until the whole correction is made or until modified by another call to either **adjtime()** with a non-NULL *delta* or to change the system time (see "Interaction with Other System Calls").

Only a user with appropriate privileges can call **adjtime()** successfully with a non-NULL *delta*. Any user can call **adjtime()** with a NULL delta to report the correction left from the previous call.

Limits

struct timeval is defined in <**time.h**> as having at least 2 members:

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

When **adjtime()** is called, if the delta.tv_sec field is greater than 31536000 (approx. 365 days), or less than –31536000, then **adjtime()** fails with an errno of EINVAL. The tv_usec field is not used in the calculations to determine the limits, and so the actual limit on adjustments are [–31536000–LONG_MIN, 31536000+LONG_MAX].

Note that the desired seconds may be negative. Since the type of the tv_sec field is (unsigned long), any negative values for tv_sec need to be cast.

Any *olddelta* value returned by the **adjtime()** function will be returned such that the signs of non-zero members are the same.

Interaction with Other System Calls

A call to change the system time terminates the **adjtime()** correction currently in effect. A subsequent call to **adjtime()** will return {0, 0} for the *olddelta* parameter. This includes system calls such as **settimeofday(), stime(),** and **clock_settime().**

RETURN VALUE

Upon successful completion, **adjtime()** returns a value of 0; otherwise, it returns a value –1 and sets **errno** to indicate the error.

ERRORS

adjtime() fails if one or more of the following is true:

[EPERM] if the process does not have the appropriate privilege.

a

[EFAULT] The address specified for *delta (or olddelta*) is invalid.

[EINVAL] If *delta.tv_sec* is greater than 31536000 (approx. 365 days) or less than −31536000. The *delta.tv_usec* field is not used in calculation of these limits. If the user wants to adjust time greater than these limits, an appropriate alternative interface should be used.

EXAMPLES

The following code snippet will take the time forward 20 minutes.

```
struct timeval forward;

forward.tv_sec = 20 * 60;   /* 20 minutes */
forward.tv_usec = 0;

if (adjtime(&forward, (struct timeval *)NULL) == -1)
    perror("adjtime() failure");

/*
 *  If adjtime() succeeds, the system time will move forward
 *  20 minutes over a period of time.
 */
```

The following code fragment will repeatedly call a user-defined function adjustment_still_in_progress() until the adjustment requested in a previous call to **adjtime()** (called from either the same process or another process) is completed.

```
struct timeval report;

if (adjtime((struct timeval *)NULL, &report) == -1)
     perror("adjtime() failure");

while (report.tv_sec || report.tv_usec) {
     adjustment_still_in_progress();

     if (adjtime((struct timeval *)NULL, &report) == -1)
          perror("adjtime() failure");
}
```

AUTHOR

adjtime() was developed by the University of California, Berkeley and AT&T.

SEE ALSO

date(1), gettimeofday(2), settimeofday(2), stime(2), clock_settime(2), getitimer(2), setitimer(2), alarm(2).

NAME

aio_cancel() - cancel an asynchronous I/O operation

SYNOPSIS

```
#include <aio.h>

int aio_cancel(int fildes, struct aiocb *aiocbp);
```

DESCRIPTION

The **aio_cancel()** function attempts to cancel the asynchronous I/O request currently outstanding for the **aiocb** referenced by *aiocbp* or, if *aiocbp* is **NULL**, any asynchronous I/O operations currently outstanding for the file descriptor *fildes*.

If an asynchronous I/O operation is successfully canceled as a result of **aio_cancel**, its status is set to **ECANCELED**, and any signal delivery specified for that operation is performed. Any outstanding requests that cannot be canceled as a result of the **aio_cancel()** remain enqueued and are unaffected by the cancellation request.

Asynchronous I/O operations that are requested as a single logical operation are either completed or canceled atomically. Once any portion of the operation has started, it cannot be canceled. Whether or not and when an asynchronous I/O operation can be canceled depends on the nature of the request.

If *aiocbp* is not **NULL**, *fildes* is ignored.

To use this function, link in the realtime library by specifying **-lrt** on the compiler or linker command line.

RETURN VALUE

The **aio_cancel()** function returns one of the following values:

AIO_CANCELED
: The asynchronous I/O operation enqueued for the **aiocb** referenced by *aiocbp* or all asynchronous I/O operations enqueued for the file referenced by *fildes* have been successfully canceled.

AIO_NOTCANCELED
: The asynchronous I/O operation enqueued for the **aiocb** referenced by *aiocbp* or at least one of the asynchronous I/O operations enqueued for the file referenced by *fildes* have not been canceled. (The **aio_error()** function must be used to determine the status of individual operations.)

AIO_ALLDONE
: The asynchronous I/O operation enqueued for the **aiocb** referenced by *aiocbp* or all of the asynchronous I/O operations enqueued for the file referenced by *fildes* completed before cancellation could be attempted.

-1
: Failure. The requested cancellation could not be initiated. **errno** is set to indicate the error.

ERRORS

If **aio_cancel()** detects one of the following error conditions, **errno** is set to the indicated value:

[EBADF]
: The *aiocbp* argument is **NULL** and the *fildes* argument is not a valid file descriptor.

[EINVAL]
: There was no asynchronous I/O operation enqueued for the **aiocb** referenced by *aiocbp*.

SEE ALSO

aio_error(2), aio_fsync(2), aio_read(2), aio_return(2), aio_suspend(2), aio_write(2), lio_listio(2), aio(5).

STANDARDS CONFORMANCE

aio_cancel(): POSIX Realtime Extensions, IEEE Std 1003.1b

a

NAME

aio_error() - return error status of an asynchronous I/O operation

SYNOPSIS

```
#include <aio.h>

int aio_error(const struct aiocb *aiocbp);
```

DESCRIPTION

The **aio_error()** function returns the error status of the asynchronous I/O operation that was initiated with the **aiocb** and referenced by **aiocbp**. The error status for an asynchronous I/O operation is the **errno** value set by the corresponding **read()**, **write()**, or **fsync()** function.

To use this function, link in the realtime library by specifying **-lrt** on the compiler or linker command line.

RETURN VALUE

If the **aiocb** is invalid or if no asynchronous I/O operation is enqueued for the **aiocb**, **aio_error()** returns **-1** and **errno** is set to indicate the error. If the operation has been queued but not completed, **aio_error()** returns **EINPROGRESS**. Otherwise, **aio_error()** returns the error status of the referenced **aiocb**. See *aio_read*(2), *read*(2), *aio_write*(2), *write*(2), *aio_fsync*(2), *fsync*(2), and *lio_listio*(2) for relevant error values.

ERRORS

If **aio_error()** detects one of the following error conditions, **errno** is set to the indicated value:

[EINVAL] There was no asynchronous I/O operation enqueued for the referenced. **aiocb**.

EXAMPLE

The following code sequence illustrates using **aio_error()** to retrieve the error status of an **aio_read()** operation.

```
#include <fcntl.h>
#include <errno.h>
#include <aio.h>
char buf[4096];
ssize_t nbytes; int retval;
struct aiocb myaiocb;
bzero( &myaiocb, sizeof (struct aiocb));
myaiocb.aio_fildes = open( "/dev/null", O_RDONLY);
myaiocb.aio_offset = 0;
myaiocb.aio_buf = (void *) buf;
myaiocb.aio_nbytes = sizeof (buf);
myaiocb.aio_sigevent.sigev_notify = SIGEV_NONE;
retval = aio_read( &myaiocb );
if (retval) perror("aio_read:");
/* continue processing */
 ...
/* wait for completion */
while ( (retval = aio_error( &myaiocb) ) == EINPROGRESS) ;
/* free the aiocb */
nbytes = aio_return( &myaiocb);
```

SEE ALSO

aio_cancel(2), aio_fsync(2), aio_read(2), aio_return(2), aio_suspend(2), aio_write(2), fsync(2), lio_listio(2), read(2), write(2), aio(5).

STANDARDS CONFORMANCE

aio_error(): POSIX Realtime Extensions, IEEE Std 1003.1b

NAME

aio_fsync() - force outstanding asynchronous operations on a file to the synchronized state

SYNOPSIS

```
#include <aio.h>

int aio_fsync(int op, struct aiocb *aiocbp);
```

DESCRIPTION

The **aio_fsync()** function asynchronously forces all I/O operations that are enqueued at the time of the call for the file or device referenced by **aiocbp->aio_fildes** to the synchronized I/O state. The function call returns when the synchronization request has been enqueued to the file or device (even when the data cannot be synchronized immediately).

Successful completion of the **aio_fsync()** request indicates that all modified data for **aiocbp->fildes** has been moved to a permanent storage device. The **aio_fsync()** function affects only those asynchronous I/O operations enqueued at the time of the call. Subsequently enqueued operations are not included in the synchronizing operation.

The **aio_fsync()** function supports synchronized I/O for regular files, block special files, and character special files.

If the **op** is **O_DSYNC**, all currently enqueued asynchronous I/O operations for **aiocbp->fildes** are completed as if by a call to **fdatasync()**. All data is forced to permanent storage but the meta-data (such as modification times) for the file descriptor is not necessarily updated. If the **op** is **O_SYNC**, all currently enqueued asynchronous I/O operations for **aiocbp->fildes** are completed as if by a call to **fsync()**. All data is forced to permanent storage and the file descriptor metadata is updated.

If an **aio_fsync()** request is issued for a file when there is already a pending **aio_fsync()** request, the first request is treated as though it were part of the second, and the second request will not complete until the first has completed.

The **aio_fsync()** function returns when the *fsync* request has been enqueued for the referenced file or device. The **aio_error()** and **aio_return()** functions must be used to retrieve the status of the synchronization operation via the **aiocb** referenced by **aiocbp**. The status returned will be **EINPROGRESS** until the last operation addressed by the initial request completes. If all operations complete successfully, the error status will be 0 (zero). Otherwise, the error status will be the error status that will be returned for the read or write operation that failed.

If **aiocbp->aio_sigevent** is a valid signal event structure, then the designated signal will be delivered when the requested synchronization operation completes, either when all subject requests have completed successfully or when any one of the requests has failed.

To use this function, link in the realtime library by specifying **-lrt** on the compiler or linker command line.

RETURN VALUE

If the **aio_fsync()** function fails, **-1** is returned and **errno** is set to indicate the error.

ERRORS

If **aio_fsync()** detects one of the following error conditions, **errno** is set to the indicated value:

[EAGAIN] The request could not be queued because a per-process or system-wide limit on asynchronous I/O operations or asynchronous threads would have been exceeded.

[EBADF] The **aiocbp->aio_fildes** is not a valid file descriptor open for writing.

[EINVAL] Synchronized I/O is not supported for the file specified by **aiocbp->aio_fildes**.

[EINVAL] The **aiocb->aio_sigevent** is not a valid address in the process virtual address space.

[EINVAL] The parameters of the indicated **sigevent** in **aiocb->aio_sigevent** are invalid.

SEE ALSO

aio_cancel(2), aio_error(2), aio_read(2), aio_return(2), aio_suspend(2), aio_write(2), fdatasync(2), fsync(2), lio_listio(2), read(2), write(2), aio(5).

STANDARDS CONFORMANCE

aio_fsync(): POSIX Realtime Extensions, IEEE Std 1003.1b

a

NAME

aio_read() - start an asynchronous read operation

SYNOPSIS

```
#include &lt;aio.h&gt;
int aio_read(struct aiocb *aiocbp);
```

DESCRIPTION

The **aio_read()** function allows the calling process to perform an asynchronous read from a previously opened file. The function call returns when the read operation has been enqueued for processing. Once enqueued, processing of the read operation may proceed concurrently with execution of the calling process thread.

If an error condition is detected that prevents the read request from being enqueued, **aio_read()** returns **-1** and sets **errno** to indicate the cause of the failure. Once the read operation has been successfully enqueued, an **aio_error()** and **aio_return()** function referencing the **aiocb** referred to by **aiocbp** must be used to determine its status and any error conditions, including those normally reported by **read()**. The request remains enqueued and consumes process and system resources until **aio_return()** is called.

The **aio_read()** function allows the calling process to read **aiocbp->aio_nbytes** from the file associated with **aiocbp->aio_fildes** into the buffer pointed to by **aiocbp->aio_buf**. The priority of the read operation is reduced by the value of **aiocbp->aio_reqprio**, which must be a value between 0 (zero) and a maximum value which can be obtained using the **sysconf()** call with the argument **_SC_AIO_PRIO_DELTA_MAX**. A value of 0 (zero) yields no reduction in priority. The **aiocbp->aio_lio_opcode** field is ignored.

The read operation takes place at the absolute position in the file given by **aiocbp->aio_offset**, as if **lseek()** were called immediately prior to the operation with **offset** equal to **aiocbp->aio_offset** and **whence** set to **SEEK_SET**. However, the value of the file offset is never changed by asynchronous I/O operations.

Altering the contents of or deallocating memory associated with the **aiocb** referred to by **aiocbp** or the buffer referred to by **aiocbp->aio_buf** while an asynchronous read operation is outstanding may produce unpredictable results because **aio_return()** has not been called for the **aiocb**.

If **aiocbp->aio_sigevent** is a valid signal event structure, then the designated signal will be delivered when the requested asynchronous read operation completes.

To use this function, link in the realtime library by specifying **-lrt** on the compiler or linker command line.

RETURN VALUE

aio_read() returns the following values:

0	Successful completion, the operation has been enqueued.
-1	Failure. The requested operation was not enqueued. **errno** is set to indicate the error.

The return value from **aio_read()** reflects the success or failure of enqueuing the requested read operation for asynchronous processing. **aio_read()** fails if an error in the function call is immediately detected, or if system resource limits prevent the request from being enqueued. Other error conditions are reported asynchronously and must be retrieved with **aio_error()** and **aio_return()**.

ERRORS

If **aio_read()** detects one of the following error conditions, **errno** is set to the indicated value:

[EAGAIN]	The request could not be queued either because of a resource shortage or because the per-process or system-wide limit on asynchronous I/O operations or asynchronous threads would have been exceeded.
[EINVAL]	**aiocb->aio_sigevent** is not a valid address in the process virtual address space.
[EINVAL]	The parameters of the indicated **sigevent** in **aiocb->aio_sigevent** are invalid.

a

[EEXIST] The **aiocbp** is already in use for another asynchronous I/O operation.

Once the read request has been enqueued by **aio_read()**, all of the errors normally reported by the **read()** function and the following errors may be reported asynchronously and returned in a subsequent call to **aio_error()** or **aio_return()** referencing the **aiocb** supplied in the successful **aio_read()** call.

[EBADF] The **aiocbp->aio_fildes** was not a valid file descriptor open for reading.

[EINVAL] The value of **aiocbp->aio_reqprio** is not valid.

[EINVAL] The value of **aiocbp->aio_nbytes** is invalid.

[EINVAL] The file offset implied by **aiocbp->aio_offset** or **aiocbp->aio_offset**+**aiocbp->aio_nbytes** are not valid for the file at the time the request is processed.

[ECANCELED]
The read operation was canceled due to a subsequent call to **aio_cancel()**.

EXAMPLE

The following code sequence and call to **aio_read()** starts an asynchronous read operation.

```
#include <fcntl.h>
#include <errno.h>
#include <aio.h>
char buf[4096];
ssize_t retval; ssize_t nbytes;
struct aiocb myaiocb;
bzero( &myaiocb, sizeof (struct aiocb));
myaiocb.aio_fildes = open( "/dev/null", O_RDONLY);
myaiocb.aio_offset = 0;
myaiocb.aio_buf = (void *) buf;
myaiocb.aio_nbytes = sizeof (buf);
myaiocb.aio_sigevent.sigev_notify = SIGEV_NONE;
retval = aio_read( &myaiocb );
if (retval) perror("aio_read:");
/* continue processing */
 ...
/* wait for completion */
while ( (retval = aio_error( &myaiocb) ) == EINPROGRESS) ;
/* free the aiocb */
nbytes = aio_return( &myaiocb);
```

SEE ALSO

aio_cancel(2), aio_error(2), aio_fsync(2), aio_return(2), aio_suspend(2), aio_write(2), lio_listio(2), read(2), aio(5).

STANDARDS CONFORMANCE

aio_read(): POSIX Realtime Extensions, IEEE Std 1003.1b

a

NAME

aio_return() - return status of an asynchronous I/O operation

SYNOPSIS

#include <aio.h>

ssize_t aio_return(struct aiocb *aiocbp);

DESCRIPTION

The **aio_return()** function returns the return status associated with the **aiocb** structure referenced by the *aiocbp* argument. The return value for an asynchronous I/O operation is the value that would be set by the corresponding **read()**, **write()**, or **fsync()** operation. If the operation has been queued but not completed, **aio_return()** returns **-1** and **errno** is set to **EINPROGRESS**. A successful **aio_return()** call frees all kernel resources associated with the calls **aiocb** referenced by *aiocbp*.

To use this function, link in the realtime library by specifying **-lrt** on the compiler or linker command line.

RETURN VALUE

If the **aiocb** is invalid or if no asynchronous I/O operation is enqueued for the **aiocb**, **aio_returns()** returns **-1** and **errno** is set to indicate the error. Otherwise, **aio_return()** returns the error status of the referenced **aiocb**. See *aio_read*(2), *read*(2), *aio_write*(2), *write*(2), *aio_fsync*(2), *fsync*(2) and *lio_listio*(2) for relevant error values.

ERRORS

If **aio_return()** detects one of the following error conditions, **errno** is set to the indicated value:

[EINVAL] The *aiocbp* is not a valid address within the process virtual address space.

[EINVAL] There was no asynchronous I/O operation enqueued for the referenced **aiocb**.

EXAMPLE

The following code sequence illustrates using **aio_return()** to retrieve the error status of an **aio_read()** operation and free the **aiocb** for future re-use.

```
#include <fcntl.h>
#include <errno.h>
#include <aio.h>
char buf[4096];
int retval; ssize_t nbytes;
struct aiocb myaiocb;
bzero( &myaiocb, sizeof (struct aiocb));
myaiocb.aio_fildes = open( "/dev/null", O_RDONLY);
myaiocb.aio_offset = 0;
myaiocb.aio_buf = (void *) buf;
myaiocb.aio_nbytes = sizeof (buf);
myaiocb.aio_sigevent.sigev_notify = SIGEV_NONE;
retval = aio_read( &myaiocb );
if (retval) perror("aio_read:");
/* continue processing */
 ...
/* wait for completion */
while ( (retval = aio_error( &myaiocb) ) == EINPROGRESS) ;
/* free the aiocb */
nbytes = aio_return( &myaiocb);
```

SEE ALSO

aio_cancel(2), aio_error(2), aio_fsync(2), aio_read(2), aio_suspend(2), aio_write(2), fsync(2), lio_listio(2), read(2), write(2), aio(5).

STANDARDS CONFORMANCE

aio_return() : POSIX Realtime Extensions, IEEE Std 1003.1b

a

NAME

aio_suspend() - wait for an asynchronous I/O operation to complete

SYNOPSIS

```
#include <aio.h>

int aio_suspend(const struct aiocb * const list[], int nent, const
    struct timespec *timeout);
```

DESCRIPTION

The **aio_suspend()** function suspends the calling process thread until at least one of the asynchronous I/O operations initiated with one of the **nent aiocb** pointers contained in **list** has completed, a signal interrupts the function, a *timeout* is not **NULL**, or the time interval specified by *timeout* has passed.

Multiple threads may issue simultaneous calls to **aio_suspend()**, referencing one or more **aiocbs** in common.

To use this function, link in the realtime library by specifying **-lrt** on the compiler or linker command line.

RETURN VALUE

aio_suspend() returns the following values:

0 Successful completion. Either there were no **non-NULL aiocbs** in **list** or at least one of the asynchronous I/O operations enqueued for an **aiocb** referenced by **list** has completed. The completion status of the referenced asynchronous I/O operations must be determined using **aio_error()** and **aio_return()** for each relevant **aiocb**.

-1 Failure. The process thread is not suspended and **errno** is set to indicate the error.

If any of the indicated asynchronous I/O operations has already completed at the time of the call to **aio_suspend()**, then **aio_suspend()** returns immediately. If **nent** is 0 (zero), the **aio_suspend()** immediately returns success. Any **NULL aiocb** in **list** is silently ignored. If all of the the **aiocbs** in **list** are **NULL**, the **aio_suspend()** immediately returns success.

ERRORS

If **aio_suspend()** detects one of the following error conditions, **errno** is set to the indicated value:

[EAGAIN] System-wide or per-process resources were not available to process the request.

[EAGAIN] The time interval specified in the **timespec** referenced by *timeout* passed before any of the asynchronous I/O operations enqueued for one of the **aiocb** entries referenced in **list** completed.

[EINVAL] The value of the **nent** argument was negative or exceeded the maximum value allowed. The maximum value allowed can be obtained using the **sysconf()** call with the argument **_SC_AIO_MAX**.

[EINVAL] One or more of the **aiocb** pointers in **list** does not identify an asynchronous operation enqueued by **aio_read()**, **aio_write()**, or **lio_listio()**, and for which **aio_return()** has not yet been called. **aiocb** pointers associated with **aio_fsync()** will yield this error.

[EINTR] A signal was delivered to the process while **aio_suspend()** was waiting. Completion of asynchronous operations can cause signal delivery.

SEE ALSO

aio_cancel(2), aio_error(2), aio_fsync(2), aio_read(2), aio_return(2), aio_write(2), lio_listio(2), suspend(2), aio(5).

STANDARDS CONFORMANCE

aio_suspend(): POSIX Realtime Extensions, IEEE Std 1003.1b

NAME

aio_write() - start asynchronous write operation

SYNOPSIS

```
#include <aio.h>

int aio_write(struct aiocb *aiocbp);
```

DESCRIPTION

The **aio_write()** function allows the calling process to perform an asynchronous write to a previously opened file. The function call returns when the write operation has been enqueued for processing. Once enqueued, processing of the write operation may proceed concurrently with execution of the calling process thread.

If an error condition is detected that prevents the write request from being enqueued, **aio_write()** returns **-1** and sets **errno** to indicate the cause of the failure. Once the write operation has been successfully enqueued, an **aio_error()** and **aio_return()** function referencing the **aiocb** referred to by *aiocbp* must be used to determine its status and any error conditions, including those normally reported by **write()**. The request remains enqueued and consumes process and system resources until **aio_return()** is called.

The **aio_write()** function allows the calling process to write **aiocbp->aio_nbytes** to the file associated with **aiocbp->aio_fildes** from the buffer pointed to by **aiocbp->aio_buf**. The priority of the write operation is reduced by the value of **aiocbp->aio_reqprio**, which must be a value between 0 (zero) and a maximum value which can be obtained using the **sysconf()** call with the argument **_SC_AIO_PRIO_DELTA_MAX**. A value of 0 (zero) yields no reduction in priority. The **aiocbp->aio_lio_opcode** field is ignored.

When the **O_APPEND** flag is *not* set for the file, the write operation takes place at the absolute position in the file given by **aiocbp->aio_offset**, as if **lseek()** were called immediately prior to the operation with **offset** equal to **aiocbp->aio_offset** and **whence** set to **SEEK_SET**. When the **O_APPEND** flag *is* set for the file, **aiocbp->aio_offset** is ignored, and asynchronous write operations append to the file in the same order as the requests were enqueued. The value of the file offset is never changed by asynchronous I/O operations.

Altering the contents of, or deallocating memory associated with the **aiocb** referred to by *aiocbp* or the buffer referred to by **aiocbp->aio_buf** while an asynchronous write operation is outstanding may produce unpredicatable results because **aio_return()** has not been called for by **aiocb**.

If **aiocbp->aio_sigevent** is a valid signal event structure, then the designated signal will be delivered when the requested asynchronous write operation completes.

To use this function, link in the realtime library by specifying **-lrt** on the compiler or linker command line.

RETURN VALUE

aio_write() returns the following values:

- **0** Successful completion, the operation has been enqueued.
- **-1** Failure. The requested operation was not enqueued. **errno** is set to indicate the error.

The return value from **aio_write()** reflects the success or failure of enqueuing the requested write operation for asynchronous processing. **aio_write()** fails if an error in the function call is immediately detected, or if system resource limits prevent the request from being enqueued. All other error conditions are reported asynchronously and must be retrieved with **aio_error()** and **aio_return()**.

ERRORS

If **aio_write()** detects one of the following error conditions, **errno** is set to the indicated value:

[EAGAIN] The request could not be queued either because of a resource shortage or because the per-process or system-wide limit on asynchronous I/O operations or asynchronous threads would have been exceeded.

[EEXIST] The *aiocbp* is already in use for another asynchronous I/O operation.

Once the write operation request has been enqueued by **aio_write()**, all of the errors normally reported by the **write()** function and the following errors may be reported asynchronously and returned in a subsequent call to **aio_error()** or **aio_return()** referencing the **aiocb** supplied in the successful **aio_write()** call.

[EBADF] The **aiocbp->aio_fildes** was not a valid file descriptor open for writing.

[EINVAL] The **aiocb->aio_sigevent** is not a valid address in the process virtual address space.

[EINVAL] The parameters of the indicated **sigevent** in **aiocb->aio_sigevent** are invalid.

[EINVAL] The value of **aiocbp->aio_reqprio** is not valid.

[EINVAL] The value of **aiocbp->aio_nbytes** is invalid.

[EINVAL] The file offset implied by **aiocbp->aio_offset** or **aiocbp->aio_offset**+ **aiocbp->aio_nbytes** are not valid for the file at the time the request is processed.

[ECANCELED] The write operation was canceled due to a subsequent call to **aio_cancel()** referencing the same **aiocb** that was used to start the operation.

EXAMPLE

The following code sequence and call to **aio_write()** starts an asynchronous write operation.

```
#include <fcntl.h>
#include <errno.h>
#include <aio.h>
char buf[4096];
int retval; ssize_t nbytes;
struct aiocb myaiocb;
bzero( &myaiocb, sizeof (struct aiocb));
bzero( &buf, sizeof (buf));
myaiocb.aio_fildes = open( "/dev/null", O_RDWR);
myaiocb.aio_offset = 0;
myaiocb.aio_buf = (void *) buf;
myaiocb.aio_nbytes = sizeof (buf);
myaiocb.aio_sigevent.sigev_notify = SIGEV_NONE;
retval = aio_write( &myaiocb );
if (retval) perror("aio_write:");
/* continue processing */
 ...
/* wait for completion */
while ( (retval = aio_error( &myaiocb) ) == EINPROGRESS) ;
/* free the aiocb */
nbytes = aio_return( &myaiocb);
```

SEE ALSO

aio_cancel(2), aio_error(2), aio_fsync(2), aio_read(2), aio_return(2), aio_suspend(2), lio_listio(2), write(2), aio(5).

STANDARDS CONFORMANCE

aio_write(): POSIX Realtime Extensions, IEEE Std 1003.1b

a

NAME

alarm - set a process's alarm clock

SYNOPSIS

```
#include <unistd.h>

unsigned int alarm(unsigned int 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 on the maximum supported alarm time. 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()** 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(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

a

NAME

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

SYNOPSIS

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

int audctl(int cmd, char *cpath, char *npath, mode_t 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* was specified while 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 is 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).

a

NAME

audswitch - suspend or resume auditing on the current process

SYNOPSIS

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

int audswitch(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** (see *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).

a

NAME

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

SYNOPSIS

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

int audwrite(const 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

bind - bind an address to a socket

SYNOPSIS

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

AF_CCITT only

```
#include <x25/x25addrstr.h>
```

AF_INET and AF_VME_LINK only

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

AF_UNIX only

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

int bind(int s, const void *addr, int addrlen);
```

_XOPEN_SOURCE_EXTENDED only (UNIX 98)

```
int bind(int s, const struct sockaddr *addr, socklen_t addrlen);
```

Obsolescent _XOPEN_SOURCE_EXTENDED only (UNIX 95)

```
int bind(int s, const struct sockaddr *addr, size_t addrlen);
```

DESCRIPTION

The **bind()** system call assigns an address to an unbound socket. When a socket is created with **socket()**, it exists in an address space (address family) but has no address assigned. **bind()** causes the socket whose descriptor is *s* to become bound to the address specified in the socket address structure pointed to by *addr*.

addrlen must specify the size of the address structure. Since the size of the socket address structure varies between socket address families, the correct socket address structure should be used with each address family (for example, **struct sockaddr_in** for AF_INET and AF_VME_LINK, and **struct sockaddr_un** for AF_UNIX). Typically, the **sizeof()** function is used to pass this value in the **bind()** call (for example, **sizeof(struct sockaddr_in)**).

The rules used in address binding vary between communication domains. For example, when binding an AF_UNIX socket to a path name (such as **/tmp/mysocket**), an open file having that name is created in the file system. When the bound socket is closed, that file still exists unless it is removed or unlinked. When binding an AF_INET socket, *sin_port* can be a port number or it can be zero. If *sin_port* is zero, the system assigns an unused port number automatically.

AF_VME_LINK Only

The **bind()** system call is used only by servers and not clients.

RETURN VALUE

bind() returns the following values:

- **0** Successful completion.
- **-1** Failure. **errno** is set to indicate the error.

ERRORS

If **bind()** fails, **errno** is set to one of the following values.

[EACCES]	The requested address is protected, and the current user has inadequate permission to access it. (This error can be returned by AF_INET only.)
[EADDRINUSE]	The specified address is already in use.
[EADDRNOTAVAIL]	The specified address is invalid or not available from the local machine, or for AF_CCITT sockets which use "wild card" addressing, the specified address space overlays the address space of an existing bind.
[EAFNOSUPPORT]	The specified address is not a valid address for the address family of this socket.
[EBADF]	*s* is not a valid file descriptor.

[EDESTADDRREQ] No *addr* parameter was specified.

[EFAULT] *addr* is not a valid pointer.

[EINVAL] The socket is already bound to an address, the socket has been shut down, *addrlen* is a bad value, or an attempt was made to **bind()** an AF_UNIX socket to an NFS-mounted (remote) name.

AF_CCITT: The protocol-ID length is negative or greater than 8, the X.121 address string contains an illegal character, or the X.121 address string is greater than 15 digits long.

AF_VME_LINK: An explicit bind can be made only to a well-known port.

[ENETDOWN] The *x25ifname* field name specifies an interface that was shut down, or never initialized, or whose Level 2 protocol indicates that the link is not working: Wires might be broken, the interface hoods on the modem are broken, the modem failed, the phone connection failed (this error can be returned by AF_CCITT only), noise interfered with the line for a long period of time.

[ENETUNREACH] The X.25 Level 2 protocol is down. The X.25 link is not working: Wires might be broken, or connections are loose on the interface hoods at the modem, the modem failed, or noise interfered with the line for an extremely long period of time.

[ENOBUFS] No buffer space is available. The **bind()** cannot complete.

[ENOMEM] No memory is available. The **bind()** cannot complete.

[ENODEV] The *x25ifname* field name specifies a nonexistent interface. (This error can be returned by AF_CCITT only.)

[ENOTSOCK] *s* is a valid file descriptor, but it is not a socket.

[EOPNOTSUPP] The socket referenced by *s* does not support address binding.

[EISCONN] The connection is already bound. (AF_VME_LINK.)

b

OBSOLESCENCE

Currently, the **socklen_t** and **size_t** types are the same size. This is compatible with both the UNIX 95 and UNIX 98 profiles. However, in a future release, **socklen_t** might be a different size, but that should not adversely affect application behavior in this case. Applications may use **socklen_t** now. But applications that need to be portable to the UNIX 95 profile should follow the X/Open specification (see *xopen_networking*(7)).

FUTURE DIRECTION

Currently, the default behavior is the **HP-UX BSD Sockets**; however, it might be changed to **X/Open Sockets** in a future release. At that time, any **HP-UX BSD Sockets** behavior that is incompatible with **X/Open Sockets** might be obsoleted. Applications that conform to the X/Open specification now will avoid migration problems (see *xopen_networking*(7)).

MULTITHREAD USAGE

The **bind()** system call is thread-safe. It has a cancellation point; and it is async-cancel safe, async-signal safe, and fork-safe.

AUTHOR

bind() was developed by HP and the University of California, Berkeley.

SEE ALSO

connect(2), getsockname(2), listen(2), socket(2), af_ccitt(7F), af_vme_link(7F), inet(7F), socketx25(7), tcp(7P), udp(7P), unix(7P), xopen_networking(7).

STANDARDS CONFORMANCE

bind(): XPG4

b

NAME

brk, sbrk - change data segment space allocation

SYNOPSIS

```
#include <unistd.h>

int brk(const void *endds);

void *sbrk(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()** 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.

Be very careful when using either **brk** or **sbrk** in conjunction with calls to the *malloc*(3C) library routines. There is only one program data segment from which all three of these routines allocate and deallocate program data memory.

RETURN VALUE

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

Upon successful completion, **sbrk()** returns the old break value. Otherwise, **SBRK_FAILED** is returned and **errno** is set to indicate the error. The symbol **SBRK_FAILED** is defined in the header <**unistd.h**>. No successful return from **sbrk()** will return the value **SBRK_FAILED**.

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

chdir, fchdir - change working directory

SYNOPSIS

```
#include <unistd.h>

int chdir(const char *path);

int fchdir(int fildes);
```

DESCRIPTION

chdir() and **fchdir()** cause a directory pointed to by *path* or *fildes* to become the current working directory, the starting point for path searches of path names not beginning with **/**. *path* points to the path name of a directory. *fildes* is an open file descriptor of a directory.

For a directory to become the current working directory, a process must have execute (search) access to the 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

chdir() fails and the current working directory remains 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 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 were encountered in translating the path name.

fchdir() fails and the current working directory remains unchanged if one or more of the following are true:

[EACCES]	Search permission is denied for *fildes*.
[EBADF]	*fildes* is not an open file descriptor.
[ENOTDIR]	The open file descriptor *fildes* does not refer to a directory.

AUTHOR

chdir() and **fchdir()** were developed by AT&T Bell Laboratories and HP.

SEE ALSO

cd(1), chroot(2).

STANDARDS CONFORMANCE

chdir(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

chmod(), fchmod() - change file mode access permissions

SYNOPSIS

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

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

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

DESCRIPTION

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

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 *mode* argument. The value of *mode* is the bit-wise 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.

To change the mode of a file, the effective user ID of the process must match that of the owner of the file or a user with appropriate privileges.

If the effective user ID of the process is not that of a user with appropriate privileges, mode bit **S_ISVTX** is cleared.

If the effective user ID of the process is not that of a user with appropriate privileges, 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, mode bit **S_ISGID** is cleared.

The mode bit **S_ENFMT** (same as **S_ISGID**) is 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()**, **creat()**, **read()**, and **write()** on such files (see *open*(2), *creat*(2), *read*(2), and *write*(2)).

If an executable file is prepared for sharing, mode bit **S_ISVTX** 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.

If the path given to **chmod()** contains a symbolic link as the last element, this link is traversed and path name resolution continues. **chmod()** changes the access mode of the symbolic link's target, rather than the access mode of the link.

Access Control Lists - HFS File Systems Only

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()** and **setacl()** (see *getacl*(2) and *setacl*(2)).

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

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

RETURN VALUE

chmod() returns the following values:

- **0** Successful completion.

-1 Failure. **errno** is set to indicate the error.

ERRORS

If **chmod()** or **fchmod()** fails, the file mode is unchanged. **errno** is set to one of the following values.

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

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

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

[EINVAL] *path* or *fildes* descriptor does not refer to an appropriate file. It may be a special file, such as a pipe or socket.

[ELOOP] Too many symbolic links were 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* or 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 a user with appropriate privileges.

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

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), read(2), lockf(2), mknod(2), open(2), setacl(2), write(2), acl(5).

STANDARDS CONFORMANCE

chmod(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

fchmod(): AES, SVID3

C

C

NAME

chown(), fchown(), lchown() - change owner and group of a file

SYNOPSIS

```
#include <unistd.h>

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

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

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

DESCRIPTION

The **chown()** system call changes the user and group ownership of a file. *path* points to the path name of a file. **chown()** sets 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 **UID_MAX** (see *limits*(5)).

Only processes with an effective user ID equal to the file owner or a user having appropriate privileges can change the ownership of a file. If privilege groups are supported, the owner of a file can change the ownership only as a member of a privilege group allowing CHOWN, as set up by the **setprivgrp** command (see *setprivgrp*(1M)). All users get the CHOWN privilege 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 has the CHOWN privilege, the file can be given to any group.

If **chown()** is invoked on a regular file by anyone 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.

If the path given to **chown()** contains a symbolic link as the last element, this link is traversed and path name resolution continues. **chown()** changes the owner and group of the symbolic link's target, rather than the owner and group of the link.

The **fchown()** system call functions exactly like **chown()**, exept that it operates on a file descriptor instead of a path name. *fildes* is a file descriptor.

The **lchown()** system call sets the owner ID and group ID of the named file just as **chown()** does, except in the case where the named file is a symbolic link. In this case, **lchown()** changes the owner and group of the symbolic link file itself.

Access Control Lists - HFS File Systems Only

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**() in conjunction with ACLs, if the new owner and/or group does not have an optional ACL entry corresponding to *user*.**%** and/or **%.***group* 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 *user*.**%** and/or **%.***group*, **chown()** sets the file's permission bits (and the three basic ACL entries) to the permissions contained in that entry.

RETURN VALUE

chown() and **fchown()** return the following values:

- **0** Successful completion.
- **-1** Failure. The owner and group of the file remain unchanged. **errno** is set to indicate the error.

ERRORS

If **chown()** or **fchown()** fails, **errno** is set to one of the following values:

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

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

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

[EINVAL] Either *owner* or *group* is greater than or equal to **UID_MAX**, or is an illegal negative value.

[ELOOP] Too many symbolic links were 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] The file named by *path* does not exist.

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

[EPERM] The effective user ID is not a user having appropriate privileges 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, the owner of the file is not a member of a privilege group allowing the CHOWN privilege.
- When changing the group of the file, 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.

C

AUTHOR

chown() was developed by AT&T.

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(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

fchown(): AES, SVID3

C

NAME

chroot - change root directory

SYNOPSIS

```
#include <unistd.h>

int chroot(const char *path);
```

DESCRIPTION

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

The effective user ID of the process must be a user having appropriate privileges 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() fails and the root directory remains unchanged if one or more of the following is 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 a user who has appropriate privileges.
[EFAULT]	*path* points outside the allocated address space of the process. 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.
[ELOOP]	Too many symbolic links were encountered in translating the path name.

SEE ALSO

chroot(1M), chdir(2).

STANDARDS CONFORMANCE

chroot(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

NAME

clock_settime(), clock_gettime(), clock_getres() - clock operations

SYNOPSIS

```
#include <time.h>

int clock_settime(
     clockid_t clock_id,
     const struct timespec *tp
);

int clock_gettime(
     clockid_t clock_id,
     struct timespec *tp
);

int clock_getres(
     clockid_t clock_id,
     struct timespec *res
);
```

DESCRIPTION

clock_settime()

The **clock_settime()** function sets the specified clock, **clock_id**, to the value specified by **tp**. Time values that are between two consecutive non-negative integer multiples of the resolution of the specified clock are truncated down to the smaller multiple of the resolution.

clock_gettime()

The **clock_gettime()** function returns the current value **tp** for the specified clock, **clock_id**.

clock_getres()

The resolution of any clock can be obtained by calling **clock_getres()**. Clock resolutions are implementation defined and are not settable by a process. If the argument **res** is not NULL, the resolution of the specified clock is stored into the location pointed to by **res**. If **res** is NULL, the clock resolution is not returned.

A clock may be system wide, that is, visible to all processes; or per-process, measuring time that is meaningful only within a process.

The following clocks are supported:

CLOCK_REALTIME
: This clock represents the realtime clock for the system. For this clock, the values returned by **clock_gettime()** and specified by **clock_settime()** represent the amount of time (in seconds and nanoseconds) since the Epoch. It is a system wide clock. Appropriate privileges are required to set this clock.

CLOCK_VIRTUAL
: This clock represents the amount of time (in seconds and nanoseconds) that the calling process has spent executing code in the user's context. It is a per-process clock. It cannot be set by the user.

CLOCK_PROFILE
: This clock represents the amount of time (in seconds and nanoseconds) that the calling process has spent executing code in both the user's context and in the operating system on behalf of the calling process. It is a per-process clock. It cannot be set by the user.

RTTIMER0 RTTIMER1
: These clocks are high resolution hardware clocks present on HP-RT realtime systems. It is included here so that applications accessing this hardware can be compiled on HP-UX systems and then ported to an HP-RT target. HP-UX does not support **RTTIMER0** or **RTTIMER1**.

RETURN VALUE

A return of zero indicates that the call succeeded. A return value of −1 indicates that an error occurred, and **errno** is set to indicate the error.

C

ERRORS

If any of the following conditions occur, the **clock_settime()**, **clock_gettime()**, and **clock_getres()** functions return −1 and set **errno** (see *errno*(2)) to the corresponding value:

[ENOSYS] The functions **clock_settime()**, **clock_gettime()**, and **clock_getres()** are not supported by this implementation.

[EINVAL] The **clock_id** argument does not specify a known clock.

[EINVAL] The **tp** argument to **clock_settime()** is outside the range for the given **clock_id**.

[EINVAL] The **tp** argument specified a nanosecond value less than zero or greater than or equal to 1000 million.

[EPERM] The requesting process does not have the necessary privileges to set the specified clock.

[EFAULT] The **tp** or **res** argument points to an invalid address.

EXAMPLES

Advance the system wide realtime clock approximately one hour:

```
#include <time.h>
#include <errno.h>

struct timespec cur_time, new_time;

if (clock_gettime(CLOCK_REALTIME, &cur_time)) {
     perror("clock_gettime(CLOCK_REALTIME) failed");
     exit(1);
}
new_time.tv_sec = cur_time.tv_sec + 3600;
new_time.tv_nsec = cur_time.tv_nsec;
if (clock_settime(CLOCK_REALTIME, &new_time)) {
     perror("clock_settime(CLOCK_REALTIME) failed");
     exit(2);
}
```

Get the resolution of the user profiling clock:

```
#include <time.h>
#include <errno.h>

struct timespec resolution;

if (clock_getres(CLOCK_PROFILE, &resolution)) {
     perror("clock_getres(CLOCK_PROFILE) failed");
     exit(1);
}
(void)printf("Resolution of user profiling clock is:\n");
(void)printf("%d seconds and %d nanoseconds.\n",
          resolution.tv_sec, resolution.tv_nsec);
```

AUTHOR

clock_settime(), **clock_gettime()**, and **clock_getres()** were derived from the proposed IEEE POSIX P1003.4 Standard, Draft 14.

SEE ALSO

timers(2).

STANDARDS CONFORMANCE

clock_getres(): POSIX.4

clock_gettime(): POSIX.4

clock_settime(): POSIX.4

C

NAME

close - close a file descriptor

SYNOPSIS

```
#include <unistd.h>

int close(int fildes);
```

DESCRIPTION

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

RETURN VALUE

Upon successful completion, **close()** returns a value of 0; otherwise, it returns −1 and sets **errno** to indicate the error.

ERRORS

close() fails if the any of following conditions are encountered:

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

[EINTR] An attempt to close a slow device or connection was interrupted by a signal. The file descriptor still points to an open device or connection.

[ENOSPC] Not enough space on the file system. This error can occur when closing a file on an NFS file system. [When a **write()** system call is executed on a local file system and if a new buffer needs to be allocated to hold the data, the buffer is mapped onto the disk at that time. A full disk is detected at this time and **write()** returns an error. When the **write()** system call is executed on an NFS file system, the new buffer is allocated without communicating with the NFS server to see if there is space for the buffer (to improve NFS performance). It is only when the buffer is written to the server (at file close or the buffer is full) that the disk-full condition is detected.]

SEE ALSO

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

STANDARDS CONFORMANCE

close(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

connect - initiate a connection on a socket

SYNOPSIS

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

AF_CCITT only

```
#include <x25/x25addrstr.h>
```

AF_INET and AF_VME_LINK only

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

AF_UNIX only

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

int connect(int s, const void *addr, int addrlen);
```

_XOPEN_SOURCE_EXTENDED only (UNIX 98)

```
int connect(int s, const struct sockaddr *addr, socklen_t addrlen);
```

Obsolescent _XOPEN_SOURCE_EXTENDED only (UNIX 95)

```
int connect(int s, const struct sockaddr *addr, size_t addrlen);
```

DESCRIPTION

The **connect()** function initiates a connection on a socket.

s is a socket descriptor.

addr is a pointer to a socket address structure containing the address of a remote socket to which a connection is to be established.

addrlen is the size of this address structure. Since the size of the socket address structure varies among socket address families, the correct socket address structure should be used with each address family (for example, **struct sockaddr_in** for AF_INET and AF_VME_LINK and **struct sockaddr_un** for AF_UNIX). Typically, the **sizeof()** function is used to pass this value (for example, **sizeof(struct sockaddr_in)**).

If the socket is of type **SOCK_DGRAM**, **connect()** specifies the peer address to which messages are to be sent, and the call returns immediately. Furthermore, this socket can only receive messages sent from this address.

If the socket is of type **SOCK_STREAM**, **connect()** attempts to contact the remote host to make a connection between the remote socket (peer) and the local socket specified by *s*. The call normally blocks until the connection completes. If nonblocking mode has been enabled with the **O_NONBLOCK** or **O_NDELAY** **fcntl()** flags or the **FIOSNBIO** **ioctl()** request and the connection cannot be completed immediately, **connect()** returns an error as described below. In these cases, **select()** can be used on this socket to determine when the connection has completed by selecting it for writing.

The **connect()** system call may complete if remote program has a pending **listen()** even though remote program had not yet issued an **accept()** system call.

O_NONBLOCK and **O_NDELAY** are defined in **<sys/fcntl.h>** and explained in *fcntl*(2), *fcntl*(5), and *socket*(7). **FIOSNBIO** is defined in **<sys/ioctl.h>** and explained in *ioctl*(2), *ioctl*(5), and *socket*(7).

If *s* is a **SOCK_STREAM** socket that is bound to the same local address as another **SOCK_STREAM** socket, **connect()** returns [EADDRINUSE] if *addr* is the same as the peer address of that other socket. This situation can only happen if the **SO_REUSEADDR** option has been set on *s*, which is an AF_INET socket (see *getsockopt*(2)).

If the AF_INET socket does not already have a local address bound to it (see *bind*(2)), **connect()** also binds the socket to a local address chosen by the system.

An AF_VME_LINK socket always binds the socket to a local address chosen by the system.

Generally, stream sockets may successfully connect only once; datagram sockets may use **connect()** multiple times to change the peer address. For datagram sockets, a side effect of attempting to connect to some invalid address (see ERRORS below) is that the peer address is no longer maintained by the system. An example of an invalid address for a datagram socket is *addrlen* set to 0 and *addr* set to any value.

AF_CCITT Only

Use the **x25addrstr** struct for the address structure. The caller must know the X.121 address of the DTE to which the connection is to be established, including any subaddresses or protocol IDs that may be needed. Refer to *af_ccitt*(7F) for a detailed description of the **x25addrstr** address structure. If address-matching by protocol ID, specify the protocol ID with the **X25_WR_USER_DATA ioctl()** call before issuing the **connect()** call. The **X25_WR_USER_DATA ioctl()** call is described in *socketx25*(7).

C

DEPENDENCIES

AF_CCITT

The **SO_REUSEADDR** option to **setsockopt()** is not supported for sockets in the AF_CCITT address family.

RETURN VALUE

connect() returns the following values:

- **0** Successful completion.
- **-1** Failure. **errno** is set to indicate the error.

ERRORS

If **connect()** fails, **errno** is set to one of the following values.

[EADDRINUSE]	The specified address is already in use. For datagram sockets, the peer address is no longer maintained by the system.
[EADDRNOTAVAIL]	The specified address is not available on this machine, or the socket is a TCP/UDP socket and the zero port number is specified. For datagram sockets, the peer address is no longer maintained by the system.
[EAFNOSUPPORT]	The specified address is not a valid address for the address family of this socket. For datagram sockets, the peer address is no longer maintained by the system.
[EALREADY]	Nonblocking I/O is enabled with **O_NONBLOCK**, **O_NDELAY**, or **FIOSNBIO**, and a previous connection attempt has not yet completed.
[EBADF]	*s* is not a valid file descriptor.
[ECONNREFUSED]	The attempt to connect was forcefully rejected.
[EFAULT]	*addr* is not a valid pointer.
[EINPROGRESS]	Nonblocking I/O is enabled using **O_NONBLOCK**, **O_NDELAY**, or **FIOSNBIO**, and the connection cannot be completed immediately. This is not a failure. Make the **connect()** call again a few seconds later. Alternatively, wait for completion by calling **select()** and selecting for write.
[EINTR]	The connect was interrupted by a signal before the connect sequence was complete. The building of the connection still takes place, even though the user is not blocked on the **connect()** call.
[EINVAL]	The socket has already been shut down or has a **listen()** active on it; *addrlen* is a bad value; an attempt was made to **connect()** an AF_UNIX socket to an NFS-mounted (remote) name; the X.121 address length is zero, negative, or greater than 15 digits. For datagram sockets, if *addrlen* is a bad value, the peer address is no longer maintained by the system.
[EISCONN]	The socket is already connected.
[ENETDOWN]	The X.25 interface specified in the *addr* struct was found but was not in the initialized state. *x25ifname* field name is an interface which has been shut down or never initialized or suffered a power failure which erased its state information.

C

[ENETUNREACH]	The network is not reachable from this host. For AF_CCITT only: X.25 Level 2 is down. The X.25 link is not working: wires might be broken, connections are loose on the interface hoods at the modem, the modem failed, or noise interfered with the line for an extremely long period of time.
[ENOBUFS]	No buffer space is available. The **connect()** has failed.
[ENOMEM]	No memory is available. The **connect()** has failed.
[ENODEV]	The *x25ifname* field refers to a nonexistent interface.
[ENOSPC]	All available virtual circuits are in use.
[ENOTSOCK]	*s* is a valid file descriptor, but it is not a socket.
[EOPNOTSUPP]	The socket referenced by *s* does not support **connect()**. With X.25 an attempt was made to issue a **connect()** call on a **listen()** socket.
[ETIMEDOUT]	Connection establishment timed out without establishing a connection. One reason could be that the connection requests queue at the remote socket may be full (see **listen(2)**).

OBSOLESCENCE

Currently, the **socklen_t** and **size_t** types are the same size. This is compatible with both the UNIX 95 and UNIX 98 profiles. However, in a future release, **socklen_t** might be a different size, but that should not adversely affect application behavior in this case. Applications may use **socklen_t** now. But applications that need to be portable to the UNIX 95 profile should follow the X/Open specification (see *xopen_networking*(7)).

FUTURE DIRECTION

Currently, the default behavior is the **HP-UX BSD Sockets**; however, it might be changed to **X/Open Sockets** in a future release. At that time, any **HP-UX BSD Sockets** behavior that is incompatible with **X/Open Sockets** might be obsoleted. Applications that conform to the X/Open specification now will avoid migration problems (see *xopen_networking*(7)).

MULTITHREAD USAGE

The **connect()** system call is thread-safe. It has a cancellation point; and it is async-cancel safe, async-signal safe, and fork-safe.

AUTHOR

connect() was developed by HP and the University of California, Berkeley.

SEE ALSO

accept(2), getsockname(2), select(2), socket(2), af_ccitt(7F), af_vme_link(7F), socket(7), socketx25(7), xopen_networking(7).

C

NAME

crashconf() - configure system crash dumps

SYNOPSIS

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

int crashconf(
int    operation,
int    includeClasses,
int    excludeClasses,
int    deviceCount,
char **devices,
int   *deviceReturn
);
```

DESCRIPTION

crashconf() changes the current system crash dump configuration. The crash dump configuration consists of three lists:

- The *crash dump device* list. This list identifies all devices that can be used to store a crash dump. The devices are used in reverse order, last specified to first.
- The *included class* list. This list identifies all system memory classes that *must* be included in any crash dump.
- The *excluded class* list. This list identifies all system memory classes that *should not* be included in a crash dump.

Most system memory classes are in neither the included class list nor the excluded class list. Instead, the system determines whether or not to dump those classes of memory based on the type of crash that occurs.

Note that certain types of system crash, such as TOC's, require a full crash dump. Also, the system operator may request a full crash dump at the time the dump is taken. In either of these cases, a full dump will be performed regardless of the contents of the excluded class list.

Configuration changes made using **crashconf()** take effect immediately and remain in effect until the next system reboot, or until changed with a subsequent call to **crashconf()**.

Parameters

operation is a bitmask specifying what **crashconf()** should do. It must have at least one of the following flags set:

DC_INCLUDE **crashconf()** will change the contents of the included class list. The *includeClasses* parameter is valid.

DC_EXCLUDE **crashconf()** will change the contents of the excluded class list. The *excludeClasses* parameter is valid.

DC_DEVICES **crashconf()** will change the contents of the crash dump device list. The *deviceCount*, *devices* and *deviceReturn* parameters are valid.

operation may also have the following flag set:

DC_REPLACE Changes to any of the lists will replace the current contents of those lists. Without this flag, changes will add to the current contents of those lists.

includeClasses is a bitmask of classes that must be dumped. If it is set to **DT_ALL**, all dumps will be full dumps. Other allowed values are described under *Classes*, below.

excludeClasses is a bitmask of classes that may not be dumped unless a full dump is required (due to the cause of the dump, or by explicit operator request). If it is set to **DT_ALL**, dumps will be disabled. Other allowed values are described under *Classes*, below.

devices is an array of *deviceCount* pathnames of block device files for crash dump devices. To be valid, a device must be accessible and must not contain a file system. Where LVM partitions are in use, the device number must be for a partition, not the physical disk that contains it, and must represent a partition that is strictly contiguous on the physical disk. (LVM bad-block reallocation, and striping features may not be in use on the partition.) Depending on the disk type, the dump space may be restricted to the first 2GB or 4GB of the physical disk.

deviceReturn is an array of *deviceCount* integers for returning the results of attempting to configure the corresponding device from the *devices* array. Upon return, each element is set to a numeric value indicating the result of configuring the corresponding device as follows:

0 Successfully configured the corresponding device as a dump device.

< 0 Failed to configure the corresponding device as a dump device. The absolute value of the returned number can be used as an index into an array of error messages. The error message strings are defined in **CCERR_STRINGS** (see below).

> 0 *Warning*, The corresponding device has been configured but there is one or more notes or warnings associated with the device. The returned value is a bitmap of warnings. the warning message strings are defined in **CCWARN_STRINGS** (see below).

Any parameters which are not used for the given *operation* should be set to zero. Note that both *devices* and *deviceReturn* must be specified if **DC_DEVICES** is specified.

C

Classes

The following system memory classes have been defined as of this writing. Refer to the output of *crashconf*(1M) or to **/usr/include/sys/crashconf.h** for definitions of any classes added since publication.

DT_UNUSED	Unused physical memory pages
DT_KCODE	Kernel code pages
DT_BCACHE	Buffer cache data pages
DT_KSDATA	Kernel static data pages
DT_KDDATA	Kernel dynamic data pages
DT_FSDATA	File system metadata pages
DT_USTACK	User process stack pages
DT_UAREA	U-Area pages
DT_USERPG	User process pages

EXAMPLES

The following examples demonstrate the usage of **crashconf()**.

Example 1: Adding a Crash Dump

```
char *device_to_add[1];

int device_return[1];

...

crashconf(DC_DEVICES, 0, 0, 1, device_to_add, device_return);
```

Example 2: Force Dumping of Buffer Cache

```
crashconf(DC_INCLUDE, DT_BCACHE, 0, 0, NULL, NULL);
```

Example 3: Disable Dumps

```
crashconf(DC_EXCLUDE | DC_REPLACE, 0, DT_ALL, 0, NULL, NULL);
```

Example 4: Using CCERR_STRINGS and CCWARN_STRINGS

Assume only one device, *devices*[0], is being added to the dump configuration. The following code will check the *device_return*[0] value and print corresponding error or warning messages.

```
char *ccerrs[] = {
  CCERR_STRINGS
};
int num_ccerrs = sizeof(ccerrs)/sizeof(*ccerrs);
char *ccwarns[] = {
CCWARN_STRINGS
};
int num_ccwarns = sizeof(ccwarns)/sizeof(*ccwarns);
char *device_to_add[1];
int   device_return[1];
...
crashconf(DC_DEVICES, 0, 0, 1, device_to_add, device_return);
if (device_return[0] < 0) {
```

```
        if (device_return[0] > -num_ccerrs)
            fprintf(stderr, "%s: error: %s", device_to_add[0],
                    ccerrs[-device_return[0]]);
    } else if (device_return[0] > 0) {
        int warn_num;
         for (warn_num = 0; warn_num < NUM_CCWARNS; warn_num++)
            if (device_return[0] & (1 << warn_num))
                fprintf(stderr,"%s: warning: %s",
                        device_to_add[0], ccwarns[warn_num]);
    }
```


RETURN VALUE

Upon successful completion, zero is returned. Otherwise, a value of –1 is returned and **errno** is set to indicate the error. If **DC_DEVICES** is set, a one (1) may be returned to indicate that at least one device has been configured but one or more devices failed to configure.

ERRORS

crashconf() fails if one or more of the following is true:

[EPERM] The calling process does not have appropriate privileges.

[EINVAL] *operation* does not have at least one of **DC_INCLUDE**, **DC_EXCLUDE**, or **DC_DEVICES** set.

[EINVAL] *operation* has both **DC_INCLUDE** and **DC_EXCLUDE** set, and the same class (bit) is specified in both *includeClasses* and *excludeClasses*.

[EINVAL] *operation* has **DC_DEVICES** set, and *deviceCount* is less than zero or greater than **DC_MAXDEVICES**.

SEE ALSO

crashconf(1M), pstat_getcrashinfo(2), pstat_getcrashdev(2).

C

NAME

creat - create a new file or rewrite an existing one

SYNOPSIS

```
#include <fcntl.h>

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

DESCRIPTION

The **creat()** system call 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 file's group ID is set to the group ID of the parent directory. Otherwise, the file'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 is cleared (see *chmod*(2)).

If the system call is made in 64 bit mode, the O_LARGEFILE status flag is automatically set (see *fcntl*(5) or *open*(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)). Each process has a limit on how many files it can open simultaneously. Refer to *getrlimit*(2) for the open files limit. This is also discussed in *open*(2). A new file can be created with a mode that forbids writing.

Access Control Lists - HFS File Systems Only

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)).

RETURN VALUE

creat() returns the following values:

n Successful completion. *n* is the value of the file descriptor. It is nonnegative.
-1 Failure. **errno** is set to indicate the error.

ERRORS

If **creat()** fails, **errno** is set to one of the following values.

[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.

[EACCES] The file exists and write permission is denied.

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

[EDQUOT] User's disk quota block or inode limit has been reached for this file system.

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

[EISDIR] The named file is an existing directory.

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

[EMFILE] More than the maximum number of file descriptors 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).

[ENOSPC] Not enough space on the file system.

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

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

[EOVERFLOW] The named file is a regular file and the size of the file cannot be represented correctly in an object of type **`off_t`**.

[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.

SEE ALSO

chmod(2), close(2), creat64(2), dup(2), fcntl(2), lockf(2), lseek(2), open(2), open64(2), read(2), setacl(2), truncate(2), umask(2), write(2), acl(5).

STANDARDS CONFORMANCE

`creat()`: AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

C

NAME

creat64(), fstat64(), getrlimit64(), lockf64(), lseek64(), lstat64(), mmap64(), open64(), prealloc64(), setrlimit64(), stat64(), statvfs64(), truncate64() - non-POSIX standard API interfaces to support large files.

SYNOPSIS

```
#include <fcntl.h>
int creat64(const char *path, mode_t mode);
#include <sys/stat.h>
int fstat64(int fildes, struct stat64 *buf);
include <sys/resource.h>
int getrlimit64(int resource, struct rlimit64 *rlp);
#include <unistd.h>
int lock64(int *fildes, int function, off64_t size);
#include <unistd.h>
off64_t lseek64(int *fildes, off64_t offset, int whence);
#include <sys/stat.h>
int lstat64(const char *, struct stat64 *);
#include <sys/mmanstat.h>
void mmap64(void addr, size_t len, int prot, int flags, int fildes, off64_t off);
#include <fcntl.h>
int open64(const char *path, int oflag,...);
#include <fcntl.h>
int prealloc64(int fildes, off64_t size);
include <sys/resource.h>
int setrlimit64(int resource, const struct rlimit64 *rlp);
#include <sys/stat.h>
int stat64(const char *path, struct stat64 *buf);
#include <sys/statvfs.h>
int statvfs64(const char *path, struct statvfs64 *buf);
#include <unistd.h>
int truncate64(const char *path, off64_t length);
```

DESCRIPTION

New API's to support large files. These API interfaces are not a part of the POSIX standard and may be removed in the future.

creat64() The **creat64()** function is identical to **creat()** in 64-bit compile environment. Both functions set **O_LARGEFILE** in the file status flag to which the returned descriptor refers. **creat64()** function returns a file descriptor which can be used to grow the file past 2 GB if desired. All other functional behaviors, returns, and errors are identical to **creat()**.

fstat64() The **fstat64()** function is identical to **fstat()** except that **fstat64()** returns file status in a **struct stat64** instead of a **struct stat**. All other functional behaviors, returns, and errors are identical.

getrlimit64 The **getrlimit64()** function is identical to **getrlimit()** except that **getrlimit64()** passes a **struct rlimit64** as its second parameter instead of a **struct rlimit**. All other functional behaviors, returns, and errors are identical.

lock64() The **lock64()** function is identical to **lockf()** except that **lockf64()** accepts an **off64_t** for the size parameter instead of **off_t**. All other functional behaviors, returns, and errors are identical.

lseek64() The **lseek64()** function is identical to **lseek()** except that **lseek64()** accepts an **off64_t** type as the desired offset and has a return value of **off64_t.** All other functional behaviors, returns, and errors are identical.

lstat64() The **lstat64()** function is identical to **lstat()** except that **lstat64()** returns file status in a **struct stat64** instead of **struct stat**. All other functional behaviors, returns, and errors are identical.

mmap64() The **mmap64()** function is identical to **mmap()** except that **mmap64()** accepts the file offset as an **off64_t**.

open64() The **open64()** function is identical to **open()** in 64-bit compile environment. Both functions set **O_LARGEFILE** in the file status flag to which the returned descriptor refers. The **open64()** function is equivalent to **open()** function (in 32-bit compile environment) with **O_LARGEFILE** flag set. **open64()** function returns a file descriptor which can be used to grow the file past 2 GB if desired. All other functional behaviors, returns, and errors are identical to **open()**.

prealloc64() The **prealloc64()** function is identical to **prealloc()** except that **prealloc64()** accepts the file offset as an **off64_t**. All other functional behaviors, returns, and errors are identical to **prealloc()**.

setrlimit64 The **setrlimit64()** function is identical to **setrlimit()** except that **setrlimit64()** passes a **struct rlimit64** as its second parameter instead of a **struct rlimit.** All other functional behaviors, returns, and errors are identical.

stat64() The **stat64()** function is identical to **stat()** except that **stat64()** returns file status in a **struct stat64** instead of a **struct stat**.

statvfs64() Refer to **fstatvfs64()**.

truncate64() The **truncate64()** function is identical to **truncate()** except that **truncate64()** accepts the length parameter as an **off64_t** instead of **off_t**. All other functional behaviors, returns, and errors are identical to **truncate()**.

d

NAME

dup - duplicate an open file descriptor

SYNOPSIS

```
#include <unistd.h>

int dup(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.

RETURN VALUE

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

ERRORS

dup() fails if one or more of the following is true:

[EBADF]	*fildes* is not a valid open file descriptor.
[EMFILE]	Request violates the maximum number of open file descriptors.

AUTHOR

dup() was developed by AT&T and HP.

SEE ALSO

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

STANDARDS CONFORMANCE

dup(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME
dup2 - duplicate an open file descriptor to a specific slot

SYNOPSIS
```
#include <unistd.h>

int dup2(int fildes, int 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* refers to an already open file, the open file 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*(3C)) should not give the **-lBSD** option to *ld*(1).

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

ERRORS
dup2() fails 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.
[EINTR]	An attempt to close *fildes2* was interrupted by a signal. The file is still open.

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

STANDARDS CONFORMANCE
dup2(): AES, SVID2, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

d

e

NAME

errno - error indicator for function calls

SYNOPSIS

```
#include <errno.h>
```

DESCRIPTION

Many functions in the HP-UX operating system indicate an error condition by returning an otherwise out-of-range value (usually **-1**). Most of these functions set the symbol **errno,** that is defined in **errno.h,** to a nonzero code value that more specifically identifies the particular error condition that was encountered.

All errors detected and the corresponding error code values stored in **errno** are documented in the ERRORS section on manual pages for those functions that set it.

The value of **errno** is zero immediately after a successful call to any of the functions described by *exec*(2) and *ptrace*(2), but it is never set to zero by any other HP-UX function. Functions for which the use of **errno** is not described may nevertheless change its value to a nonzero value.

Since **errno** is *not* cleared on successful function calls, its value should be checked or used *only* when an error has been indicated and when the function's ERRORS section documents the error codes.

Applications should not attempt to take the address of **errno.** The practice of defining **errno** as **extern int errno** is obsolescent.

The following is a complete list of the error codes. The numeric values can be found in <**errno.h**> but they should not be used in an application program because they can vary from system to system.

[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] Resource temporarily unavailable. This is likely a temporary condition, and later calls to the same routine may complete normally.

[EALREADY] Operation already in progress. An operation was attempted on a nonblocking 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 nonsharable 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()** call (see *shutdown*(2)).

[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 is 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()** call (see *semop*(2)).

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

[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)).

[EILSEQ] Illegal byte sequence. A wide character code has been detected that does not correspond to a valid character, or a byte sequence does not form a valid wide character code.

[EINPROGRESS]
Operation now in progress. An operation that takes a long time to complete was attempted on a nonblocking 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 (such as unmounting a device that is not currently mounted, mentioning an undefined signal in **signal()** or **kill()**, or 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.

e

[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 can 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()** system call (see *pathconf*(2)).

[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 **open()**s 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 (such as 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 does not, or when one of the directories in a path name does not exist. It also occurs with **msgget()**, **semget()**, and **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.

[ENOLCK] System lock table is full. Too many files have file locks on them, or there are too many record locks on files, or there are too many instances of a reading or writing process sleeping until an enforcement mode lock clears. This error may also indicate system problems in handling a lock request on a remote NFS file. This error is also currently returned for all attempts to perform locking operations on a remote NFS file that has its locking enforcement mode bit set, since the stateless nature of NFS prevents maintaining the necessary lock information.

[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 can also occur 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()** or **setsockopt()** call (see *getsockopt*(2)).

[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()**, **semget()**, or **semop()** while **SEM_UNDO** flag is set.

[ENOSYM] Symbol does not exist in executable. The dynamic loader was unable to resolve a symbolic reference in a shared library during a call to one of the dynamic loader interface routines (see *shl_load*(3X). The program may be in an inconsistent state and should be terminated immediately.

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

[ENOTBLK] Block device required. A nonblock file was mentioned where a block device was required, such as 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 nondirectory was specified where a directory is required, such as in a path prefix or as an argument to **chdir()**.

[ENOTEMPTY] Directory not empty. An attempt was made to remove a nonempty directory.

[ENOTSOCK] Socket operation on nonsocket. An operation was attempted on something that is not a socket.

e

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

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

[EOPNOTSUPP] Operation not supported. The requested operation on a socket 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.

[EPERM] Not owner. Typically, this error indicates an attempt to modify a file in some way forbidden except to its owner or the superuser, such as to change its mode. It is also returned for attempts by ordinary users to do things for which they need, but lack, a special privilege.

[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. Data has been written to a pipe for which the other (reading) end has been closed. This most often occurs when the reading process exits before the writing process. This condition also generates the signal **SIGPIPE**; 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, ARPA Internet UDP protocol cannot be used 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()** 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()** 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 (timeout period varies, depending 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 an object in nonblocking 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 was previously deleted.

STANDARDS CONFORMANCE

errno: AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

execl(), execle(), execlp(), execv(), execve(), execvp() - execute a file

SYNOPSIS

```
#include <unistd.h>

extern char **environ;

int execl(const char *path,
          const char *arg0, ...
          /*
           *  [const char *arg1, ..., const char *argn,]
           *  (char *)0
           */
);

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

int execlp(const char *file,
           const char *arg0, ...
           /*
            *  [const char *arg1, ..., const char *argn,]
            *  (char *)0
            */
);

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

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

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

Remarks

The ANSI C "**, ...**" construct denotes a variable length argument list whose optional and required members are given in the associated comment (**/* */**).

DESCRIPTION

The **exec*()** system calls, in all their forms, load a program from an ordinary, executable file into 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 the latter case, the file of data is also called a script file.

If the calling process is multi-threaded, a call to any of the exec functions will cause all threads and light weight processes in the calling process to be terminated and the new executable image to be loaded and executed. No thread specific data destructor functions are called. If the exec function fails and returns to the caller, threads and light weight processes (LWPs) in the calling process will not be terminated.

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 POSIX shell (see *sh-posix*(1)) 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 (int argc, char **argv, char **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. 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 in the environment variable **PATH** (see *environ*(5)). The environment is supplied by the shell (see *sh*(1)). If *file* does not have an executable magic number (see *magic*(4)), it is passed to the POSIX shell as a shell script.

arg0, ..., *argn* are one or more 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 to *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 **execle()** and **execve()**, 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.

Multi-threaded applications should not use the *environ* variable to access or modify any environment variable while another thread is concurrently modifying any environment variable. Calling any function which is dependent upon any environment variable is considered a use of the environ variable to access that environment variable.

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 values (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-ID and set-group-ID functions do not apply to scripts; thus, if **execlp()** or **execvp()** executes a script, the set-user-ID and set-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-ID or set-group-ID is in effect.

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

Text and data segment memory locks are not passed on to the new program (see *plock*(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-execution
- file size limit (see *ulimit*(2))
- interval timers (see *getitimer*(2))
- nice value (see *nice*(2))
- nice value (see parent process ID)
- pending signals
- process ID
- process group ID
- real user ID
- real group ID
- process start time

- 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 the **PT_SETTRC** request of *ptrace*(2))
- **tms_utime**, **tms_stime**, **tms_cutime**, and **tms_cstime** (see *times*(2))

For a script file, the initial line of a script file must begin with **#!** as the first two bytes, followed by zero or more spaces, followed by *interpreter* or *interpreter argument*, as in:

#! *interpreter* [*argument*]

One or more spaces or tabs must separate *interpreter* and *argument*. The first line should end with either a newline or a null character.

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 *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]**. **argv[0]** is passed as specified in the **exec*()** call. 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]**).

Some interpreters process the *interpreter* and the *argument* internally, and do not provide the *interpreter* and the *argument* to the users script.

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.

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

For a executable object file, the arguments are passed as **argv[1],** ..., **argv[n]**. **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]**.

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

If **exec*()** fails and returns to the calling program, **errno** is set to one of the following values:

[E2BIG] The number of bytes in the new program's argument list plus environment 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 the 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.

[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 will be recognized.

[ELOOP] Too many symbolic links were 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 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.

[ENOEXEC] The function call is not **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 the file's 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.

e

WARNINGS

Unsharable executable files are not supported. These are files whose **EXEC_MAGIC** magic number was produced with the **-N** option of **ld** (see *ld*(1)).

DEPENDENCIES

HP Process Resource Manager

If the optional HP Process Resource Manager (PRM) software is installed and configured, the process's process resource group ID is not changed by **exec*()**. See *prmconfig*(1) for a description of how to configure HP PRM, and *prmconf*(4) for the definition of process resource group.

SEE ALSO

sh(1), sh-posix(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).

HP Process Resource Manager: prmconfig(1), prmconf(4) in *HP Process Resource Manager User's Guide*.

STANDARDS CONFORMANCE

environ: AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

execl(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

execle(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

execlp(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

execv(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

execve(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

execvp(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

exit, _exit - terminate process

SYNOPSIS

```
#include <stdlib.h>

void exit(int status);

#include <unistd.h>

void _exit(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 is undefined if *main* does not take care to return a value or to call **exit()** explicitly).

If the calling process is multithreaded, all threads/lightweight process in the process will be terminated.

exit() 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 standard I/O buffers, while **_exit()** does not. Both **exit()** and **_exit()** terminate the calling process with the following consequences. The exact order of these consequences is unspecified.

> Functions registered by **atexit()** (see *atexit*(2)) are called in reverse order of registration.
>
> All file descriptors open in the calling process are closed.
>
> All files created by **tmpfile()** are removed (see *tmpfile*(3S)).
>
> 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()** (see *times*(2)).
>
> The parent process ID is set to 1 for all of the calling process's existing child processes and zombie processes. This means the initialization process (proc1) inherits each of these processes.
>
> Threads/LWPs terminated by a call to **exit()** shall not invoke their cancellation cleanup handlers or their thread specific data destructor functions.
>
> 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 are 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), pthread_cancel(3), pthread_exit(3), pthread_key_create(3), semop(2), shmop(2), times(2), vfork(2), wait(2), signal(5).

STANDARDS CONFORMANCE

exit(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

_exit(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

fcntl - file control

SYNOPSIS

```
#include <fcntl.h>

int fcntl(int fildes, int cmd, ... /* arg */);
```

Remarks

The ANSI C "**, ...**" construct denotes a variable length argument list whose optional [or required] members are given in the associated comment (**/* */**).

DESCRIPTION

fcntl() provides for control over open files. *fildes* is an open file descriptor.

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

F_DUPFD Return a new file descriptor having the following characteristics:

- Lowest numbered available file descriptor greater than or equal to the third argument, *arg*, taken as an integer of type **int**.
- 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 the third argument, *arg*, taken as an integer of type **int** (see **F_GETFD**).

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

F_SETFL Set file status flags to the third argument, *arg*, taken as an integer of type **int**. 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 the third argument, *arg*, taken as a pointer to type **struct flock**. 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 the third argument, *arg*, taken as a pointer to type **struct flock** (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.

F_GETOWN If *fildes* refers to a socket, **fcntl()** returns the process or process group ID specified to receive **SIGURG** signals when out-of-band data is available. Positive values indicate a process ID; negative values, other than -1, indicate a process group ID.

F_SETOWN If *fildes* refers to a socket, **fcntl()** sets the process or process group ID specified to receive **SIGURG** signals when out-of-band data is available, using the value of the

f

third argument, arg, taken as type int. Positive values indicate a process ID; negative values, other than -1, indicate a process group ID.

F_GETLK64 Same as **F_GETLK**, except *arg* is a pointer to **struct flock64** instead of **struct flock**.

F_SETLK64 Same as **F_SETLK**, except *arg* is a pointer to **struct flock64** instead of **struct flock**.

F_SETLKW64 Same as **F_SETLKW**, except *arg* is a pointer to **struct flock64** instead of **struct flock**.

Turning the **O_LARGEFILE** flag on and off can be done with **F_SETFL**.

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.

f

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()** and **write()** system calls on the file are affected by the record locks in effect.

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
             */
```

NETWORKING FEATURES

NFS

The advisory record-locking capabilities of **fcntl()** 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.

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 than –1.

F_SETLK Value other than –1.

F_SETLKW Value other than –1.

F_GETOWN Value of process or process group ID specified to receive **SIGURG** signals when out-of-band data is available.

F_SETOWN Value other than –1.

F_GETLK64 Value other than –1.

F_SETLK64 Value other than –1.

F_SETLKW64 Value other than –1.

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

ERRORS

fcntl() fails if any of the following conditions occur:

[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).

[EINVAL] *cmd* is **F_DUPFD** and *arg* is greater than or equal to the maximum number of file descriptors.

[EINVAL] *cmd* is **F_DUPFD** and *arg* is negative.

[EINVAL] *cmd* is **F_GETLK**, **F_SETLK**, or **F_SETLKW**, and *arg* 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**) is a read lock (**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 an NFS file with access bits set for enforcement mode.

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

[EOVERFLOW]
cmd is **F_GETLK** and the blocking lock's starting offset or length would not fit in the caller's structure.

[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.

[EAGAIN] *cmd* is **F_SETLK** or **F_SETLKW**, and the file is mapped in to virtual memory via the **mmap()** system call (see *mmap*(2)).

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

[ENOTSOCK] *cmd* is **F_GETOWN** or **F_SETOWN**, and *fildes* does not refer to a socket.

AUTHOR

fcntl() was developed by HP, AT&T and the University of California, Berkeley.

SEE ALSO

lockd(1M), statd(1M), chmod(2), close(2), creat64(2), exec(2), lockf(2), open(2), read(2), write(2), fcntl(5).

STANDARDS CONFORMANCE

fcntl(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

fork - create a new process

SYNOPSIS

```
#include <unistd.h>

pid_t fork(void);
```

DESCRIPTION

The **fork()** system call causes the creation of a new process. The new child process is created wth exactly one thread or lightweight process. The new child process contains a replica of the calling thread (if the calling process is multi-threaded) and its entire address space, possibly including the state of mutexes and other resources.

If the calling process is multi-threaded, the child process may only execute async-signal safe functions until one of the exec functions is called. Fork handlers may be installed via **pthread_atfork()** in order to maintain application invariants across **fork()** calls (i.e, release resources such as mutexes in the child process).

The child process inherits the following attributes from the parent process:

- Real, effective, and saved user IDs.
- Real, effective, and saved group IDs.
- 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 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 the *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 for **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.

f

If a parent and child process both have a file opened and the parent or child closes the file, the file is still open for the other process.

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

f

If **fork()** fails, **errno** is set to one of the following values.

- [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.

DEPENDENCIES

HP Process Resource Manager

If the optional HP Process Resource Manager (PRM) software is installed and configured, the child process inherits the parent's process resource group ID. See *prmconfig*(1) for a description of how to configure HP PRM, and *prmconf*(4) for the definition of process resource group.

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), pthread_atfork(3T), ptrace(2), rtprio(2), semop(2), setpgrp(2), setuid(2), shmop(2), times(2), ulimit(2), umask(2), vfork(2), wait(2), fclose(3S), stdio(3S), acct(4), signal(5).

HP Process Resource Manager: prmconfig(1), prmconf(4) in *HP Process Resource Manager User's Guide*.

STANDARDS CONFORMANCE

fork(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

fsctl - file system control

SYNOPSIS

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

int fsctl(
     int fildes,
     int command,
     void *outbuf,
     size_t outlen
);
```

f

DESCRIPTION

fsctl() provides 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 *command*s 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 *command*s 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 that are common to all extended attribute records. (See EXAMPLES below 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 **CDMAXNAMLEN** defined in <**sys/cdfsdir.h**> gives the maximum length a file identifier can have. Thus, **CDMAXNAMLEN** + 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. **CDMAXNAMLEN** + 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. **CDMAXNAMLEN** + 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 fragment 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.

ERRORS

fsctl() fails if any of the following conditions are encountered:

[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] *fildes* does not refer to a CDFS file system.

SEE ALSO

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

NAME

fstat - get file status

SYNOPSIS

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

int fstat(int fildes, struct stat *buf);
```

DESCRIPTION

The **fstat()** function obtains information about an open file associated with the file descriptor *fildes*, and writes it to the area pointed to by *buf*. The *buf* argument is a pointer to a **stat** structure, as defined in **<sys/stat.h>**, into which information is placed concerning the file.

The structure members *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atime*, *st_ctime*, and *st_mtime* will have meaningful values for all file types defined in this document. The value of the member *st_nlink* will be set to the number of links to the file.

An implementation that provides additional or alternative file access control mechanisms may, under implementation-dependent conditions, cause **fstat()** to fail.

The **fstat()** function updates any time-related fields as described in File Times Update (see the XBD specification, Chapter 4, Character Set), before writing into the **stat** structure.

RETURN VALUE

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

ERRORS

The **fstat()** function will fail if:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EIO]	An I/O error occurred while reading from the file system.

The **fstat()** function may fail if:

[EOVERFLOW]	One of the values is too large to store into the structure pointed to by the *buf* argument.

SEE ALSO

lstat(2), stat(2), <sys/stat.h>, <sys/types.h>.

CHANGE HISTORY

First released in Issue 1.

Derived from Issue 1 of the SVID.

Issue 4

The following changes are incorporated in the DESCRIPTION section for alignment with the ISO POSIX-1 standard:

- A paragraph defining the contents of **stat** structure members is added.
- The words "extended security controls" are replaced by "additional or alternative file access control mechanisms."

Another change is incorporated as follows:

- The header **<sys/types.h>** is now marked as optional (OH); this header need not be included on XSI-conformant systems.

Issue 4, Version 2

The ERRORS section is updated for X/OPEN UNIX conformance as follows:

- The EIO error is added as a mandatory error indicated the occurrence of an I/O error.
- The EOVERFLOW error is added as an optional error indicating that one of the values is too large to store in the area pointed to by *buf*.

HP-UX EXTENSIONS

DESCRIPTION

If the chosen path name or file descriptor refers to a Multi-Level Directory (MLD), and the process does not have the multilevel effective privilege, the i-node number returned in *st_ino* is the i-node of the MLD itself.

The parameters for the **fstat()** function is as follows:

buf is a pointer to a **stat()** structure, which is where the file status information is stored.

fildes is a file descriptor for an open file, which is created with the successful completion of an **open()**, **creat()**, **dup()**, **fcntl()**, or **pipe()** system call (see *open*(2), *creat*(2), *dup*(2), *fcntl*(2), or *pipe*(2)).

f

The **stat** structure contains the following members:

```
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 sysfs(2) */
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 */
long     st_blksize;   /* File system block size */
uint     st_acl:1;     /* Set if the file has optional */
                       /* access control list entries */
                       /* HFS File Systems only */
```

(Note that the position of items in this list does not necessarily reflect the order of the members in the structure.)

The fields contain the following information:

st_atime Time when file data was last accessed. Changed by the following system calls: **creat()**, **mknod()**, **pipe()**, **read()**, **readv()** (see *read*(2)), and **utime()**. If a file is mapped into virtual memory, accesses of file data through the mapping may also modify *st_mtime*. See *mmap*(2).

st_mtime Time when data was last modified. Changed by the following system calls: **creat()**, **truncate()**, **ftruncate()**, (see *truncate*(2)), **mknod()**, **pipe()**, **prealloc()**, **utime()**, **write()**, and **writev()** (see *write*(2)). Also changed by **close()** when the reference count reaches zero on a named pipe (FIFO special) file that contains data. If a file is mapped into virtual memory, updates of file data through the mapping may also modify *st_mtime*. See *mmap*(2).

st_ctime Time when file status was last changed. Changed by the following system calls: **chmod()**, **chown()**, **creat()**, **fchmod()**, **fchown()**, **truncate()**, **ftruncate()**, (see *truncate*(2)), **link()**, **mknod()**, **pipe()**, **prealloc()**, **rename()**, **setacl()**, **unlink()**, **utime()**, **write()**, and **writev()** (see *write*(2)). The **touch** command (see *touch*(1) can be used to explicitly control the times of a file.

st_mode The value returned in this field is the bit-wise 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 users with appropriate privileges 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()** clears this bit under these conditions (see *getaccess*(2).

f

ERRORS

[EFAULT] *buf* or *path* points to an invalid address. The reliable detection of this error is implementation-dependent.

[EOVERFLOW] The file size in bytes or the number of blocks allocated to the file cannot be represented correctly in the structure pointed to by *buf*.

NFS

The *st_basemode* and *st_acl* fields are zero on files accessed remotely. *st_acl* field is applicable to HFS File Systems only.

WARNINGS

Access Control Lists - HFS File Systems only

Access control list descriptions in this entry apply only to HFS file systems on standard HP-UX operating systems.

DEPENDENCIES

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.

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), fstat64(2), link(2), mknod(2), pipe(2), read(2), rename(2), setacl(2), sysfs(2), time(2), truncate(2), unlink(2), utime(2), write(2), acl(5), stat(5).

STANDARDS CONFORMANCE

fstat(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

f

NAME

fsync, fdatasync - synchronize a file's in-core and on-disk states

SYNOPSIS

```
#include <unistd.h>

int fsync(int fildes);

int fdatasync(int fildes);
```

DESCRIPTION

fsync() and **fdatasync()** cause 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()** and **fdatasync()** apply to ordinary files, and apply to block special devices on systems which permit I/O to block special devices.

fsync() and **fdatasync()** should be used by programs that require a file to be in a known state, such as when building a simple transaction facility.

fdatasync() causes all modified data and file attributes of *fildes* required to retrieve the data to be written to disk.

fsync() causes all modified data and all file attributes of *fildes* (including access time, modification time and status change time) to be written to disk.

Together, **fsync()** and **fdatasync()** constitute support for File Synchronization.

RETURN VALUE

fsync() and **fdatasync()** return 0 on success or −1 if an error occurs, and set **errno** to indicate the error.

ERRORS

fsync and *fdatasync* fail if any of the following conditions are encountered:

[EBADF]	*fildes* is not a valid descriptor.
[EINVAL]	*fildes* refers to a file type to which **fsync()** or **fdatasync()** does not apply.

WARNINGS

The current implementation of these functions is inefficient for large files.

AUTHOR

fsync() was developed by the the University of California, Berkeley and HP.

SEE ALSO

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

STANDARDS CONFORMANCE

fsync(): AES, SVID3, XPG3, XPG4, POSIX.4

fdatasync(): POSIX.4

NAME

ftime - get date and time more precisely

SYNOPSIS

#include <sys/timeb.h>

int ftime(struct timeb *tp);

Remarks

This facility is provided for backwards compatibility with Version 7 systems. Either **time()** or **gettimeofday()** should be used in new programs.

DESCRIPTION

ftime() fills in a structure pointed to by its argument, as defined by <**sys/timeb.h**>:

```
/*
 * Structure returned by ftime system call
 */
struct timeb {
    time_t         time;
    unsigned short millitm;
    short          timezone;
    short          dstflag;
};
```

The structure contains the time in seconds since 00:00:00 UTC (Coordinated Universal Time), January 1, 1970, up to 1000 milliseconds of more-precise interval, the local timezone (measured in minutes of time westward from UTC), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year. Consult *gettimeofday*(2) for more details on the meaning of the timezone field.

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

WARNINGS

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

SEE ALSO

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

f

NAME

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

SYNOPSIS

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

int getaccess(
     const char *path,
     uid_t uid,
     int ngroups,
     const gid_t *gidset,
     void *label,
     void *privs
);
```


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 logical 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, **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 (HFS File Systems only)	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

g

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 **errno** is set to indicate the error.

ERRORS

getaccess() fails if any of the following conditions are encountered:

[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 is 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).

g

NAME

getacl, fgetacl - get access control list (ACL) information (HFS File Systems only)

SYNOPSIS

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

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

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

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*.

nentries 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 **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

getacl() and **fgetacl()** are only supported on HFS file system on standard HP-UX operating system.

AUTHOR

getacl() and **fgetacl()** were developed by HP.

SEE ALSO

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

g

NAME

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

SYNOPSIS

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

int getaudid(void);
```

DESCRIPTION

getaudid() returns the audit ID (*aid*) for the current process. This call is restricted to the super-user.

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 super-user.

AUTHOR

getaudid() was developed by HP.

SEE ALSO

setaudid(2).

g

NAME

getaudproc - get the audit process flag for the calling process

SYNOPSIS

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

int getaudproc(void);
```

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 super-user.

RETURN VALUE

Upon successful completion, the audit process flag is returned; otherwise, a **-1** is returned and **errno** is set to indicate the error.

ERRORS

getaudproc() fails if the following is true:

[EPERM] The caller is not the super-user.

AUTHOR

getaudproc() was developed by HP.

SEE ALSO

setaudproc(2).

NAME

getcontext, setcontext - get and set current user context

SYNOPSIS

```
#include <ucontext.h>

int getcontext(ucontext_t *ucp);

int setcontext(const ucontext_t *ucp);
```

DESCRIPTION

The **getcontext()** function initializes the structure pointed to by *ucp* to the current user context of the calling process. The *ucontext_t* type that *ucp* points to defines the user context and includes the contents of the calling process' machine registers, the signal mask, and the current execution stack.

The **setcontext()** function restores the user context pointed to by *ucp*. A successful call to **setcontext()** does not return; program execution resumes at the point specified by the *ucp* argument passed to **setcontext()**. The *ucp* argument should be created either by a prior call to **getcontext()**, or by being passed as an argument to a signal handler. If the *ucp* argument was created with **getcontext()**, program execution continues as if the corresponding call of **getcontext()** had just returned. If the *ucp* argument was created with **makecontext()**, program execution continues with the function passed to **makecontext()**. When that function returns, the process continues as if after a call to **setcontext()** with the *ucp* argument that was input to **makecontext()**. If the *ucp* argument was passed to a signal handler, program execution continues with the program instruction following the instruction interrupted by the signal. If the *uc_link* member of the **ucontext_t** structure pointed to by the *ucp* argument is equal to 0, then this context is the main context, and the process will exit when this context returns. The effects of passing a *ucp* argument obtained from any other source are unspecified.

RETURN VALUE

On successful completion, **setcontext()** does not return and **getcontext()** returns 0. Otherwise, a value of –1 is returned.

WARNINGS

Context APIs are not recommended due to possible compatibility problems from release to release, because context APIs are very architecture-specific. The context APIs "expose" the architecture to the application, such that the application may not be compatible with all releases.

If you must use context APIs, be aware of the following:

- Do not copy the context yourself. It is not contiguous. The context may have pointers that may point back to the original context rather than in the copied context; hence, it will be broken.
- The size of the context will vary in length from release to release.

ERRORS

No errors are defined.

APPLICATION USAGE

When a signal handler is executed, the current user context is saved and a new context is created. If the process leaves the signal handler via **longjmp()**, then it is unspecified whether the context at the time of the corresponding **setjmp()** call is restored and thus whether future calls to **getcontext()** will provide an accurate representation of the current context, since the context restored by **longjmp()** may not contain all the information that **setcontext()** requires. Signal handlers should use **siglongjmp()** or **setcontext()** instead.

Portable applications should not modify or access the *uc_mcontext* member of *ucontext_t*. A portable application cannot assume that context includes any process-wide static data, possibly including **errno**. Users manipulating contexts should take care to handle these explicitly when required.

SEE ALSO

bsd_signal(), makecontext(2), setjmp(3C), sigaction(2), sigaltstack(2), sigprocmask(2), sigsetjmp(), <ucontext.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

g

g

NAME

getdirentries() - get entries from a directory in a file-system-independent format

SYNOPSIS

```
#include <ndir.h>

int getdirentries(
    int            fildes,
    struct direct *buf,
    size_t         nbytes,
    off_t         *basep
);
```

DESCRIPTION

The **getdirentries()** system call and the <**ndir.h**> header file have been obsoleted starting from HP-UX 10.30 by the functions described in *directory*(3C). **getdirentries()** will not be supported for 64-bit applications.

The **getdirentries()** system call places directory entries from the directory referenced by the file descriptor *fildes* into the buffer pointed to by *buf*, in a file-system-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 **st_blksize** in *stat*(2). (Smaller block sizes can cause errors on certain file systems.) *nbytes* must be less than or equal to 65536 (64K).

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()**; see *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()**, 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

getdirentries() returns the following values:

- *n* Successful completion. *n* is the number of bytes actually transferred.
- **-1** Failure. **errno** is set to indicate the error.

ERRORS

If **getdirentries()** fails, **errno** is set to one of the following values:

- [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.
- [EINVAL] *nbytes* is greater than the size of the **direct** structure pointed to by *buf*.
- [EINVAL] *nbytes* is greater than 65536 or is smaller than the size of a single directory entry.

[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

lseek(2), open(2), directory(3C).

g

g

NAME

getdomainname, setdomainname - get/set name of current Network Information Service domain

SYNOPSIS

int getdomainname(char *name, int namelen);

int setdomainname(char *name, int namelen);

DESCRIPTION

getdomainname() returns the name of the Network Information Service (NIS) domain for the current processor, as previously set by **setdomainname()**. The parameter *namelen* specifies the size of the *name* array. The returned value is null-terminated unless the area pointed to by *name* is not large enough to hold the domain name plus the null byte. In this case, only the *namelen* number of bytes is returned.

setdomainname() sets the domain of the host machine to *name*, which has a length of *namelen*. This call is restricted to the superuser and is normally used only when the system is booted.

These Network Information Service domains enable two distinct networks with common host names to merge. Each network is distinguished by having a different domain name. Currently, only the Network Information Service uses these domains.

RETURN VALUE

If the call succeeds, a value of 0 is returned. If the call fails, a value of –1 is returned and **errno** is set to indicate the error.

ERRORS

If **getdomainname()** or **setdomainname()** fail, **errno** is set to one of the following values:

[EFAULT] *name* points outside the accessible address space.

[EPERM] The caller is not superuser. This error only applies to **setdomainname()**.

WARNINGS

The length of the *name* array should be at least 65; NIS domain names can be up to 64 characters long.

NIS servers use the NIS domain name as the name of a subdirectory of **/var/yp**. Since the NIS domain name can be as long as 64 characters, the domain name set with **setdomainname()** can exceed the maximum file name length allowed on the local file system. If that length is exceeded, the name of the subdirectory is the truncated NIS domain name.

AUTHOR

getdomainname was developed by Sun Microsystems, Inc.

SEE ALSO

domainname(1), ypserv(1M), ypfiles(4).

NAME

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

SYNOPSIS

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

int getevent(
     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 super-user.

RETURN VALUE

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

ERRORS

getevent() fails if the following is true:

[EPERM] The caller is not super-user.

AUTHOR

getevent() was developed by HP.

SEE ALSO

setevent(2), audevent(1M).

NAME

getfh() - return file handle for file on remote node

SYNOPSIS

```
#include <errno.h>
#include <time.h>
#include <nfs/nfs.h>
#include <rpc/rpc.h>

int getfh(char *path, fhandle_t *fhp);
```

DESCRIPTION

The **getfh()** system call returns a file handle in the **struct** pointed to by *fhp* for the file pointed to by *path*. This information is used to perform an NFS mount for a remote node. **getfh()** is executed on the remote node; results are passed back to the program doing the NFS mount. The caller should never examine the file handle contents. The file handle only identifies a file to the node that produced the file handle. (The term "file handle" refers to an NFS concept.)

The effective user ID of the calling process must be superuser.

RETURN VALUE

getfh() returns the following values:

- **0** Successful completion.
- **-1** Failure. **errno** is set to indicate the error.

ERRORS

If **getfh()** fails, **errno** is set to one of the following values.

[EINVAL]	Invalid argument, or the file or directory has not been exported by **exportfs** (see *exportfs*(1M)).
[ENOENT]	File or directory specified by *path* does not exist.
[EPERM]	The effective user ID is not superuser.
[EREMOTE]	The file or directory specified by *path* is a remote file or directory.

WARNINGS

This call should be used only by HP-supplied commands and is not recommended for use by non-HP-supplied programs.

AUTHOR

getfh() was developed by Sun Microsystems, Inc.

SEE ALSO

exportfs(1M), mount(1M), vfsmount(2).

NAME

getgroups - get group access list

SYNOPSIS

```
#include <unistd.h>

int getgroups(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_MAX**, as defined in **<limits.h>**, is ever 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.

g

EXAMPLES

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

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

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

RETURN VALUE

If successful, **getgroups()** returns a non-negative value indicating the number of elements returned in *gidset*. If an error occurs, a value of −1 is returned and **errno** is set to indicate the type of error.

ERRORS

getgroups() fails if any of the following conditions are encountered:

[EFAULT] *gidset* specifies an invalid address. The reliable detection of this error is 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(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

gethostid - get an identifier for the current host

SYNOPSIS

```
#include <unistd.h>

long gethostid(void);
```

DESCRIPTION

The **gethostid()** function retrieves a 32-bit identifier for the current host.

RETURN VALUE

Upon successful completion, **gethostid()** returns an identifier for the current host.

ERRORS

No errors are defined.

g

APPLICATION USAGE

X/Open does not define the domain in which the return value is unique.

SEE ALSO

random(3M), <unistd.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

NAME

gethostname - get name of current host

SYNOPSIS

```
#include <unistd.h>

int gethostname(char *hostname, size_t size);
```

DESCRIPTION

gethostname() returns in the array to which *hostname* points, the standard host name for the current processor as set by **sethostname()** (see *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, it returns –1 and sets **errno** 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.

AUTHOR

gethostname() was developed by the University of California, Berkeley.

SEE ALSO

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

g

g

NAME

getitimer, setitimer - get/set value of interval timer

SYNOPSIS

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

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

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

DESCRIPTION

The **getitimer()** function stores the current value of the timer specified by *which* into the structure pointed to by *value*. The **setitimer()** function sets the timer specified by *which* to the value specified in the structure pointed to by *value*, and if *ovalue* is not a null pointer, stores the previous value of the timer in the structure pointed to by *ovalue*.

A timer value is defined by the **itimerval** structure. 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, regardless of the value of *it_interval*. Setting *it_interval* to 0 disables a timer after its next expiration (assuming *it_value* is non-zero).

Implementations may place limitations on the granularity of timer values. For each interval timer, if the requested timer value requires a finer granularity than the implementation supports, the actual timer *value* will be rounded up to the next supported value.

Implementations may place limitations on the timer value. To make sure that a process gets at least as much time as requested, the timer value is rounded up to the next timer tick (a timer tick is the smallest supported value). The timer value is rounded up to the next timer tick because, the timer will be initialize somewhere between timer ticks. If a **setitimer()** is followed by a **getitimer()** without a timer tick in between, it is possible that the *value* returned by **getitimer()** may be more than the initial *value* requested by **setitimer()** due to this rounding.

An XSI-conforming implementation provides each process with at least three interval timers, which are indicated by the *which* argument:

ITIMER_REAL — Decrements in real time. A **SIGALRM** signal is delivered when this timer expires.

ITIMER_VIRTUAL — Decrements in process virtual time. It runs only when the process is executing. A **SIGVTALRM** signal is delivered when it expires.

ITIMER_PROF — 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.

The interaction between **setitimer()** and any of **alarm()**, **sleep()** or **usleep()** is unspecified.

RETURN VALUE

Upon successful completion, **getitimer()** or **setitimer()** returns 0. Otherwise, −1 is returned and **errno** is set to indicate the error.

ERRORS

The **setitimer()** function will fail if:

[EINVAL] The value argument is not in canonical form.(In canonical form, the number of microseconds is a non-negative integer less than 1,000,000 and the number of seconds is a non-negative integer.)

The **getitimer()** and **setitimer()** functions may fail if:

[EINVAL] The which argument is not recognized.

SEE ALSO

alarm(2), sleep(3C), ualarm(2), usleep(2), <signal.h>, <sys/time.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

HP-UX EXTENSIONS

DESCRIPTION

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

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

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).

g

Each time the **ITIMER_PROF** timer expires, the **SIGPROF** signal is delivered. Since this signal can 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 <**sys/time.h**>:

timerclear Set a time value to zero.

timerisset Test if a time value is non-zero.

timercmp Compare two time values. (Beware that **>=** and **<=** do not work with the **timercmp** macro.)

The timer used with **ITIMER_REAL** is also used by **alarm()** (see *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.

ERRORS

getitimer() or **setitimer()** fail if any of the following conditions are encountered:

[EFAULT] The *value* structure specified a bad address. Reliable detection of this error is 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()** 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

getksym - get information for a global kernel symbol

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ksym.h>
#include <sys/libelf.h>
int getksym(char *symname, char *modname, uint64_t *value,
            uint64_t *info);
```

Remarks

getksym is currently implemented as a macro.

DESCRIPTION

There are two ways that **getksym** can be used to retrieve kernel symbol information. As detailed below, if *symname* is provided, **getksym** attempts to retrieve the value of the symbol; if *value* is provided, the associated symbol name is retrieved.

getksym, given a *symname*, looks for a global (**STB_GLOBAL** or **STB_WEAK**) symbol of that name in the symbol table of the static kernel and all currently loaded kernel modules. If it finds a match, **getksym** returns the value associated with that symbol (typically its address) in the space pointed to by *value*, and the type of that symbol in the space pointed to by *info*.

g

The types returned are:

STT_NOTYPE	unknown type
STT_FUNC	text symbol (typically function)
STT_PARISC_MILLI	millicode function
STT_OBJECT	data symbol

The symbol name can be no more than *MAXSYMNMLEN* characters. If *modname* is set to the name (basename only) of a dynamically loaded module, then the search for the symbol name will only be in that module. If *modname* is **NULL**, then the search order for the symbol name will be the static kernel followed by each of the currently loaded modules in the order in which they were loaded. The module name can be no more that *MODMAXNAMELEN* characters.

If **getksym** is given a valid address in the statically configured kernel or one of the currently loaded modules in the space pointed to by *value*, it will return, in the space pointed to by *symname*, the name of the symbol whose value is the closest one less than or equal to the given value and, in space pointed to by *info*, the difference between the address given and the value of the symbol found. The space pointed to by *symname* must be at least *MAXSYMNMLEN* characters long.

RETURN VALUE

getksym returns **0** upon successful completion. If an error occurs, a value of **-1** is returned and **errno** is set to indicated the error.

ERRORS

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

[ENOMATCH]	The symbol name given is not found, or the value given is not a currently valid address.
[EINVAL]	*modname* does not represent a currently loaded module.
[ENAMETOOLONG]	*modname* is greater than *MODMAXNAMELEN* characters long, or *symname* is greater that *MAXSYMNMLEN* characters long.

SEE ALSO

kmem(7).

g

NAME

getmsg, getpmsg - receive next message from a STREAMS file

SYNOPSIS

```
#include <stropts.h>

int getmsg(
     int fildes,
     struct strbuf *ctlptr,
     struct strbuf *dataptr,
     int *flagsp
);

int getpmsg(
     int fildes,
     struct strbuf *ctlptr,
     struct strbuf *dataptr,
     int *band,
     int *flagsp
);
```

DESCRIPTION

The **getmsg()** function retrieves the contents of a message located at the head of the stream head read queue associated with a STREAMS file and places the contents into one or more buffers. The message contains either a data part, a control part, or both. The data and control parts of the message are placed into separate buffers, as described below. The semantics of each part is defined by the originator of the message.

The **getpmsg()** function does the same thing as **getmsg()**, but provides finer control over the priority of the messages received. Except where noted, all requirements on **getmsg()** also pertain to **getpmsg()**.

The *fildes* argument specifies a file descriptor referencing a STREAMS-based file.

The *ctlptr* and *dataptr* arguments each point to a **strbuf** structure, in which the *buf* member points to a buffer in which the data or control information is to be placed, and the *maxlen* member indicates the maximum number of bytes this buffer can hold. On return, the *len* member contains the number of bytes of data or control information actually received. The *len* member is set to 0 if there is a zero-length control or data part and *len* is set to -1 if no data or control information is present in the message.

When **getmsg()** is called, *flagsp* should point to an integer that indicates the type of message the process is able to receive. This is described further below.

The *ctlptr* argument is used to hold the control part of the message, and *dataptr* is used to hold the data part of the message. If *ctlptr* (or *dataptr*) is a null pointer or the *maxlen* members is −1, the control (or data) part of the message is not processed and is left on the stream head read queue, and if the *ctlptr* (or *dataptr*) is not a null pointer, *len* is set to −1. If the *maxlen* member is set to 0 and there is a zero-length control (or data) part, that zero-length part is removed from the read queue and *len* is set to 0. If the *maxlen* member is set to 0 and there are more than 0 bytes of control (or data) information, that information is left on the read queue and *len* is set to 0. If the *maxlen* member in *ctlptr* (or *dataptr*) is less than the control (or data) part of the message, *maxlen* bytes are retrieved. In this case, the remainder of the message is left on the stream head read queue and a on-zero return value is provided.

By default, **getmsg()** processes the first available message on the stream head read queue. However, a process may choose to retrieve only high-priority messages by setting the integer pointed to by *flagsp* to **RS_HIPRI**. In this case, **getmsg()** will only process the next message if it is a high-priority message. When the integer pointed to by *flagsp* is 0, any message will be retrieved. In this case, on return, the integer pointed to by *flagsp* will be set to **RS_HIPRI** if a high-priority message was retrieved, or 0 otherwise.

For **getpmsg()**, the flags are different. The *flagsp* argument points to a bitmask with the following mutually-exclusive flags defined. **MSG_HIPRI**, **MSG_BAND**, and **MSG_ANY**. Like **getmsg()**, **getpmsg()** processes the first available message on the stream head read queue. A process may choose to retrieve only high-priority message by setting the integer pointed to by *flagsp* to **MSG_HIPRI** and the integer pointed to by *bandp* to 0. In this case, **getpmsg()** will only process the next message if is a high-priority message. In a similar manner, a process may choose to retrieve a message from a particular priority band by setting the integer pointed to by *flagsp* to **MSG_BAND** and the integer pointed to by *bandp* to the priority band of interest. In this case, **getpmsg()** will only process the next message if it is in a priority band equal to, or greater than, the integer pointed to by *bandp*, or if it is a high-priority message. If a process just wants to get the first message off the queue, the integer pointed to by *bandp* should be set to 0. On return, if the message retrieved was a high-priority message, the integer pointed to by *flagsp* will be set to **MSG_HIPRI** and the integer pointed to by *bandp* will be set to 0. Otherwise, the integer pointed to by *flagsp* will be set to **MSG_BAND** and the integer pointed to by *bandp* will be set to the priority band of the message.

If **O_NONBLOCK** is not set, **getmsg()** and **getpmsg()** will not block until a message of the type specified by *flagsp* is available at the front of the stream head read queue. If **O_NONBLOCK** is set and a message of the specified type is not present at the front of the read queue, **getmsg()** and **getpmsg()** fail and set **errno** to [EAGAIN].

If a hangup occurs on the stream from which messages are to be retrieved, **getmsg()** and **getpmsg()** continue to operate normally, as described above, until the stream head read queue is empty. Thereafter, they return 0 in the *len* members of *ctlptr* and *dataptr*.

MULTITHREAD USAGE

The **getmsg()** and **getpmsg()** functions are safe to be called by multithreaded applications, and they are thread-safe for both POSIX Threads and DCE User Threads. The **getmsg()** and **getpmsg()** functions have cancellation points. They are async-signal safe and fork-safe. They are not async-cancel safe.

RETURN VALUE

Upon successful completion, **getmsg()** and **getpmsg()** return a non-negative value. A value of 0 indicates that a full message was read successfully. A return value of **MORECTL** indicates that more control information is waiting for retrieval. A return value of **MOREDATA** indicates that more data is waiting for retrieval. A return value of the bitwise logical OR of **MORECTL** and **MOREDATA** indicates that both types of information remain. Subsequent **getmsg()** and **getpmsg()** calls retrieve the remainder of the message. However, if a message of higher priority has come in on the stream head read queue, the next call to **getmsg()** or **getpmsg()** retrieves that higher-priority message before retrieving the remainder of the previously-received partial message.

Upon failure, **getmsg()** and **getpmsg()** return -1 and set **errno** to indicate the error.

ERRORS

The **getmsg()** and **getpmsg()** functions will fail if:

[EAGAIN] The **O_NONBLOCK** flag is set and no messages are available.

[EBADF] The *fildes* argument is not a valid file descriptor open for reading.

[EBADMSG] The queued message to be read is not valid for **getmsg()** or **getpmsg()** or a pending file descriptor is at the stream head.

[EINTR] A signal was caught during **getmsg()** or **getpmsg()**.

[EINVAL] An illegal value was specified by *flagsp*, or the stream or multiplexor referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexor.

[ENOSTR] A stream is not associated with *fildes*.

SEE ALSO

poll(2), putmsg(2), read(2), write(2), <stropts.h>, streamio(7).

g

NAME

getpagesize - get the current page size

SYNOPSIS

```
#include <unistd.h>

int getpagesize(void);
```

DESCRIPTION

The **getpagesize()** function returns the current page size.

The **getpagesize()** function is equivalent to **sysconf(_SC_PAGE_SIZE)** and **sysconf(_SC_PAGESIZE)**.

RETURN VALUE

The **getpagesize()** function returns the current page size.

ERRORS

No errors are defined.

APPLICATION USAGE

The value returned by **getpagesize()** need not be the minimum value that **malloc()** can allocate. Moreover, the application cannot assume that an object of this size can be allocated with **malloc()**.

SEE ALSO

brk(2), getrlimit(2), mmap(2), mprotect(2), munmap(2), msync(2), sysconf(2), <unistd.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

NAME

getpeername - get address of connected peer

SYNOPSIS

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

AF_CCITT only:

```
#include <x25/x25addrstr.h>

int getpeername(int s, void *addr, int *addrlen);
```

_XOPEN_SOURCE_EXTENDED only (UNIX 98)

```
int getpeername(int s, struct sockaddr *addr, socklen_t *addrlen);
```

Obsolescent _XOPEN_SOURCE_EXTENDED only (UNIX 95)

```
int getpeername(int s, struct sockaddr *addr, size_t *addrlen);
```

DESCRIPTION

getpeername() returns the address of the peer socket connected to the socket indicated by *s*, where *s* is a socket descriptor. *addr* points to a socket address structure in which this address is returned. *addrlen* points to a variable that should be initialized to indicate the size of the address structure. On return, the variable contains the actual size of the address returned (in bytes). If *addr* does not point to enough space to contain the whole address of the peer, only the first *addrlen* bytes of the address are returned.

AF_CCITT only:

The *addr* struct contains the X.25 addressing information of the *remote* peer socket connected to socket *s*. However, the **x25ifname[]** field of the *addr* struct contains the name of the *local* X.25 interface through which the call arrived.

g

RETURN VALUE

Upon successful completion, **getpeername()** returns 0; otherwise it returns −1 and sets **errno** to indicate the error.

ERRORS

getpeername() fails if any of the following conditions are encountered:

[EBADF]	*s* is not a valid file descriptor.
[ENOTSOCK]	*s* is a valid file descriptor, but it is not a socket.
[ENOTCONN]	The socket is not connected.
[ENOBUFS]	No buffer space is available to perform the operation.
[EFAULT]	*addr* or *addrlen* are not valid pointers.
[EINVAL]	The socket has been shut down.
[EOPNOTSUPP]	Operation not supported for AF_UNIX sockets.

OBSOLESCENCE

Currently, the **socklen_t** and **size_t** types are the same size. This is compatible with both the UNIX 95 and UNIX 98 profiles. However, in a future release, **socklen_t** might be a different size. In that case, passing a **size_t** pointer will evoke compile-time warnings, which must be corrected in order for the application to behave correctly. Applications that use **socklen_t** now, where appropriate, will avoid such migration problems. On the other hand, applications that need to be portable to the UNIX 95 profile should follow the X/Open specification (see *xopen_networking*(7)).

FUTURE DIRECTION

Currently, the default behavior is the **HP-UX BSD Sockets**; however, it might be changed to **X/Open Sockets** in a future release. At that time, any **HP-UX BSD Sockets** behavior that is incompatible with **X/Open Sockets** might be obsoleted. Applications that conform to the X/Open specification now will avoid migration problems (see *xopen_networking*(7)).

MULTITHREAD USAGE

The **getpeername()** system call is thread-safe. It has a cancellation point; and it is async-cancel safe, async-signal safe, and fork-safe.

AUTHOR

getpeername() was developed by HP and the University of California, Berkeley.

SEE ALSO

bind(2), socket(2), getsockname(2), inet(7F), af_ccitt(7F), xopen_networking(7).

NAME

getpid(), getpgid(), getpgrp(), getpgrp2(), getppid() - get process, process group and parent process ID.

SYNOPSIS

```
#include <unistd.h>

pid_t getpgid (pid_t pid);

pid_t getpgrp(void);

pid_t getpgrp2(pid_t pid);

pid_t getpid(void);

pid_t getppid(void);
```

DESCRIPTION

These functions return process, process group and parent process IDs, as follows:

getpgid() Process group ID of the specified process. If *pid* is zero, the call applies to the calling process. Same result as **getpgrp2()**.

getpgrp() Process group ID of the calling process.

getpgrp2() Process group ID of the specified process. If *pid* is zero, the call applies to the calling process. Same result as **getpgid()**.

getpid() Process ID of the calling process.

getppid() Parent process ID of the calling process.

RETURN VALUE

The functions return the following values:

n Successful completion. *n* is a nonnegative process ID, as described above.
-1 Failure: **getpgid()** and **getgrp2()** only. **errno** is set to indicate the error.

ERRORS

If **getpgid()** or **getpgrp2()** fails, **errno** is set to one of the following values:

[EPERM] The current process and *pid* are not in the same session (see *setsid*(2)).

[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), setpgid(2), setsid(2), signal(5).

STANDARDS CONFORMANCE

getpid(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

getpgrp(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

getppid(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

getpriority, setpriority - get or set process priority

SYNOPSIS

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

int getpriority(int which, int who);

int setpriority(int which, int who, int priority);
```

DESCRIPTION

getpriority() returns the priority of the indicated processes.

setpriority() sets the priority of the indicated processes to *priority*.

The processes are indicated by *which* and *who*, where *which* can have one of the following values:

- **PRIO_PROCESS** Get or set the priority of the specified process where *who* is the process ID. A *who* of **0** implies the process ID of the calling process.
- **PRIO_PGRP** Get or set the priority of the specified process group where *who* is the process-group ID, indicating all processes belonging to that process-group. A *who* of **0** implies the process-group ID of the calling process.
- **PRIO_USER** Get or set the priority of the specified user where *who* is the user ID, indicating all processes owned by that user. A *who* of **0** implies the user ID of the calling process.

If more than one process is indicated, the value returned by **getpriority()** is the lowest valued priority of all the indicated processes, and **setpriority()** sets the priority of all indicated processes.

priority is a value from **-20** to **20**, where lower values indicate better priorities. The default priority for a process is 0.

If the calling process contains more than one thread or lightweight process (i.e., the process is multi-threaded) these functions shall apply to all threads or lightweight processes in the calling process. The priority specified (or retrieved) is the same for all threads or lightweight processes in a process. Negative priorities require appropriate privileges.

RETURN VALUE

getpriority() returns the following values:

- *n* Successful completion. *n* is an integer priority in the range **-20** to **20**.
- **-1** Failure. **errno** is set to indicate the error. See WARNINGS below.

setpriority() returns the following values:

- **0** Successful completion.
- **-1** Failure. **errno** is set to indicate the error.

ERRORS

If **getpriority()** or **setpriority()** fails, **errno** is set to one of the following values:

- [EACCES] The calling process does not have access rights to change one or more of the indicated processes. All processes for which access is allowed are still affected.
- [EINVAL] *which* is not one of the choices listed above, or *who* is out of range.
- [EPERM] The calling process attempted to change the priority of a process to a smaller priority value without having appropriate privileges.
- [ESRCH] Processes indicated by *which* and *who* cannot be found.

WARNINGS

getpriority() can return **-1** both when it successfully finds a priority of **-1** and when it fails. To determine whether a failure occurred, set **errno** to **0** before calling **getpriority()**, then examine **errno** after the call returns.

AUTHOR

getpriority() and **setpriority()** were developed by the University of California, Berkeley.

SEE ALSO

nice(1), renice(1M), nice(2).

g

g

NAME

getprivgrp(), setprivgrp() - get and set special attributes for group

SYNOPSIS

#include <sys/privgrp.h>

int getprivgrp(struct privgrp_map *grplist);

int setprivgrp(gid_t grpid, const int *mask);

DESCRIPTION

getprivgrp()

The **getprivgrp()** system call 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 group ID with a privilege mask. Privilege masks are formed by ORing 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 value 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 the user's real or effective group ID, is returned to an ordinary user. The complete set is returned to a privileged user.

setprivgrp()

The **setprivgrp()** system call associates a kernel capability with a group ID. This allows subletting of superuser-like privileges to members of a particular group or groups. **setprivgrp()** takes two arguments: *grpid*, the integer group ID, and *mask*, 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 bits **1<<(5-1)** or 16. More generally, privilege *p* is represented by:

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

where **BITS_PER_INT** is **8*sizeof(mask[0])** given 8 bits per byte. 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 can 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 that have 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 that can be assigned privileges. One of these is always the psuedo-group **PRIV_GLOBAL**, allowing for **PRIV_MAXGRPS - 1** actual groups.

Only processes with appropriate privileges can use **setprivgrp()**.

RETURN VALUE

getprivgrp() and **setprivgrp()** return the following values:

- **0** Successful completion.
- **-1** Failure. **errno** is set to indicate the error.

ERRORS

If **getprivgrp()** fails, **errno** is set to one of the following values.

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

If **setprivgrp()** fails, **errno** is set to one of the following values.

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

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

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

[EINVAL] *grpid* is out of range.

[EPERM] The caller is not a privileged 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);
    }
}
```

g

AUTHOR

getprivgrp() and **setprivgrp()** were developed by HP.

SEE ALSO

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

g

NAME

getrlimit(), setrlimit() - control maximum resource consumption

SYNOPSIS

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

int getrlimit(int resource, struct rlimit *rlp);

int setrlimit(int resource, const struct rlimit *rlp);
```

DESCRIPTION

Limits on the consumption of a variety of resources by the calling process may be obtained with **getrlimit()** and set with **setrlimit()**. Each call to either **getrlimit()** or **setrlimit()** identifies a specific resource to be operated upon as well as a resource limit. A resource limit is represented by an **rlimit** structure, pointed to by the *rlp* argument and includes the following members:

```
rlim_t rlim_cur;        /* Current (soft) limit */
rlim_t rlim_max;        /* Hard limit */
```

The **rlim_cur** member specifies the current or soft limit and the **rlim_max** member specifies the maximum or hard limit. Soft limits may be changed by a process to any value that is less than or equal to the hard limit. A process may (irreversibly) lower its hard limit to any value that is greater than or equal to the soft limit. Only a process with appropriate privileges can raise a hard limit. Both hard and soft limits can be changed in a single call to **setrlimit()** subject to the constraints described above.

The value **RLIM_INFINITY**, defined in <**sys/resource.h**>, is considered to be larger than any other limit value. If a call to **getrlimit()** returns **RLIM_INFINITY** for a resource, it means the implementation does not enforce limits on that resource. Specifying **RLIM_INFINITY** as any resource limit value on a successful call to **setrlimit()** inhibits enforcement of that resource limit.

The following resources are defined:

RLIMIT_CORE — This is the maximum size of a core file in bytes that may be created by a process. A limit of 0 will prevent the creation of a core file. If this limit is exceeded, the writing of a core file will terminate at this size.

RLIMIT_CPU — This is the maximum amount of CPU time in seconds allowed for a UNIX 95 conforming application. If this limit is exceeded, **SIGXCPU** is generated for the application. The default action for a UNIX 95 conforming application is to kill the process and leave a core file. If the process is blocking, catching or ignoring **SIGXCPU**, the behavior is unspecified. If the application is a Classic HP-UX application, the kernel will not send the signal as a result of exceeding the CPU limit. However, if this signal is sent explicitly to a Classic HP-UX application by another application or via the **kill -XCPU** command, this signal will be delivered and the default action will be taken. In order for an application to be UNIX 95, it must be linked with **unix95.o** either directly or indirectly. For example:

```
% cc /usr/lib/unix95.o prog.c
```

Or,

```
% export UNIX95=1
% cc prog.c
```

RLIMIT_DATA — This is the maximum size of a process' data segment in bytes. If this limit is exceeded, the **brk()**, **malloc()**, and **sbrk()** functions will fail with **errno** set to ENOMEM.

RLIMIT_FSIZE — This option is only applicable to UNIX 95 conforming applications. Please see **RLIMIT_CPU** option above for explanation on UNIX 95 conforming applications. This is the maximum size of a file in bytes that may be created by a process. A limit of 0 will prevent the creation of a file. If a write or truncate operation would cause this limit to be exceeded, **SIGXFSZ** is generated for the process. If the process is blocking, catching or ignoring **SIGXFSZ,** continued attempts to increase the size of a file from end-of-file to beyond the limit will fail with **errno** set to EFBIG.

RLIMIT_NOFILE This is a number one greater than the maximum value that the system may assign to a newly-created descriptor. If this limit is exceeded, functions that allocate new file descriptors may fail with **errno** set to EMFILE. This limit constrains the number of file descriptors that a process may allocate.

RLIMIT_STACK This is the maximum size of a process' stack in bytes. The implementation will not automatically grow the stack beyond this limit. If this limit is exceeded, **SIGSEGV** is generated for the process. If the process is blocking or ignoring **SIGSEGV**, or is catching **SIGSEGV** and has not made arrangements to use an alternate stack, the disposition of **SIGSEGV** will be set to **SIG_DFL** before it is generated.

RLIMIT_AS This is the maximum size of a process' total available memory, in bytes. If this limit is exceeded, the **brk()**, **malloc()**, **mmap()**, and **sbrk()** functions will fail with **errno** set to ENOMEM. In addition, the automatic stack growth will fail with the effects outlined above.

g

RETURN VALUE

Upon successful completion, **getrlimit()** and **setrlimit()** return 0. Otherwise, these functions return −1 and set **errno** to indicate the error.

ERRORS

The **getrlimit()** and **setrlimit()** functions will fail if:

[EINVAL] An invalid resource was specified; or in a **setrlimit()** call, the new *rlim_cur* exceeds the new *rlim_max*.

[EFAULT] The address specified for *rlp* is invalid. Reliable detection of this error is implementation dependent.

[EPERM] The limit specified to **setrlimit()** would have raised the maximum limit value, and the calling process does not have appropriate privileges.

The **setrlimit()** function may fail if:

[EINVAL] The limit specified cannot be lowered because current usage is already higher than the limit.

[EPERM] The *rlp* argument specified a hard or soft limit higher than the current hard limit value, and the caller does not have the appropriate privileges.

[EINVAL] A user with appropriate privileges has attempted to raise *rlp->rlim_cur* or *rlp->rlim_max* to a value greater than the system is capable of supporting.

[EINVAL] The value of *rlp->rlim_cur* is less than the number of file descriptors the process already has allocated.

[EINVAL] The value of *rlp->rlim_max* is less than the current soft limit.

WARNINGS

The maximum size of a file returned by **getrlimit()** is in terms of bytes. The maximum size of a file returned by **ulimit** (see *ulimit*(2)) with **UL_GETFSIZE** is in terms of blocks of size 512 bytes. The value returned by **ulimit** with **UL_GETFSIZE** may thus have to be rounded down to a multiple of 512.

AUTHOR

getrlimit() and **setrlimit()** were developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

brk(2), exec(2), fork(2), getdtablesize(), getrlimit64(2), malloc(3C), open(2), setrlimit64(2), sigaltstack(2), sysconf(2), ulimit(2), <stropts.h>, <sys/resource.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

g

NAME

getrusage - get information about resource utilization

SYNOPSIS

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

int getrusage(int who, struct rusage *r_usage);
```

DESCRIPTION

The **getrusage()** function provides measures of the resources used by the current process or its terminated and waited-for child processes. If the value of the who argument is **RUSAGE_SELF**, information is returned about resources used by the current process. If the value of the who argument is **RUSAGE_CHILDREN**, information is returned about resources used by the terminated and waited-for children of the current process. If the child is never waited for (for instance, if the parent has **SA_NOCLDWAIT** set or sets **SIGCHLD** to **SIG_IGN**), the resource information for the child process is discarded and not included in the resource information provided by **getrusage()**. The *r_usage* argument is a pointer to an object of type **struct rusage** in which the returned information is stored.

If the current process is multi-threaded, **getrusage()** returns the information about the resources used by all the active and the reaped threads/light-weight processes in the current process if the value of the *who* argument is **RUSAGE_SELF**. If the value of the *who* argument is **RUSAGE_CHILDREN** in a multi-threaded process, information returned is the same as it would be for a single-threaded process.

RETURN VALUE

Upon successful completion, **getrusage()** returns **0**. Otherwise, **-1** is returned, and **errno** is set to indicate the error.

ERRORS

The **getrusage()** function will fail if:

[EINVAL] The value of the who argument is not valid.

SEE ALSO

exit(2), sigaction(2), time(1), times(2), wait(1), <sys/resource.h>.

NAME
getsid() - get session ID

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

pid_t getsid (pid_t pid);
```

DESCRIPTION
The **getsid()** function returns the session ID of the specified process. If *pid* is **0**, 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.

RETURN VALUE
Upon successful completion, **getsid()** returns the session ID of the specified process. Otherwise, it returns a value of **−1** and sets **errno** to indicate the error.

ERRORS
If the **getsid()** function fails, it sets **errno** (see *errno*(2)) to one of the following values:

[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*.

SEE ALSO
exec(2), fork(2), getpgid(2), getpid(2), setpgid(2), setsid(2), tcgetsid(3C).

g

g

NAME

getsockname - get socket address

SYNOPSIS

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

AF_CCITT only:

```
#include <x25/x25addrstr.h>

int getsockname(int s, void *addr, int *addrlen);
```

_XOPEN_SOURCE_EXTENDED only (UNIX 98)

```
int getsockname(int s, struct sockaddr *addr, socklen_t *addrlen);
```

Obsolescent _XOPEN_SOURCE_EXTENDED only (UNIX 95)

```
int getsockname(int s, struct sockaddr *addr, size_t *addrlen);
```

DESCRIPTION

getsockname() returns the local address of the socket indicated by *s*, where *s* is a socket descriptor. *addr* points to a socket address structure in which this address is returned. *addrlen* points to a variable that should be initialized to indicate the size of the address structure. On return it contains the actual size of the address returned (in bytes). If *addr* does not point to enough space to contain the whole address of the socket, only the first *addrlen* bytes of the address are returned.

AF_CCITT only:

The **x25_host[]** field of the *addr* struct returns the X.25 addressing information of the local socket *s*. The **x25ifname[]** field of the *addr* struct contains the name of the local X.25 interface through which the call arrived.

RETURN VALUE

Upon successful completion, **getsockname()** returns 0; otherwise, it returns −1 and sets **errno** to indicate the error.

ERRORS

getsockname() fails if any of the following conditions are encountered:

[EBADF]	*s* is not a valid file descriptor.
[ENOTSOCK]	*s* is a valid file descriptor, but it is not a socket.
[ENOBUFS]	No buffer space is available to perform the operation.
[EFAULT]	*addr* or *addrlen* are not valid pointers.
[EINVAL]	The socket has been shut down.
[EOPNOTSUPP]	Operation not supported for AF_UNIX sockets.

OBSOLESCENCE

Currently, the **socklen_t** and **size_t** types are the same size. This is compatible with both the UNIX 95 and UNIX 98 profiles. However, in a future release, **socklen_t** might be a different size. In that case, passing a **size_t** pointer will evoke compile-time warnings, which must be corrected in order for the application to behave correctly. Applications that use **socklen_t** now, where appropriate, will avoid such migration problems. On the other hand, applications that need to be portable to the UNIX 95 profile should follow the X/Open specification (see *xopen_networking*(7)).

FUTURE DIRECTION

Currently, the default behavior is the **HP-UX BSD Sockets**; however, it might be changed to **X/Open Sockets** in a future release. At that time, any **HP-UX BSD Sockets** behavior that is incompatible with **X/Open Sockets** might be obsoleted. Applications that conform to the X/Open specification now will avoid migration problems (see *xopen_networking*(7)).

MULTITHREAD USAGE

The **getsockname()** system call is thread-safe. It has a cancellation point; and it is async-cancel safe, async-signal safe, and fork-safe.

AUTHOR

getsockname() was developed by HP and the University of California, Berkeley.

SEE ALSO

bind(2), socket(2), getpeername(2), inet(7F), af_ccitt(7F), xopen_networking(7).

STANDARDS CONFORMANCE

getsockname(): XPG4

g

NAME

getsockopt(), setsockopt() - get and set options on sockets

SYNOPSIS

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

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

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

_XOPEN_SOURCE_EXTENDED Only (UNIX 98)

```
int getsockopt(
    int          s,
    int          level,
    int          optname,
    void        *optval,
    socklen_t   *optlen
);

int setsockopt(
    int          s,
    int          level,
    int          optname,
    const void  *optval,
    socklen_t    optlen
);
```

Obsolescent _XOPEN_SOURCE_EXTENDED Only (UNIX 95)

```
int getsockopt(
    int          s,
    int          level,
    int          optname,
    void        *optval,
    size_t      *optlen
);

int setsockopt(
    int          s,
    int          level,
    int          optname,
    const void  *optval,
    size_t       optlen
);
```

DESCRIPTION

The **getsockopt()** and **setsockopt()** system calls manipulate options associated with a socket. The socket is identified by the socket descriptor *s*. Options can exist at multiple protocol levels, and they are always present at the uppermost "socket" level (see *socket*(2)).

When manipulating socket options, the level at which the option resides (*level*) and the name of the option (*optname*) must be specified. To manipulate options at the "socket" level, *level* is specified as **SOL_SOCKET**. To specify options at another level, *level* should be the protocol number specified in <**netinet/in.h**> (for example, **IPPROTO_TCP**).

The parameters *optval* and *optlen* specify the value of the option. *optval* is the address of the data structure that contains the option value, and *optlen* is the length of the data structure. The type and value of the data structure that *optval* points to depends on the option. For "boolean" options, the value may be zero (not set) or non-zero (set). The value of other options depends on the purpose of the option. Usually, neither *optval* nor *optlen* may be the NULL address or zero; see individual protocol manual entries for any exceptions, such as *tcp*(7P) and *ip*(7P).

For **setsockopt()**, *optval* and *optlen* are used to pass information from the application to the system. *optval* is the address of a location in memory that contains the option information to be passed to the system. The parameter *optlen* is an integer value that specifies the size, in bytes, of the data structure pointed to by *optval*.

For **getsockopt()**, *optval* and *optlen* are used to pass information from the system to the application. The parameter *optlen* is the address of a variable. Before calling **getsockopt()**, the application should set the value of the variable to the maximum size, in bytes, of the data structure pointed to by *optval*. Normally, upon return, the variable pointed to by *optlen* is set to the actual size the data returned in the structure pointed to by *optval*, if **getsockopt()** returns without error.

The following “socket” level option names (*optname*) are defined in <**sys/socket.h**>. The type of the variable pointed to by *optval* is indicated in parentheses. Options for other protocol levels are described in the individual protocol manual pages, such as *tcp*(7P) and *ip*(7P).

SO_ACCEPTCONN (**int**; boolean) Returns a non-zero value if socket listening is enabled, otherwise returns a zero value.

SO_BROADCAST (**int**; boolean; AF_INET SOCK_DGRAM sockets only) Allows the application to send messages to a broadcast address. **Default**: disallowed.

SO_DEBUG (**int**; boolean; AF_INET SOCK_STREAM sockets only) Enables or disables the recording of internal debug information. **Default**: disabled.

SO_DONTROUTE (**int**; boolean; AF_INET sockets only) Causes outbound messages to bypass normal routing facilities. Instead, messages are sent through the appropriate network interface based on the network portion of the destination address. **Default**: disabled.

SO_ERROR (**int**) Returns any pending error on the socket, and clears the error status. The value returned by **SO_ERROR** would be the value of **errno** after the next socket data transfer system call.

SO_KEEPALIVE (**int**; boolean; AF_INET SOCK_STREAM sockets only) If enabled, keeps an otherwise idle TCP connection active. **Default**: disabled.

SO_LINGER (**struct linger**; AF_INET SOCK_STREAM sockets only) Controls whether or not an application "lingers" (waits) if there are untransmitted data in the send socket buffer when the socket is closed. The data type **struct linger** is defined in <**sys/socket.h**>. **Default**: disabled, as if **l_onoff** had been set to zero. (See details below.)

SO_OOBINLINE (**int**; boolean; AF_INET SOCK_STREAM sockets only) If enabled, specifies that out-of-band data (TCP "urgent data") should be left "in-line" among the normal data stream. Otherwise, one byte of out-of-band data is pulled out of the data stream, and it is accessible only by setting MSG_OOB in the *flags* parameter when the application reads the data (see *recv*(2)). **Default**: disabled.

SO_RCVBUF (**int**) Specifies the maximum size, in bytes, of the receive socket buffer. For SOCK_DGRAM sockets, the receive buffer size may limit the maximum size of messages that the socket can receive. **Default**: protocol-dependent; see individual protocol manual entries, such as *tcp*(7P) and *udp*(7P).

SO_REUSEADDR (**int**; boolean; AF_INET sockets only) If enabled, allows a local address to be reused in subsequent calls to **bind()**. **Default**: disallowed.

SO_REUSEPORT (**int**; boolean; AF_INET sockets only) If enabled, allows a local address and port to be reused in subsequent calls to **bind()**. **Default**: disallowed.

SO_SNDBUF (**int**) Specifies the maximum size, in bytes, of the send socket buffer. For SOCK_STREAM sockets, the send buffer size limits how much data can be queued for transmission before the application is blocked. For SOCK_DGRAM sockets, the send buffer size may limit the maximum size of messages that the application can send through the socket. **Default**: protocol-dependent; see individual protocol manual entries, such as *tcp*(7P) and *udp*(7P).

SO_TYPE (**int**) Returns the socket type.

SO_USELOOPBACK (**int**; boolean) Not used internally; provided only for compatibility.

Setting the **SO_BROADCAST** option allows the application to send messages through the SOCK_DGRAM socket to a broadcast destination address.

If **SO_DONTROUTE** is set, the system does not use the network routing tables when determining which interface to use to send an outbound message. Instead, the system sends the message through the interface whose network address matches the network portion of the destination address. If **SO_DONTROUTE** is not set (default), the system uses the network routing tables.

g

If **SO_KEEPALIVE** is disabled (default), a TCP connection may remain idle until the connection is released at the protocol layer. If **SO_KEEPALIVE** is enabled and the connection has been idle for two hours, TCP sends a packet to the remote socket, expecting the remote TCP to acknowledge that the connection is still active. If the remote TCP does not respond in a timely manner, TCP continues to send keepalive packets according to the normal retransmission algorithm. If the remote TCP does not respond within a particular time limit, TCP drops the connection. The next socket system call (for example, **recv()**) returns an error, and **errno** is set to ETIMEDOUT.

SO_LINGER controls the actions to be taken when there are untransmitted data in a SOCK_STREAM send socket buffer when the socket is closed, either due to an explicit call to **close()** or because the application terminates normally or abnormally. The action is determined by the values of members of the **struct linger** data structure pointed to by *optval* in a call to **setsockopt()**. The data type **struct linger** is defined in <**sys/socket.h**>. If **l_onoff** is zero (the default action), **close()** returns immediately, but the system tries to transmit any unsent data and release the protocol connection gracefully. If **l_onoff** is non-zero and **l_linger** is zero, **close()** returns immediately, any unsent data is discarded, and the protocol connection is aborted. If both **l_onoff** and **l_linger** are non-zero, **close()** does not return until the system has tried to transmit all unsent data and release the connection gracefully. In that case, **close()** can return an error, and **errno** may be set to ETIMEDOUT, if the system is unable to transmit the data after a protocol-defined time limit. Note that the value of **l_linger** is treated simply as a boolean; a non-zero value is *not* interpreted as a time limit(see **_XOPEN_SOURCE_EXTENDED Only** below). **SO_LINGER** does not affect the actions taken when the function **shutdown()** is called.

If **SO_OOBINLINE** is set, out-of-band data (TCP "urgent data") is left "in-line" among the normal data stream. In that case, the SIOCATMARK **ioctl()** request must be used to determine if the inbound data stream has been read up to the point where the out-of-band data begins. If multiple transmissions of out-of-band data are received before the application reads them, all of the data is left in-line; however, SIOCATMARK indicates the location of only the last transmission of out-of-band data. If **SO_OOBINLINE** is not set (default), only one byte of out-of-band is saved. This byte is pulled out of the normal data stream, and it is accessible only by setting MSG_OOB in the *flags* parameter when the application reads the data (see *recv*(2)). In that case, if multiple transmissions of out-of-band data are received before the application reads them, previous bytes of out-of-band data are lost.

Setting the **SO_REUSEADDR** option allows the local socket address to be reused in subsequent calls to **bind()**. This permits multiple SOCK_STREAM sockets to be bound to the same local address, as long as all existing sockets with the desired local address are in a connected state before **bind()** is called for a new socket. For SOCK_DGRAM sockets, **SO_REUSEADDR** allows multiple sockets to receive UDP multicast datagrams addressed to the bound port number. For all SOCK_DGRAM sockets bound to the same local address, **SO_REUSEADDR** must be set *before* calling **bind()**.

Setting the **SO_REUSEPORT** option allows multiple SOCK_DGRAM sockets to share the same address and port. Each one of those sockets, including the first one to use that port, must specify this option before calling **bind()**.

SO_RCVBUF and **SO_SNDBUF** specify the maximum number of bytes that the system may allocate, as needed, for the receive and send buffers, respectively. These limits are merely approximate because of the way in which memory is allocated. For example, a large number of small transmissions may require more memory than the sum of the number of data bytes sent. The default receive and send buffer sizes are protocol-specific. For more information, see the appropriate manual entries, such as *tcp*(7P) and *udp*(7P).

For SOCK_STREAM sockets, larger buffer sizes can improve performance. An application can increase the size of the receive buffer at any time; however, it can decrease the receive buffer size only prior to calling **connect()** or **listen()**. An application can increase or decrease the send buffer at any time.

For SOCK_DGRAM sockets, the size of the receive and send buffers limits the size of the maximum datagram that can be received and sent, respectively. These limits include socket buffer space that is also used to save the sender's socket address (**struct sockaddr**) which is associated with each datagram transmission. The sender's socket address can be returned in the *from* parameter when **recvfrom()** is called (see *recv*(2)).

AF_CCITT

SO_SNDBUF and **SO_RCVBUF** are the only options supported for sockets of the AF_CCITT address family.

_XOPEN_SOURCE_EXTENDED Only

The value of **l_linger** in the *linger* structure is interpreted as a time limit in seconds.

RETURN VALUE

getsockopt() and **setsockopt()** return the following values:

- **0** Successful completion.
- **-1** Failure. **errno** is set to indicate the error.

ERRORS

If **getsockopt()** or **setsockopt()** fails, **errno** is set to one of the following values:

[EBADF]	The argument *s* is not a valid descriptor.
[EFAULT]	The *optval* or *optlen* address is not valid.
[EINVAL]	The *level* or *optlen* value is not valid; or *optval* is the NULL address; or the protocol connection has been released.
[ENOBUFS]	Insufficient memory is available for internal system data structures.
[ENOPROTOOPT]	The option is not recognized at the specified option level.
[ENOTSOCK]	The argument *s* is not a socket descriptor.
[EOPNOTSUPP]	The option is not supported by the socket family or socket type.

OBSOLESCENCE

Currently, the **socklen_t** and **size_t** types are the same size. This is compatible with both the UNIX 95 and UNIX 98 profiles. However, in a future release, **socklen_t** might be a different size. In that case, passing a **size_t** pointer will evoke compile-time warnings, which must be corrected in order for the application to behave correctly. Applications that use **socklen_t** now, where appropriate, will avoid such migration problems. On the other hand, applications that need to be portable to the UNIX 95 profile should follow the X/Open specification (see *xopen_networking*(7)).

FUTURE DIRECTION

Currently, the default behavior is the **HP-UX BSD Sockets**; however, it might be changed to **X/Open Sockets** in a future release. At that time, any **HP-UX BSD Sockets** behavior that is incompatible with **X/Open Sockets** might be obsoleted. Applications that conform to the X/Open specification now will avoid migration problems (see *xopen_networking*(7)).

MULTITHREAD USAGE

The **getsockopt()** and **setsockopt()** system calls are thread-safe. They each have a cancellation point; and they are async-cancel safe, async-signal safe, and fork-safe.

AUTHOR

getsockopt() and **setsockopt()** were developed by HP and the University of California, Berkeley.

g

SEE ALSO

socket(2), getprotoent(3N), af_ccitt(7F), tcp(7P), udp(7P), unix(7P), xopen_networking(7).

STANDARDS CONFORMANCE

getsockopt() : XPG4

g

NAME

gettimeofday - get the date and time

SYNOPSIS

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

int gettimeofday(struct timeval *tp, void *tzp);
```

DESCRIPTION

The **gettimeofday()** function obtains the current time, expressed as seconds and microseconds since 00:00 Coordinated Universal Time (UTC), January 1, 1970, and stores it in the **timeval** structure pointed to by *tp*. The resolution of the system clock is unspecified.

If *tzp* is not a null pointer, the behaviour is unspecified.

RETURN VALUE

The **gettimeofday()** function returns 0 and no value is reserved to indicate an error.

ERRORS

No errors are defined.

SEE ALSO

ctime(3C), ftime(2), <sys/time.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

g

HP-UX EXTENSIONS

g

SYNOPSIS

```
#include <time.h>

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

DESCRIPTION

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 UTC */
    int  tz_dsttime;      /* type of DST correction to apply */
};
```

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

Security Restrictions

Only a user with appropriate privileges can set the time of day.

RETURN VALUE

gettimeofday() return the following values:

- **0** Successful completion.
- **-1** Failure. **errno** is set to indicate the error.

ERRORS

gettimeofday() fails, **errno** is set to the following value.

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

EXAMPLES

The following example calls **gettimeofday()** 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;
```

WARNINGS

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

AUTHOR

`gettimeofday()` was developed by the University of California, Berkeley, and SecureWare Inc.

SEE ALSO

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

g

NAME

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

SYNOPSIS

```
#include <unistd.h>

uid_t getuid(void);

uid_t geteuid(void);

gid_t getgid(void);

gid_t getegid(void);
```

DESCRIPTION

The following functions return the information indicated:

getuid() Real-user-ID of the calling process.

geteuid() Effective-user-ID of the calling process.

getgid() Real-group-ID of the calling process.

getegid() Effective-group-ID of the calling process.

No means is available for ascertaining the saved-user-ID or saved-group-ID of a process.

SEE ALSO

setuid(2).

STANDARDS CONFORMANCE

getuid(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

getegid(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

geteuid(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

getgid(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

g

NAME

ioctl - control device

SYNOPSIS

```
#include <stropts.h>

int ioctl(int fildes, int request, ... /* arg */);
```

Remarks

The ANSI C "**, ...**" construct denotes a variable length argument list whose optional [or required] members are given in the associated comment (**/* */**).

DESCRIPTION

ioctl() performs a variety of functions on character special files (devices), or regular files and directories on VxFS file systems. 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 which 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 (meaning that the argument is copied from the application to the driver).

IOC_OUT Argument is written by the driver (meaning that the argument is copied from the driver to the application). Ignored if an error occurs.

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 that do not require any data to be passed and requests that 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** accordingly.

_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* must *not* contain any pointers.

RETURN VALUE

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

ioctl() fails if one or more of the following are true: IOC_OUT is ignored if an error occurs.

[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.

i

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 and HP.

SEE ALSO

ioctl(5), arp(7P), socket(7), termio(7).

STANDARDS CONFORMANCE

`ioctl()`: SVID2, SVID3, XPG2

NAME

iscomsec - check if the system has been converted to a trusted system

SYNOPSIS

```
#include <prot.h>

int iscomsec();
```

DESCRIPTION

iscomsec returns a zero (0) if the system is not a trusted system. If the system has been converted to a trusted system, **iscomsec** returns a one (1).

NOTES

iscomsec determines if the system is a trusted system or not by checking the file, **/tcb/files/auth/system/default**. If the file exists, then the system is a trusted system. If the file does not exist, then the system is not a trusted system.

On a trusted system, **/tcb/files/auth/system/default** should never be deleted.

AUTHOR

iscomsec was developed by HP.

FILES

/tcb/files/auth/system/default	Trusted system default file

NAME

kill(), raise() - send a signal to a process or a group of processes

SYNOPSIS

```
#include <signal.h>

int kill(pid_t pid, int sig);

int raise(int sig);
```

DESCRIPTION

The **kill()** system call sends a signal to a process or a group of processes, as 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**.

The **raise()** system call sends a signal 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 a user who has appropriate privileges.

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.

k

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 a user who has appropriate privileges. *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 a user who has appropriate privileges, *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.

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

If **kill()** fails, no signal is sent. **errno** is set to one of the following values.

- [EINVAL] *sig* is neither a valid signal number nor zero.
- [EINVAL] *sig* is **SIGKILL** or **SIGSTOP** and *pid* is **1** (process 1).
- [EPERM] The user ID of the sending process is not a user who has appropriate privileges 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 and the real or effective user ID does not match the real or saved user ID of the receiving process.
- [ESRCH] No process or process group can be found corresponding to that specified by *pid*.

If **raise()** fails, no signal is sent. **errno** is set to one of the following values.

[EINVAL] *sig* is neither a valid signal number nor zero.

APPLICATION USAGE

Threads Considerations

kill() can be used to post signals to another process but cannot be used to post signals to a specific thread in another process. For information on posting signals to specific threads within the same process, see *pthread_kill*(3T).

LWP (Lightweight Processes) Considerations

Signals cannot be posted to specific LWPs in another process.

AUTHOR

kill() was developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

kill(1), getpid(2), setsid(2), signal(2), sigqueue(2), pthread_kill(3T).

STANDARDS CONFORMANCE

kill(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

raise(): AES, SVID3, XPG4, ANSI C

NAME

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

SYNOPSIS

```
#include <signal.h>

int killpg(int pgrp, int sig);

int getpgrp(int pid);

int setpgrp(int pid, int pgrp);

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

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

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()`**.

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*(3C).

APPLICATION USAGE

Threads Considerations

The signal disposition (such as **catch/ignore/default**) established by **`sigvec()`** and **`signal()`** is shared by all threads in the process.

For more information regarding signals and threads, refer to *signal*(5).

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(3C), sigstack(2), signal(5).

NAME

link() - link to a file

SYNOPSIS

```
#include <unistd.h>

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

DESCRIPTION

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

RETURN VALUE

Upon successful completion, **link()** returns zero. Otherwise, it returns −1 and sets **errno** (see *errno*(2)) to indicate the error.

ERRORS

The **link()** system call fails and no link is created if one or more of the following is true:

[EACCES]	A component of either path prefix denies search permission.
[EACCES]	The requested link requires writing in a directory that does not permit writing.
[EDQUOT]	The user's disk quota block limit has been reached for this file system.
[EEXIST]	The link named by *path2* exists.
[ENOENT]	The file named by *path1* does not exist.
[ENOENT]	A component of either path prefix does not exist.
[ENOENT]	*path2* points to a null path name.
[ENOSPC]	The directory to contain the file cannot be extended.
[ENOTDIR]	A component of either path prefix is not a directory.
[EPERM]	The file named by *path1* is a directory and the effective user ID is not a user who has appropriate privileges. Some file systems return this error whenever *path1* names a directory, regardless of the user ID.
[EXDEV]	The link named by *path2* and the file named by *path1* are on different logical devices (file systems).
[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 is implementation dependent.
[ENOENT]	*path1* or *path2* is null.
[EMLINK]	The maximum number of links to a file would be exceeded.
[ENAMETOOLONG]	Either the specified path exceeds **PATH_MAX** bytes, or a component of either specified path exceeds **NAME_MAX** while **POSIX_NO_TRUNC** is in effect.
[ELOOP]	Too many symbolic links were encountered in translating either path name.

DEPENDENCIES

Series 700

If *path2* names a symbolic link, **link()** fails without creating the link, it returns −1, and sets **errno** to the following value:

[EEXIST]	*path2* names a symbolic link.

SEE ALSO

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

l

STANDARDS CONFORMANCE

`link()`: AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

lio_listio() - start a list of asynchronous I/O operations

SYNOPSIS

```
#include <aio.h>

int lio_listio(int mode, struct aiocb * const list[], int nent, struct
    sigevent *sig);
```

DESCRIPTION

The **lio_listio()** function allows the calling process to request a list of asynchronous I/O operations with a single function call. The function call returns when all operation requests have been enqueued for processing. Once enqueued, processing of the operations may proceed concurrently with execution of the calling process thread.

The **list** argument is an array of **nent** pointers to **aiocb** structures. Each **aiocb** in **list** is treated as if it were being handled in a separate call to **aio_read()** or **aio_write()** depending on the value of its **aio_lio_opcode**. When **aio_lio_opcode** is **LIO_READ**, the **aiocb** is treated as though it had been referenced in a call to **aio_read()**, and the **aio_fildes**, **aio_buf**, and **aio_nbytes** fields are interpreted accordingly. When **aio_lio_opcode** is **LIO_WRITE**, the **aiocb** is treated as though it had been referenced in a call to **aio_write()**, and the **aio_fildes, aio_buf,** and **aio_nbytes** fields are interpreted accordingly. If **aio_lio_opcode** is **LIO_NOP**, nothing is enqueued.

l

If an error condition is detected that prevents the list from being processed, **lio_listio()** returns **-1** and sets **errno** to indicate the cause of the failure. If any requests are enqueued by the call to **lio_listio()**, and **mode** is **LIO_WAIT**, then the function returns only after all enqueued operation requests have completed. The *sig* argument of the call is ignored. If **mode** is **LIO_NOWAIT**, the function returns as soon as all requests are enqueued. The **sigevent** action specified by *sig* is performed after all enqueued requests have completed.

Once the requested operations have been successfully enqueued, an **aio_error()** and **aio_return()** function referencing the corresponding **aiocb** from **list** must be used to determine their status and any error conditions, including those normally reported by **read()** or **write()**, as appropriate. Requests remain enqueued and consume process and system resources until **aio_return()** is called for each one.

Re-using, altering the contents of, or deallocating memory associated with the **list** or any **aiocb** referenced in **list** or the buffer referred to by **list[n]->aio_buf** while an asynchronous I/O operation is outstanding may produce unpredictable results because **aio_return()** has not been called for the **aiocb**.

To use this function, link in the realtime library by specifying **-lrt** on the compiler or linker command line.

RETURN VALUE

When **LIO_NOWAIT** is set, **lio_listio()** returns the following values:

- **0** Success. All of the non-empty operations, if any, were successfully enqueued.
- **-1** Failure or partial success. At least one requested operation was either not enqueued or completed with an error before the **lio_listio()** function call returned. **errno** is set to indicate the error.

When **LIO_WAIT** is set, **lio_listio()** returns the following values:

- **0** Success. All of the non-empty operations, if any, were successfully enqueued and completed.
- **-1** Failure or partial success. At least one requested operation was either not enqueued or completed with an error. **errno** is set to indicate the error.

The three **errno** values **EAGAIN**, **EINTR**, and **EIO** are the only ones associated with partial success. **aio_error()** and **aio_return()** must be used to determine the outcomes of individual requests.

ERRORS

If **lio_listio()** detects one of the following error conditions, **errno** is set to the indicated value:

[EAGAIN] At least one request could not be queued either because of a resource shortage or because the per-process or system-wide limit on asynchronous I/O operations or asynchronous threads would have been exceeded.

[EINVAL] The **sigevent** specified by *sig* is not valid.

[EINVAL] The **mode** argument is neither **LIO_WAIT** nor **LIO_NOWAIT**.

[EINVAL] The value of the **nent** argument is negative or greater than the maximum value allowed. The maximum value allowed can be obtained using the **sysconf()** call with the argument **_SC_AIO_LISTIO_MAX**.

[EINTR] The **mode** argument was **LIO_WAIT** and a signal was delivered while waiting for the requested operations to complete. This signal may result from completion of one or more of the requested operations and other requests may still be pending or completed.

Once an operation has been enqueued by **lio_listio()**, all of the errors normally reported by the appropriate **read()** or **write()** function and the following errors may be reported asynchronously and returned in a subsequent call to **aio_error()** referencing the **aiocb** pointer supplied in the successful **lio_listio()** call.

[EBADF] The **aiocbp->aio_fildes** was not a valid file descriptor open for reading or writing as appropriate to the **aio_lio_opcode**.

[EINVAL] The value of **aiocbp->aio_reqprio** is not valid, or the value of **aiocbp->aio_nbytes** is invalid, or the file offset implied by **aiocbp->aio_offset** or **aiocbp->aio_offset**+**aiocbp->aio_nbytes** is not valid.

[EIO] One or more of the enqueued operations did not complete successfully.

EXAMPLE

The following code sequence and call to **lio_listio()** starts two asynchronous write operations and one asynchronous read operation and waits for all operations to complete.

```
#include <fcntl.h>
#include <errno.h>
#include <aio.h>
char buf1[4096], buf2[4096], buf3[4096];
int nent;
struct aiocb myaiocb1, myaiocb2, myaiocb3;
struct aiocb *list[] = { &myaiocb1, &myaiocb2, &myaiocb3 };
bzero( &myaiocb1, sizeof (struct aiocb));
bzero( &myaiocb2, sizeof (struct aiocb));
bzero( &myaiocb3, sizeof (struct aiocb));
myaiocb1.aio_fildes = open( "/dev/null", O_RDWR);
myaiocb3.aio_fildes = myaiocb2.aio_fildes = myaiocb1.aio_fildes;
myaiocb1.aio_offset = 0;
myaiocb3.aio_offset = myaiocb2.aio_offset = myaiocb1.aio_offset;
myaiocb1.aio_buf = (void *) buf1;
myaiocb2.aio_buf = (void *) buf2;
myaiocb3.aio_buf = (void *) buf3;
myaiocb3.aio_nbytes = sizeof (buf3);
myaiocb2.aio_nbytes = sizeof (buf2);
myaiocb1.aio_nbytes = sizeof (buf1);
myaiocb1.aio_lio_opcode = myaiocb3.aio_lio_opcode = LIO_WRITE;
myaiocb2.aio_lio_opcode = LIO_READ;
myaiocb1.aio_sigevent.sigev_notify = SIGEV_NONE;
myaiocb2.aio_sigevent.sigev_notify = SIGEV_NONE;
myaiocb3.aio_sigevent.sigev_notify = SIGEV_NONE;
retval = lio_listio( LIO_WAIT, list, 3, NULL );
if (retval) perror("lio_listio:");
while ( nent-- ) { (void) aio_return( list[nent] ); }
```

SEE ALSO

aio_cancel(2), aio_error(2), aio_fsync(2), aio_read(2), aio_return(2), aio_suspend(2), aio_write(2), read(2), write(2), aio(5).

STANDARDS CONFORMANCE

lio_listio() : POSIX Realtime Extensions, IEEE Std 1003.1b

NAME

listen - listen for connections on a socket

SYNOPSIS

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

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

DESCRIPTION

To accept connections, a socket is first created using **socket()**, a queue for incoming connections is activated using **listen()**, and then connections are accepted using **accept()**. **listen()** applies only to unconnected sockets of type SOCK_STREAM. Except for AF_VME_LINK, if the socket has not been bound to a local port before **listen()** is invoked, the system automatically binds a local port for the socket to listen on (see *inet*(7F)). For sockets in the address family AF_CCITT and AF_VME_LINK, the socket *must* be bound to an address by using **bind()** before connection establishment can continue, otherwise an EADDREQUIRED error is returned.

A listen queue is established for the socket specified by the *s* parameter, which is a socket descriptor.

backlog defines the desirable queue length for pending connections. The actual queue length may be greater than the specified *backlog*. If a connection request arrives when the queue is full, the client will receive an ETIMEDOUT error.

backlog is limited to the range of 0 to **SOMAXCONN,** which is defined in **<sys/socket.h>. SOMAXCONN** is currently set to 20. If any other value is specified, the system automatically assigns the closest value within the range. A *backlog* of 0 specifies only 1 pending connection is allowed at any given time.

DEPENDENCIES

AF_CCITT:

Call-acceptance can be controlled by the **X25_CALL_ACPT_APPROVAL ioctl()** call described in RETURN VALUE . Upon successful completion, **listen()** returns 0; otherwise, it returns −1 and sets **errno** to indicate the error.

ERRORS

listen() fails if any of the following conditions are encountered:

[EBADF]	*s* is not a valid file descriptor.
[EDESTADDRREQ]	The socket *s* has not been bound to an address by using **bind()**.
[ENOTSOCK]	*s* is a valid file descriptor but it is not a socket.
[EOPNOTSUPP]	The socket referenced by *s* does not support **listen()**.
[EINVAL]	The socket has been shut down or is already connected (see *socketx25*(7)).

FUTURE DIRECTION

Currently, the default behavior is the **HP-UX BSD Sockets**; however, it might be changed to **X/Open Sockets** in a future release. At that time, any **HP-UX BSD Sockets** behavior that is incompatible with **X/Open Sockets** might be obsoleted. Applications that conform to the X/Open specification now will avoid migration problems (see *xopen_networking*(7)).

MULTITHREAD USAGE

The **listen()** system call is thread-safe. It has a cancellation point; and it is async-cancel safe, async-signal safe, and fork-safe.

AUTHOR

listen() was developed by HP and the University of California, Berkeley.

SEE ALSO

accept(2), connect(2), socket(2), socketx25(7), xopen_networking(7), af_ccitt(7F), af_vme_link(7F), inet(7F).

STANDARDS CONFORMANCE

listen(): XPG4

NAME

lockf - provide semaphores and record locking on files

SYNOPSIS

```
#include <unistd.h>

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

DESCRIPTION

The **lockf()** function allows regions of a file to be used as semaphores (advisory locks) or restricts access to only the locking process (enforcement-mode record locks). Other processes that attempt to access the locked resource either return an error or sleep until the resource becomes unlocked. All locks for a process are released upon the first close of the file, even if the process still has the file opened, and all locks held by a process are released when the process 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 *getprivgrp*(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. 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 result in an error return if not implemented.

F_TEST is used to detect whether 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 which case **errno** is 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, because such locks can exist past the end of the file.

Regions locked with **F_LOCK** or **F_TLOCK** can, in whole or in 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 but the 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** causes the calling process to sleep until the resource is available, whereas **F_TLOCK** returns an EACCES error if the region is already locked by another process.

F_ULOCK requests can, 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 sleep until the entire region is available if **O_NDELAY** is clear, but return −1 with **errno** set if **O_NDELAY** is set. File access by other system functions, such as **exec()**, are not subject to the enforcement policy. Locks on directories, pipes, and special files are advisory only; no enforcement policy is 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()**, **lockf()**, **read()**, or **write()** (see *fcntl*(2), *lockf*(2), *read*(2), and *write*(2)) scan for a deadlock prior to sleeping on a locked resource. Deadlock is not checked for the **wait()** and **pause()** system calls (see *wait*(2) and *pause*(2)), so potential for deadlock is not eliminated. A **creat()** call or an **open()** call with the **O_CREATE** and **O_TRUNC** flags set on a regular file returns error EAGAIN 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()** 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] Either *function* is not one of the functions specified above, or *size* plus current offset produces a negative offset into the file.

[EINVAL] *size* plus current offset cannot be represented correctly by an object of size **off_t.**

[ENOLCK] Either *function* is **F_TLOCK** or **F_LOCK** and the file is an NFS file with access bits set for enforcement mode, or the file is an NFS file and a system error occurred on the remote node.

WARNINGS

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

When a file descriptor is closed, all locks on the file from the calling process are deleted, even if other file descriptors for that file (obtained through **dup()** or **open()**, for example) still exist.

Unexpected results may occur in processes that use buffers in the user address space. The process may later read or 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 can 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 because it is provided for backward compatibility only. This feature may be modified or dropped from future HP-UX releases.

Locks default to advisory mode unless the **setgid** bit of the file permissions is set.

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

lockd(1M), statd(1M), chmod(2), close(2), creat(2), fcntl(2), creat64(2), open(2), pause(2), read(2), stat(2), wait(2), write(2), unistd(5).

STANDARDS CONFORMANCE

lockf(): SVID2, SVID3, XPG2

NAME

lseek - move read/write file pointer; seek

SYNOPSIS

```
#include <unistd.h>

off_t lseek(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 an 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.

For all files that are not character or block special files, the integer returned on successful completion is non-negative. For character or block special files that correspond to disk sections larger than 2 gigabytes, a non-negative integer is returned for successful seeks beyond 2 gigabytes. This value is the resulting file offset as measured in bytes from the beginning of the file, when taken as an unsigned value. **-1** always indicates an error return, even when encountered on greater than 2 gigabyte disk sections. The **lseek()** call succeeds for NFS directories even if the resulting file offset becomes negative.

1

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, socket, or FIFO.
[EINVAL]	*whence* is not one of the supported values.
[EINVAL]	The resulting file offset would be negative.
[EINVAL]	The resulting file *offset* would be a value which cannot be represented correctly in an object of type **off_t**.

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), lseek64(2), open(2), unistd(5).

STANDARDS CONFORMANCE

lseek(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

lstat - get symbolic link status

SYNOPSIS

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

int lstat(const char *path, struct stat *buf);
```

DESCRIPTION

The `lstat()` function has the same effect as `stat()`, except when *path* refers to a symbolic link. In that case `lstat()` returns information about the link, while `stat()` returns information about the file the link references.

For symbolic links, the *st_mode* member will contain meaningful information when used with the file type macros, and the *st_size* member will contain the length of the pathname contained in the symbolic link. File mode bits and the contents of the remaining members of the `stat` structure are unspecified. The value returned in the *st_size* member is the length of the contents of the symbolic link, and does not count any trailing null.

RETURN VALUE

Upon successful completion, `lstat()` returns 0. Otherwise, it returns –1 and sets `errno` to indicate the error.

ERRORS

The `lstat()` function will fail if:

[EACCES]	A component of the path prefix denies search permission.
[EIO]	An error occurred while reading from the file system.
[ELOOP]	Too many symbolic links were encountered in resolving path.
[ENAMETOOLONG]	The length of a pathname exceeds **{PATH_MAX}**, or pathname component is longer than **{NAME_MAX}**.
[ENOTDIR]	A component of the *path* prefix is not a directory.
[ENOENT]	A component of *path* does not name an existing file or *path* is an empty string.
[EOVERFLOW]	The file size in bytes or the number of blocks allocated to the file cannot be represented correctly in the structure pointed to by *buf*.

The `lstat()` function may fail if:

[ENAMETOOLONG]	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds **{PATH_MAX}**.

SEE ALSO

fstat(2), readlink(2), stat(2), symlink(2), <sys/stat.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

HP-UX EXTENSIONS

SYNOPSIS

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

DESCRIPTION

If the chosen path name or file descriptor refers to a Multi-Level Directory (MLD), and the process does not have the multilevel effective privilege, the i-node number returned in *st_ino* is the i-node of the MLD itself.

The parameters for the **lstat()** function is as follows:

path is a pointer to a path name of any file within the mounted file system.(All directories listed in the path name must be searchable.)

buf is a pointer to a **stat** structure, which is where the file status information is stored.

The **stat** structure contains the following members:

```
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 sysfs(2) */
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 */
long     st_blksize;    /* File system block size */
uint     st_acl:1;      /* Set if the file has optional */
                        /* access control list entries */
                        /* HFS File Systems only */
```

(Note that the position of items in this list does not necessarily reflect the order of the members in the structure.)

ERRORS

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

No ERROR for the following:

[EIO] An error occurred while reading from the file system.

NFS

The *st_basemode* and *st_acl* fields are zero on files accessed remotely. *st_acl* field is applicable to HFS File Systems only.

WARNINGS

Access Control Lists - HFS File Systems only

Access control list descriptions in this entry apply only to HFS file systems on standard HP-UX operating systems.

DEPENDENCIES (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.

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), lstat64(2), mknod(2), pipe(2), read(2), rename(2), setacl(2), sysfs(2), time(2), truncate(2), unlink(2), utime(2), write(2), stat(5), privileges(5), acl(5), stat(5).

STANDARDS CONFORMANCE

lstat(): AES, SVID3

NAME

madvise() - advise the system of a process's expected paging behavior

SYNOPSIS

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

int madvise(
     caddr_t addr,
     size_t len,
     int behav );
```

DESCRIPTION

The **madvise** system call permits a process to advise the system about its expected future behavior in referencing a mapped file or an anonymous memory region. Certain implementations can use this information to optimize the use of resources.

addr and *len* specify the address and length in bytes of the region to which the advice refers. For **MADV_DONTNEED**, the address and length must be contained within a successful call to **mmap()** (see *mmap*(2)); otherwise, **madvise()** fails with an [EINVAL] error.

The *behav* argument is one the following flags defined in the header **<sys/mman.h>**:

MADV_NORMAL
: Removes any previous advice and sets the default behavior. By default, the kernel tracks access patterns on data objects and performs I/Os based on process trends (that is, sequential versus random). Sequential trends cause larger "read-ahead" I/Os, while random accesses reduce the amount of I/O to avoid unnecessary I/O.

MADV_RANDOM
: Informs the kernel that any objects mapped in this range will be accessed in a random matter. The kernel will read only the minimal amount of data to satisfy the user fault.

MADV_SEQUENTIAL
: Informs the kernel that any objects mapped in this range will be accessed in a sequential matter. The kernel will perform the maximum read-ahead for every fault. The kernel does not pay attention to access patterns and trends, but instead assumes sequentiality for every access on the object.

MADV_DONTNEED
: Informs the kernel that the specified range is no longer needed by the process. This allows the kernel to release the physical pages associated with an address range back to the system for use by other processes.

: **MADV_DONTNEED** is restricted to object ranges created with calls to **mmap()**. Attempting to use **MADV_DONTNEED** on an object that was not created using a call to **mmap()** will result in [EINVAL] being returned to the caller.

MADV_WILLNEED
: Will need these pages.

MADV_SPACEAVAIL
: Ensure that resources are reserved.

WARNINGS

The current implementation of **madvise()** defines **MADV_SPACEAVAIL** and **MADV_WILLNEED** as null operations.

RETURN VALUE

madvise() returns the following values:

- **0** Successful completion.
- **-1** Failure. **errno** is set to indicate the error.

ERRORS

If **madvise()** fails, **errno** is set to one of the following values.

[EFAULT] The range specified by (*addr*, *addr+len*) is invalid for a process's address space, or permission was incorrect on the object for the *behav* specified.

m

[EINVAL] *behav* contains an invalid value, or *addr* is not a multiple of the page size as returned by the system call **sysconf(_SC_PAGE_SIZE)**.

[EINVAL] The address range specified by *addr* and *len* was not created by a successful call to **mmap()**.

AUTHOR

madvise() was developed by HP and OSF.

SEE ALSO

mmap(2), sysconf(2).

STANDARDS CONFORMANCE

madvise(): AES, SVID3

NAME

makecontext, swapcontext - manipulate user contexts

SYNOPSIS

`#include <ucontext.h>`

`void makecontext(ucontext_t *ucp, (void *func)(), int argc, ...);`

`int swapcontext(ucontext_t *oucp, const ucontext_t *ucp);`

DESCRIPTION

The **makecontext()** function modifies the context specified by *ucp*, which has been initialized using **getcontext()**. When this context is resumed using **swapcontext()** or **setcontext()**, program execution continues by calling **func()**, passing it the arguments that follow *argc* in the makecontext() call.

Before a call is made to **makecontext()**, the context being modified should have a stack allocated for it. The value of *argc* must match the number of integer arguments passed to **func()**, otherwise the behavior is undefined.

The *uc_link* member is used to determine the context that will be resumed when the context being modified by **makecontext()** returns. The *uc_link* member should be initialized prior to the call to **makecontext()**.

The **swapcontext()** function saves the current context in the context structure pointed to by *oucp* and sets the context to the context structure pointed to by *ucp*.

RETURN VALUE

On successful completion, **swapcontext()** returns 0. Otherwise, −1 is returned and **errno** is set to indicate the error.

WARNINGS

Context APIs are not recommended due to possible compatibility problems from release to release, because context APIs are very architecture-specific. The context APIs "expose" the architecture to the application, such that the application may not be compatible with all releases.

If you must use context APIs, be aware of the following:

- Do not copy the context yourself. It is not contiguous. The context may have pointers that may point back to the original context rather than in the copied context; hence, it will be broken.
- The size of the context will vary in length from release to release.

ERRORS

The **makecontext()** and **swapcontext()** functions will fail if:

[ENOMEM]	The *ucp* argument does not have enough stack left to complete the operation.

SEE ALSO

exit(2), getcontext(2), sigaction(2), sigprocmask(2), <ucontext.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

m

NAME

mkdir - make a directory file

SYNOPSIS

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

int mkdir(const char *path, mode_t mode);
```

DESCRIPTION

The **mkdir()** system call 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 the 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.

m

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

mkdir() returns one of the following values:

0	Successful completion.
-1	Failure. An error code is stored in **errno**.

ERRORS

If **mkdir()** fails, no directory is created and **errno** is set to one of the following values:

[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.

[EDQUOT] User's disk quota block or inode limit has been reached for this file system.

AUTHOR

mkdir() was developed by the University of California, Berkeley.

SEE ALSO

chmod(2), setacl(2), stat(2), umask(2), acl(5), limits(5).

STANDARDS CONFORMANCE

mkdir(): AES, SVID2, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

mknod() - make directory, special, or ordinary file

SYNOPSIS

#include <sys/stat.h>

int mknod(const char *path, mode_t mode, dev_t dev);

DESCRIPTION

The **mknod()** system call 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.

Symbolic constants that define 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 bit-wise inclusive OR of the values of the desired file type, miscellaneous mode bits, and access permissions. See *stat*(5) for a description of the components of the file mode.

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 new file's group ID is set to the group ID of the parent directory. Otherwise, the new file'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)).

The new file is created with three base access-control-list (ACL) entries, corresponding to the file access permission bits (see *acl*(5)).

The *dev* argument 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. The value of *dev* is created by using the **makedev()** macro defined in **<sys/mknod.h>**. The **makedev()** macro takes as arguments the major and minor device numbers, and returns a device identification number which is of type **dev_t**. The value and interpretation of the major and minor device numbers are implementation-dependent. For more information, see *mknod*(5) and the System Administration manuals for your system.

m

Only users having appropriate privileges can invoke **mknod()** for file types other than FIFO files.

RETURN VALUE

mknod() returns the following values:

0 Successful completion.
-1 Failure. The new file is not created. **errno** is set to indicate the error.

ERRORS

If **mknod()** fails, **errno** is set to one of the following values.

[EACCES] The directory in which *path* would be created denies write permission, *mode* is for a FIFO file and the caller does not have appropriate privileges.

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

[EDQUOT] The user's disk quota block or inode limit has been reached for this file system.

[EEXIST] The named path already exists.

[EFAULT] The *path* argument points outside the process's allocated address space. 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.

[ENOENT] The *path* argument 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 a user who has appropriate privileges, 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

mknod(1M), chmod(2), exec(2), mkdir(2), setacl(2), umask(2), fs(4), acl(5), mknod(5), stat(5), types(5).

STANDARDS CONFORMANCE

mknod(): SVID2, SVID3, XPG2

m

NAME

mlock() - lock a segment of the process virtual address space in memory

SYNOPSIS

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

int mlock( const void * addr, size_t len) ;
```

DESCRIPTION

The **mlock()** system call allows the calling process to lock a segment of the process virtual address space into memory. Any addressable segment of the process' address space may be locked. Locked segments are immune to all routine swapping.

addr must be a valid address in the process virtual address space. *addr* + *len* must also be a valid address in the process virtual address space.

Locks are applied at page boundaries that encompass the range from *addr* to *addr* + *len*. If any address within the range is not valid, an error is returned and no locks are applied.

munlock() or **munlockall()** can be used to unlock memory segments (or all memory segments) locked with **mlock()**.

Regardless of how many times a process locks a page, a single **munlock()** or **munlockall()** will unlock it. An **munlock()** of a page within a range specified in an **mlock()** call results in only the range specified in the **munlock()** being unlocked.

When memory is shared by multiple processes and mlocks are applied to the same physical page by multiple processes, a page remains locked until the last lock is removed from that page.

Locks applied with **mlock()** are not inherited by a child process.

The effective user ID of the calling process must be a superuser or the user must be a member of a group that has the MLOCK privilege (see *getprivgrp*(2) and *setprivgrp*(1M))

Although **plock()** and the **mlock()** family of functions may be used together in an application, each may affect the other in unexpected ways. This practice is not recommended.

RETURN VALUE

mlock() returns the following values:

- **0** Successful completion.
- **-1** Failure. The requested operation is not performed. **errno** is set to indicate the error.

ERRORS

If **mlock()** fails, **errno** is set to one of the following values:

[ENOMEM] One or more addresses in the specified range is not valid within the process address space.

[EAGAIN] There is not enough lockable memory in the system to satisfy the locking request.

[EINVAL] The **len** parameter was zero.

[EPERM] The effective user ID of the calling process is not a superuser and the user does not belong to a group that has the **MLOCK** privilege.

EXAMPLES

The following call to **mlock()** locks the first 10 pages of the calling process in memory:

```
mlock(sbrk(0), 40960);
```

SEE ALSO

setprivgrp(1M), getprivgrp(2), mlockall(2), munlock(2), munlockall(2), plock(2)

STANDARDS CONFORMANCE

mlock(): POSIX Realtime Extensions, IEEE Std 1003.1b

NAME

mlockall() - lock a process virtual address space in memory

SYNOPSIS

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

int mlockall( constant int flags);
```

DESCRIPTION

The **mlockall()** system call allows the calling process to lock its entire virtual address space into memory, making it immune to all routine swapping.

flags may be one or both of the following:

MCL_CURRENT Lock the current process virtual address space. All addressable pages of the address space are locked.

MCL_FUTURE Lock any future additions to the process virtual address space.

Note that **MCL_FUTURE** does not imply **MCL_CURRENT**.

munlockall() or **munlock()** can be used to unlock all or a portion of the address space locked with **mlockall()**. A single call to **munlockall()** removes all locks from the process virtual address space. An **munlock()** call results in only the specified pages being unlocked.

Regardless of how many times a process locks a page, a single **munlock()** or **munlockall()** will unlock it.

When memory is shared by multiple processes and mlocks are applied to the same physical page by multiple processes, a page remains locked until the last lock is removed from that page.

Locks and **MCL_FUTURE** applied with **mlockall()** are not inherited by a child process.

The effective user ID of the calling process must be a superuser or the user must be a member of a group that has the MLOCK privilege (see *getprivgrp*(2) and *setprivgrp*(1M)).

Although **plock()** and the **mlock()** family of functions may be used together in an application, each may affect the other in unexpected ways. This practice is not recommended.

m

RETURN VALUE

mlockall() returns the following values:

0 Successful completion.
-1 Failure. The requested operation is not performed. **errno** is set to indicate the error.

ERRORS

If **mlockall()** fails, **errno** is set to one of the following values:

[EINVAL] The **flags** field did not contain either **MCL_CURRENT** and/or **MCL_FUTURE**.

[EAGAIN] There is not enough lockable memory in the system to satisfy the locking request.

[EPERM] The effective user ID of the calling process is not a superuser and the user does not belong to a group that has the **MLOCK** privilege.

EXAMPLES

The following call to **mlockall()** locks the entire process virtual address space in memory and ensures that any future additions to the address space will also be locked in memory:

```
mlockall( (MCL_CURRENT | MCL_FUTURE) );
```

SEE ALSO

setprivgrp(1M), getprivgrp(2), mlock(2), munlock(2), munlockall(2), plock(2).

STANDARDS CONFORMANCE

mlockall(): POSIX Realtime Extensions, IEEE Std 1003.1b

NAME

mmap - map pages of memory

SYNOPSIS

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

void *mmap(void *addr, size_t len, int prot, int flags,
           int fildes, off_t off);
```

DESCRIPTION

Note: This manpage contains HP-UX extensions.

The **mmap()** function establishes a mapping between a process' address space and a file. The format of the call is as follows:

```
pa=mmap(addr, len, prot, flags, fildes, off);
```

The **mmap()** function establishes a mapping between the process' address space at an address *pa* for *len* bytes and the file associated with the file descriptor *fildes* at offset *off* for *len* bytes. The value of *pa* is an unspecified function of the argument *addr* and values of flags, further described below. A successful **mmap()** call returns *pa* as its result. The address ranges covered by [*pa*, *pa+len*] and [*off*, *off+len*] must be legitimate for the possible (not necessarily current) address space of a process and the file, respectively.

If the size of the mapped file changes after the call to **mmap()**, the effect of references to portions of the mapped region that correspond to added or removed portions of the file is unspecified.

The **mmap()** function is supported for regular files. Support for any other type of file is unspecified.

The *prot* argument determines whether read, write, execute, or some combination of accesses are permitted to the pages being mapped. The protection options are defined in **<sys/mman.h>**:

PROT_READ	Page can be read.
PROT_WRITE	Page can be written.
PROT_EXEC	Page can be executed.
PROT_NONE	Page cannot be accessed.

m

Implementations need not enforce all combinations of access permissions. However, writes shall only be permitted when **PROT_WRITE** has been set.

The *flags* argument provides other information about the handling of the mapped pages. The options are defined in **<sys/mman.h>**:

MAP_SHARED	Share changes.
MAP_ADDR32	Share changes between 32-bit and 64-bit processes.
MAP_PRIVATE	Changes are private.
MAP_FIXED	Interpret *addr* exactly.

The **MAP_PRIVATE** and **MAP_SHARED** flags control the visibility of write references to the memory region. Exactly one of these flags must be specified. The mapping type is retained across a **fork()**.

If **MAP_SHARED** is set in flags, write references to the memory region by the calling process may change the file and are visible in all **MAP_SHARED** mappings of the same portion of the file by any process of the same executable type. That is, an application compiled as a 32-bit process will be able to share the same mappings with other 32-bit processes, and an application compiled as a 64-bit process will be able to share the same mappings with other 64-bit processes.

If a 64-bit and a 32-bit application want to share the same mapping, both **MAP_ADDR32** and **MAP_SHARED** must be set in *flags* by the 64-bit application. The 32-bit application does not need to set **MAP_ADDR32** in *flags*. When **MAP_SHARED** is set in *flags*, write references to the memory region by the calling process may change the file. Changes are visible in all 32-bit processes which specify **MAP_SHARED** and by all 64-bit processes which specify both **MAP_SHARED** and **MAP_ADDR32** for the same portion of the file.

If **MAP_PRIVATE** is set in flags, write references to the memory region by the calling process do not change the file and are not visible to any process in other mappings of the same portion of the file.

It is unspecified whether write references by processes that have mapped the memory region using **MAP_SHARED** are visible to processes that have mapped the same portion of the file using **MAP_PRIVATE**.

It is also unspecified whether write references to a memory region mapped with **MAP_SHARED** are visible to processes reading the file and whether writes to a file are visible to processes that have mapped the modified portion of that file, except for the effect of **msync()**.

When **MAP_FIXED** is set in the *flags* argument, the implementation is informed that the value of *pa* must be *addr*, exactly. If **MAP_FIXED** is set, **mmap()** may return **MAP_FAILED** and set **errno** to **EINVAL**. If a **MAP_FIXED** request is successful, the mapping established by **mmap()** replaces any previous mappings for the process' pages in the range [*pa*, *pa+len*].

When **MAP_FIXED** is not set, the implementation uses *addr* in an unspecified manner to arrive at *pa*. The *pa* so chosen will be an area of the address space which the implementation deems suitable for a mapping of **len** bytes to the file. All implementations interpret an *addr* value of 0 as granting the implementation complete freedom in selecting *pa*, subject to constraints described below. A non-zero value of *addr* is taken to be a suggestion of a process address near which the mapping should be placed. When the implementation selects a value for *pa*, it never places a mapping at address 0, nor does it replace any extant mapping, nor map into dynamic memory allocation areas.

The *off* argument is constrained to be aligned and sized according to the value returned by **sysconf()** when passed **_SC_PAGESIZE** or **_SC_PAGE_SIZE**. When **MAP_FIXED** is specified, the argument *addr* must also meet these constraints. The implementation performs mapping operations over whole pages. Thus, while the argument *len* need not meet a size or alignment constraint, the implementation will include, in any unmapping operation, any partial page specified by the range [*pa*, *pa+len*].

The implementation always zero-fills any partial page at the end of a memory region. Further, the implementation never writes out any modified portions of the last page of a file that are beyond the end of the mapped portion of the file. If the mapping established by **mmap()** extends into pages beyond the page containing the last byte of the file, an application reference to any of the pages in the mapping that are beyond the last page results in the delivery of a **SIGBUS** or **SIGSEGV** signal. The **mmap()** function adds an extra reference to the file associated with the file descriptor *fildes* which is not removed by a subsequent **close()** on that file descriptor. This reference is removed when there are no more mappings to the file. The *st_atime* field of the mapped file may be marked for update at any time between the **mmap()** call and the corresponding **munmap()** call. The initial read or write reference to a mapped region will cause the file's *st_atime* field to be marked for update if it has not already been marked for update.

The *st_ctime* and *st_mtime* fields of a file that is mapped with **MAP_SHARED** and **PROT_WRITE**, will be marked for update at some point in the interval between a write reference to the mapped region and the next call to **msync()** with **MS_ASYNC** or **MS_SYNC** for that portion of the file by any process. If there is no such call, these fields may be marked for update at any time after a write reference if the underlying file is modified as a result.

There may be implementation-dependent limits on the number of memory regions that can be mapped (per process or per system). If such a limit is imposed, whether the number of memory regions that can be mapped by a process is decreased by the use of **shmat()** is implementation-dependent.

RETURN VALUE

Upon successful completion, **mmap()** returns the address, (*pa*), at which the mapping was placed. Otherwise, it returns **MAP_FAILED** (defined in **<sys/mman.h>**) and sets **errno** to indicate the error.

ERRORS

The **mmap()** function will fail if:

[EBADF] The *fildes* argument is not a valid open file descriptor.

[EACCES] The *fildes* argument is not open for read, regardless of the protection specified, or *fildes* is not open for write and **PROT_WRITE** was specified for a **MAP_SHARED** type mapping.

[ENXIO] Addresses in the range [*off*, *off+len*] are invalid for *fildes*.

[EINVAL] The *addr* argument (if **MAP_FIXED** was specified) or off is not a multiple of the page size as returned by **sysconf()**, or are considered invalid by the implementation.

[EINVAL] The value of *flags* is invalid (neither **MAP_PRIVATE** nor **MAP_SHARED** is set).

[EINVAL] The mapping already exists in 64-bit address space, but the application performing the current **mmap()** request has been compiled as a 32-bit executable.

[EINVAL] The mapping already exists in 32-bit address space, but the application performing the current **mmap()** request has been compiled as a 64-bit executable and did not specify **MAP_ADDR32** in the *flags* argument.

[EMFILE] The number of mapped regions would exceed an implementation-dependent limit (per process or per system).

[ENODEV] The *fildes* argument refers to a file whose type is not supported by **mmap()**.

[ENOMEM] **MAP_FIXED** was specified, and the range [*addr*, *addr*+*len*] exceeds that allowed for the address space of a process; or if **MAP_FIXED** was not specified and there is insufficient room in the address space to effect the mapping.

APPLICATION USAGE

Use of **mmap()** may reduce the amount of memory available to other memory allocation functions.

Use of **MAP_FIXED** may result in unspecified behavior in further use of **brk()**, **sbrk()**, **malloc()**, and **shmat()**. The use of **MAP_FIXED** is discouraged, as it may prevent an implementation from making the most effective use of resources.

The application must ensure correct synchronization when using **mmap()** in conjunction with any other file access method, such as **read()** and **write()**, standard input/output, and **shmat()**.

The **mmap()** function allows access to resources via address space manipulations, instead of **read()**/**write()**. Once a file is mapped, all a process has to do to access it is use the data at the address to which the file was mapped. So, using pseudo-code to illustrate the way in which an existing program might be changed to use **mmap()**, the following:

m

```
fildes = open(...)
lseek(fildes, some_offset)
read(fildes, buf, len)
/* use data in buf */
```

becomes:

```
fildes = open(...)
address = mmap(0, len, PROT_READ, MAP_PRIVATE, fildes, some_offset)
/* use data at address */
```

SEE ALSO

exec(2), fcntl(2), fork(2), lockf(2), msync(2), munmap(2), mprotect(2), shmop(2), sysconf(2).

HP-UX EXTENSIONS

NAME

mmap - map pages of memory

SYNOPSIS

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

caddr_t mmap(
     caddr_t addr,
     size_t len,
     int prot,
     int flags,
     int fildes,
     off_t off);
```

DESCRIPTION

MAP_FILE Create a mapped file region.

MAP_ANONYMOUS Create an unnamed memory region.

MAP_VARIABLE Place region at implementation-computed address.

MAP_NORESERVE Lazily evaluate swap space reservation.

The **MAP_FILE** and **MAP_ANONYMOUS** flags control whether the region to be mapped is a mapped file region or an anonymous shared memory region. Exactly one of these flags must be selected.

If **MAP_FILE** is set in *flags*:

- A new mapped file region is created, mapping the file associated with *fildes*.
- *off* specifies the file byte offset at which the mapping starts. This offset must be a multiple of the page size returned by **sysconf(_SC_PAGE_SIZE)**.
- If the end of the mapped file region is beyond the end of the file, any reference to an address in the mapped file region corresponding to an offset beyond the end of the file results in the delivery of a **SIGBUS** signal to the process, unless the address lies in the last partial page corresponding to the range beyond the end of the file. The last partial page mapping the range beyond the end of the file is always initialized to zeros, and any modified portions of the last page of a file which are beyond its end are not written back to the file.

If **MAP_ANONYMOUS** is set in *flags*:

- A new memory region is created and initialized to all zeros. This memory region can be shared only with descendants of the current process.
- If the *fildes* argument is not −1, an EINVAL error is generated.
- The value of *off* is meaningless because there is no underlying file object for the memory region.

The **MAP_VARIABLE** and **MAP_FIXED** flags control the placement of the region as described below. Exactly one of these flags must be selected.

If **MAP_VARIABLE** is set in *flags*:

- If the requested address is NULL, or if it is not possible for the system to place the region at the requested address, the region is placed at an address selected by the system. If the requested address is not a multiple of the page size returned by **sysconf(_SC_PAGE_SIZE)**, the system treats the address as if it were rounded up to the next larger page size multiple.

If **MAP_FIXED** is set in *flags*:

- *addr* must be a multiple of the page size returned by **sysconf(_SC_PAGE_SIZE)**.

If **MAP_NORESERVE** is set in *flags*:

- no swap space is initially be reserved for the private mapping. Without this flag, the creation of a MAP_PRIVATE region reserves swap space equal to the size of the mapping. When a page in the mapping is first modified (written into), a private page is created and the swap space which had been reserved is used to hold the private copy of the data in the event of a page-out. An initial write into a page of a MAP_NORESERVE mapping produces results which depend on the current availability of system swap space since the swap space reservation occurs at the time of the first write and only for the affected page. If the swap space reservation can be made for the page, the write succeeds and is processed as described above; if not, the write fails and a **SIGBUS** signal is posted to the writing process for the effective virtual address. **madvise(...,MADV_DONTNEED)** on a MAP_NORESERVE object will release swap space reservations for relevant pages.

The *prot* argument can be **PROT_NONE**, or any combination of **PROT_READ**, **PROT_WRITE**, and **PROT_EXEC** OR'ed together. If **PROT_NONE** is not specified, the system may grant other access permissions to the region in addition to those explicitly requested, except that write access will not be granted unless **PROT_WRITE** is specified.

mmap() cannot create a mapped file region unless the file descriptor used to map the file is open for reading. For a mapped file region that is mapped with **MAP_SHARED**, **mmap()** grants write access permission only if the file descriptor is open for writing. If a region was mapped with either **MAP_PRIVATE** or **MAP_ANONYMOUS**, **mmap()** grants all requested access permissions.

After the successful completion of **mmap()**, *fildes* can be closed without effect on the mapped region or on the contents of the mapped file. Each mapped region creates a file reference, similar to an open file descriptor, that prevents the file data from being deallocated.

Whether modifications made to the file using **write()** are visible to mapped regions, and whether modifications to a mapped region are visible with **read()**, is undefined except for the effect of **msync()**.

If an enforcement-mode file lock is in effect for any range of a file, a call to **mmap()** to map any range of the file with access rights that would violate the lock fails. The **msem_lock()** and **msem_unlock()** semaphore interfaces can be used to coordinate shared access to a region created with the **MAP_SHARED** flag. The advisory locks of the **lockf()** or **fcntl()** interfaces have no effect on memory mapped access, but they can be used to coordinate shared access to a **MAP_SHARED** mapped file region.

m

For a memory mapped file, the *st_atime* and *st_mtime* values returned by **stat()** are updated when a page in the memory mapped region is read from or written to the file system.

After a call to **fork()**, the child process inherits all mapped regions with the same data and the same sharing and protection attributes as in the parent process. Each mapped file and anonymous memory region created with **mmap()** is unmapped upon process exit, and by a successful call to any of the **exec** functions.

MAP_NORESERVE attribute is inherited across a **fork()** call; at the time of the **fork()**, swap space for a mapping is reserved in the child only for dirtied private pages that currently exist in the parent; thereafter the child's mapping reservation policy is as described above.

A **SIGBUS** signal is delivered to a process when a write reference to a mapped file region would cause a file system error condition such as exceeding quota or file system space limits.

A **SIGBUS** signal is delivered to a process upon a write reference to a region without **PROT_WRITE** protection, or any reference to a region with **PROT_NONE** protection.

A call to **mmap()** with **PROT_EXECUTE** specified, but without **PROT_WRITE** specified for a **MAP_SHARED|MAP_FILE** mapping is treated by the system as the execution of the underlying file. This implies that such a call fails if the file is currently open for writing or mapped with **MAP_SHARED|PROT_WRITE** options by any process, and that if the call succeeds, the file cannot be opened for writing or subsequently mapped with **MAP_SHARED|PROT_WRITE** options as long as such mappings are present. A file's status as an active executable file is determined only at the time of an **exec()**, **exit()**, or **mmap()** operation. **mprotect()** operations on a **MAP_FILE|MAP_SHARED** mapping have no effect on the underlying file's status as an active executable file.

ERRORS

[EACCES] The file referred to by *fildes* is not open for read access, or the file is not open for write access and **PROT_WRITE** was set for a **MAP_SHARED** mapping operation, or **PROT_EXECUTE** was set for a **MAP_SHARED** mapping operation and the underlying file does not have execute permission.

[EOVERFLOW] The file is a regular file and the value of *off+len* exceeds the offset maximum established in the open file description associated with *fildes*.

[ETXTBSY] **MAP_SHARED** and **MAP_FILE** are set, and **PROT_EXECUTE** is set and **PROT_WRITE** is not set, and the file being mapped is currently open for writing.

[EINVAL] The value of *off+len* exceeds the maximum supported offset for mapped files.

DEPENDENCIES

Series 700/800

Because the PA-RISC memory architecture utilizes a globally shared virtual address space between processes and discourages multiple virtual address translations to the same physical address, all concurrently existing **MAP_SHARED** mappings of a file range must share the same virtual address offsets and hardware translations. PA-RISC-based HP-UX systems allocate virtual address ranges for shared memory and shared mapped files in the range 0x80000000 through 0xefffffff for those applications compiled as 32-bit executables or for those 64-bit applications which specify **MAP_SHARED** and **MAP_ADDR32** in the *flags* argument of the **mmap()** function. For applications compiled as 64-bit executables which specify **MAP_SHARED** and do not specify **MAP_ADDR32**, the shared mapped files are in the range 0x00000011 00000000 through 0x000003ff ffffffff and 0xc0000000 00000000 through 0xc00003ff ffffffff. These address ranges are used globally for *all* memory objects shared between processes.

This implies the following:

- Any single range of a file cannot be mapped multiply into different virtual address ranges.
- After the initial **MAP_SHARED mmap()** of a file range, all subsequent **MAP_SHARED** calls to **mmap()** to map the same range of a file must either specify **MAP_VARIABLE** in *flags* and inherit the virtual address range the system has chosen for this range, or specify **MAP_FIXED** with an *addr* that corresponds exactly to the address chosen by the system for the initial mapping. Only after all mappings for a file range have been destroyed can that range be mapped to a different virtual address.
- In most cases, two separate calls to **mmap()** cannot map overlapping ranges in a file. The virtual address range reserved for a file range is determined at the time of the initial mapping of the file range into a process address space. The system allocates only the virtual address range necessary to represent the initial mapping. As long as the initial mapping exists, subsequent attempts to map a different file range that includes any portion of the initial range may fail with an ENOMEM error if an extended contiguous address range that preserves the mappings of the initial range cannot be allocated.
- Separate calls to **mmap()** to map contiguous ranges of a file do not necessarily return contiguous virtual address ranges. The system may allocate virtual addresses for each call to **mmap()** on a first available basis.
- The use of **MAP_FIXED** is strongly discouraged because it is not portable. Using **MAP_FIXED** is generally unsuccessful on this implementation, and when it is successful, it may prevent the system from optimally allocating virtual address space.

MAP_FIXED is discouraged, but there are some applications which by design must fix pointer offsets into file data. The application must map the file at a specific address in order for the file offsets embedded in the file to make sense.

Processes cannot control the usage of global virtual address space, but they can control what happens within their private data area. The Series 700/800 allows a single process to *exclusively* map a file **MAP_SHARED** into its private data space. When a process specifies **MAP_SHARED** and **MAP_FIXED** with a private data address (i.e. second quadrant for 32-bit executable, third quadrant for 64-bit executable), the kernel interprets this as an exclusive mapping of the file. The request will only succeed if no other processes in the system currently have that file mapped through **MAP_SHARED**. If the file is already mapped the caller receives an EBUSY error. If the call is successful, the calling process is the only process allowed to map that file using **MAP_SHARED** until it unmaps the file using **munmap()**. Because it is exclusive, the **mmap()** is not inherited across **fork()**. When a file is exclusively mapped only **MAP_PRIVATE** mappings are allowed by other processes.

The following combinations of protection modes are supported:

```
PROT_NONE
PROT_READ
PROT_READ|PROT_EXECUTE
PROT_READ|PROT_WRITE
PROT_READ|PROT_WRITE|PROT_EXECUTE
```

m

If a **MAP_PRIVATE** mapping is created of a file for which a **MAP_SHARED** mapping exists, a separate copy of a page for the **MAP_PRIVATE** mapping is created at the time of the first access to the page through the private mapping.

AUTHOR

mmap() was developed by HP, AT&T, and OSF.

SEE ALSO

fcntl(2), fork(2), truncate(2), lockf(2), madvise(2), creat64(2), mprotect(2), msem_init(2), msem_lock(2), msem_unlock(2), msync(2), munmap(2), sysconf(2), mman(5), stat(5).

STANDARDS CONFORMANCE

mmap(): AES, SVID3

NAME

modload - load kernel modules on demand

SYNOPSIS

```
#include <sys/mod.h>
int modload(char *pathname);
```

DESCRIPTION

modload allows processes with appropriate privilege to demand-load a kernel module into the running kernel. The module must be of a supported type and must have been registered via *config*(1M) or *kmmodreg*(1M) before it can be loaded.

The module to be loaded is specified by *pathname*. *pathname* may be either a module name or an absolute pathname. If *pathname* is a module name, a list of directories specified by modpath is searched for a match. If *pathname* is absolute, only *pathname* is used to access the object file. The file must be an ELF64 relocatable object file.

NOTES

modload is currently implemented as a macro.

RETURN VALUE

On successful completion, **modload** returns a module identifier that can be passed to **moduload** or **modstat**. On failure it returns -1 and sets **errno** to identify the error.

ERRORS

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

[EACCES]	A component of *pathname* denies search permission.
[ENOENT]	The file named by *pathname* does not exist.
[ENOREG]	Module being loaded is not currently registered.
[EINVAL]	The file named by *pathname* is not appropriately pre-configured or has invalid dependency on other modules.
[EPERM]	The calling process does not have appropriate privilege.
[ERELOC]	A relocation error occurred during the attempt to load the module, or the module references symbols not defined in the running kernel, or the module references symbols in another loadable module but it did not declare its dependence on this module in its *master*(4) file.
[ENAMETOOLONG]	*pathname* is more than **MAXPATHLEN** characters long.
[EBADVER]	The module wrapper has an incorrect version number.
[ENOSYS]	The Dynamically Loadable Kernel Module feature is not initialized.

SEE ALSO

config(1M), kmadmin(1M), kmmodreg(1M), modstat(2), moduload(2), master(4).

NAME

modpath - change global search path for dynamically loadable kernel modules

SYNOPSIS

#include <sys/mod.h>

int modpath(const char ****pathname*);**

DESCRIPTION

modpath allows users with appropriate privilege to modify the global search path used to locate object files for dynamically loadable kernel modules. The search path modifications take effect immediately and affect all subsequent loads for all users on the system.

pathname may be either a colon-separated list of absolute pathnames or NULL. If the former, these path names represent directories which should be searched for all autoloads of loadable kernel modules and for demand loads (see *modload*(2)) where the module is given by a simple file name. This list of directories will be prepended to the existing list of directories and so will be searched before any directories given in previous calls to **modpath** and before the default location which is always searched last. The directories do not have to exist on the system at the time **modpath** is called, or when a load actually takes place. If *pathname* is equal to NULL, the global search path is set back to its initial default value, **/stand/dlkm/mod.d**.

NOTES

modpath is currently implemented as a macro.

RETURN VALUE

On success, **modpath** returns 0, otherwise it returns -1 and sets *errno* to indicate the error.

ERRORS

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

[EINVAL]	The list of directories specified by *pathname* is malformed.
[ENOSYS]	The Dynamically Loadable Kernel Module feature is not initialized.
[ENAMETOOLONG]	*pathname* is more than **MAXPATHLEN** characters long.
[EPERM]	The calling process does not have appropriate privilege.

SEE ALSO

kmadmin(1M), modload(2).

m

NAME

modstat - get information for a dynamically loaded kernel module

SYNOPSIS

```
#include <sys/mod.h>
int modstat(int module_id, struct modstatus *stbuf,
            int get_next_module);
```

DESCRIPTION

The **modstat** function allows processes with appropriate privilege to get information for dynamically loaded kernel modules. It fills in the elements of the **modstatus** structure, specified by *stbuf*, with the information available for the given module identifier *module_id*. If the value of *get_next_module* is TRUE, **modstat** returns the information for the next module whose identifier is greater than or equal to *module_id*. Any *module_id* associated with a registered module (see *kmadmin*(1M)) may be queried by **modstat**.

The **struct modstatus** and **struct modspecific_stat** definitions are:

```
struct modstatus {
     int32_t  ms_id;                  /* numeric id of module */
     uint64_t ms_base;                /* base address of module */
     uint32_t ms_size;                /* amount of memory of module
                                      when loaded */
     uint64_t ms_bss;                 /* base address of BSS */
     uint32_t ms_bss_size;            /* memory size of BSS */
     int32_t  ms_rev;                 /* version number */
     char     ms_path[MAXPATHLEN];    /* loaded module path */
     time_t   ms_unload_delay;        /* unload delay */
     int32_t  ms_holdcnt;             /* hold count */
     int32_t  ms_depcnt;              /* dependent count */
     struct modspecific_stat          /* module type specific info */
              ms_msinfo[MODMAXLINK];
};
struct modspecific_stat {
    char     mss_linkinfo[MODMAXLINKINFOLEN]; /* informational */
    int32_t  mss_type;                        /* type of module */
    int32_t  mss_p0[2];                       /* type specific info */
    int32_t  mss_p1[2];                       /* type specific info */
}
```

m

NOTES

modstat is currently implemented as a macro.

RETURN VALUE

On success, **modstat** returns 0, otherwise it returns -1 and sets **errno** to indicate the error.

ERRORS

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

[EINVAL] *module_id* does not match any loaded or registered module when *get_next_module* is FALSE or *module_id* is greater than the identifier for any loaded module when *get_next_module* is TRUE.

[ENOSYS] The Dynamically Loadable Kernel Module feature is not initialized.

[EPERM] The calling process does not have appropriate privilege.

SEE ALSO

kmadmin(1M), modload(2).

NAME

moduload - unload a kernel module on demand

SYNOPSIS

#include <sys/mod.h>

int moduload(long *module_id***);**

DESCRIPTION

moduload allows users with appropriate privilege to demand unload one or all unloadable modules from the running kernel. A module is considered *unloadable* if it is not currently in use, if no module depending on it is currently loaded, and if the module is not being loaded or unloaded from the kernel. If *module_id* is set to 0 (zero), **moduload** attempts to unload all unloadable modules, otherwise **moduload** attempts to unload the module specified by *module_id*.

NOTES

moduload is currently implemented as a macro.

RETURN VALUE

On success, **moduload** returns 0, otherwise it returns -1 and sets **errno** to indicate the error.

ERRORS

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

- **[EINVAL]** *module_id* does not correspond to any valid currently loaded kernel module.
- **[EPERM]** The calling process does not have appropriate privilege.
- **[EBUSY]** There are outstanding references to the module, or modules that depend on this module are currently loaded, or profiling is enabled, or the module is in the process of being loaded or unloaded from the kernel.
- **[ENOSYS]** The Dynamically Loadable Kernel Module feature is not initialized.

m

SEE ALSO

kmadmin(1M), modload(2).

NAME

mount() - mount a file system

SYNOPSIS

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

int mount(const char *fs, const char *path, int mflag);

int mount(const char *fs,
          const char *path,
                int   mflag,
          const char *fstype,
          const char *dataptr,
                int   datalen);
```

DESCRIPTION

The **mount()** system call requests that a file system identified by *fs* be mounted on the file identified by *path*.

mflag contains a bit-mask of flags (described below). Note that the **MS_DATA** flag must be set for the six-argument version of the call.

fstype is the file system type name. It is the same name that *sysfs*(2) uses.

The last two arguments together describe a block of file-system-specific data at address *dataptr* of length *datalen*. This is interpreted by file-system-specific code within the operating system and its format depends upon the file system type. A particular file system type may not require this data, in which case *dataptr* and *datalen* should both be zero. The mounting of some file system types may be restricted to a user with appropriate privileges.

mount() can be invoked only by a user who has appropriate privileges.

Upon successful completion, references to the file *path* will refer to the root directory of the mounted file system.

mflag contains a bit-mask of flag values, which includes the following defined in **<sys/mount.h>**:

MS_DATA This is ordinarily required. It indicates the presence of the *fstype*, *dataptr*, and *datalen* arguments.

(For backward compatibility, if this flag is not set, the *fstype* is assumed to be that of the root file system, and *dataptr* and *datalen* are assumed to be zero.)

MS_RDONLY This is used to control write permission on the mounted file system. If not set, writing is permitted according to individual file accessibility.

MS_NOSUID This flag disables set-user-ID and set-group-ID behavior on this file system.

MS_QUOTA This causes quotas to be enabled if the file system supports quotas.

If *fstype* is specified as:

MNTTYPE_HFS

Mount a local HFS file system. *dataptr* points to a structure of the following format, if the options described below need to be specified for the mount:

```
struct ufs_args {
        char *fspec;
        int   flags;
};
```

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, *fs*, of the system call.

flags points to a bit map that sets options. The following values of the bits are defined in **<sys/mount.h>**:

MS_DELAY Writes to disks are to be delayed until the buffer needs to be reused. This is the default on Series 800 systems, as it was prior to release 10.0.

m

MS_BEHIND Writes to disks are to be done asynchronously, where possible, without waiting for completion. This is the default on Series 700 systems, as it was prior to release 10.0.

MS_BEHIND and **MS_DELAY** are mutually exclusive.

MS_NO_FSASYNC Rigorous posting of file system metadata is to be used. This is the default.

MS_FSASYNC Relaxed posting of file system metadata is to be used. This may lead to better performance for certain applications; but there is increased potential for data loss in case of a crash.

MS_FSASYNC and **MS_NO_FSASYNC** are mutually exclusive.

RETURN VALUE

mount() returns the following values:

0 Successful completion.
-1 Failure. **errno** is set to indicate the error.

ERRORS

If **mount()** fails, **errno** is set to one of the following values.

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

[EBUSY] *path* is currently mounted on, is someone's current working directory, or is otherwise busy.

[EBUSY] The file system associated with *fs* is currently mounted.

[EBUSY] The system cannot allocate the necessary resources for this mount.

[EFAULT] *fs*, *path* or *dataptr* points outside the allocated address space of the process. The reliable detection of this error is implementation dependent.

[EINVAL] An argument to the system call is invalid, or a sanity check failed.

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

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

[ENODEV] *fstype* is a file system that is not been configured into the kernel.

[ENOENT] A named file does not exist.

[ENOENT] *fs* or *path* is null.

[ENOTBLK] *fs* is not a block special device and the file system type requires it to be.

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

[ENOTDIR] *path* is not a directory.

[ENXIO] The device associated with *fs* does not exist and the file system type requires it to be.

[EPERM] The process does not have the appropriate privilege and the file system type requires it.

[EROFS] The requested file system is write protected and *mflag* requests write permission.

WARNINGS

If **mount()** is called from the program level (i.e., not called with the **mount** command (see *mount*(1M)), the table of mounted devices contained in **/etc/mnttab** is not updated. The updating of **/etc/mnttab** is performed by the **mount** and **syncer** commands (see *mount*(1M) and *syncer*(1M)).

SEE ALSO

mount(1M), syncer(1M), sysfs(2), umount(2).

NAME

mpctl - multiprocessor control

SYNOPSIS

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

int mpctl(
     mpc_request_t request,
     spu_t spu,
     pid_t pid
);

int mpctl(
     mpc_request_t request,
     spu_t spu,
     lwpid_t lwpid
);
```

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

mpctl provides a means of determining how many processors are installed in the system and assigning proceses/lightweight processes to run on specific processors.

This call is expected to be used to increase performance in certain applications, but should not be used to ensure correctness of an application. Specifically, cooperating processes/lightweight processes should not rely on processor assignment in lieu of a synchronization mechanism (such as semaphores).

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

MPC_GETNUMSPUS
: This request returns the number of spus (processors) in the system. It will always be greater than or equal to 1. The *spu* and *pid* (or *lwpid*) arguments are ignored.

MPC_GETFIRSTSPU
: This request returns the number of the first processor in the system. The *spu* and *pid* (or *lwpid*) arguments are ignored.

MPC_GETNEXTSPU
: This request returns the number of the next processor in the system after *spu*. Typically, MPC_GETFIRSTSPU is called to determine the first spu. MPC_GETNEXTSPU is then called in a loop (until the call returns -1) to determine the numbers of the remaining spus. The *pid* (or *lwpid*) argument is ignored.

MPC_GETCURRENTSPU
: This request returns the number of the processor the process is currently running on (NOT the processor assignment of the caller). The number of the processor the process is currently running on is undefined if the process is multithreaded. The *spu* and *pid* (or *lwpid*) arguments are ignored. This information may be out-of-date arbitrarily soon after the call completes.

MPC_SETPROCESS
: This call is **advisory**. This request asynchronously assigns process *pid* to processor *spu*. Since the new *spu* assignment is returned, the *spu* MPC_SPUNOCHANGE may be passed to read the current assignment. The *pid* MPC_SELFPID may be used to refer to the calling process. The *spu* MPC_SPUFLOAT may be used to break any specific-processor assignment. This allows the process to float to any processor.

Note: This call is *advisory*. If the scheduling policy for a process conflicts with this processor assignment, the scheduling policy takes precedence. For example, when a processor is ready choose another process/lightweight process to execute, and the highest priority **SCHED_FIFO** process is bound to a different processor, that process will execute on the selecting processor rather than waiting for the specified

m

processor to which it was bound.

If the process specified by *pid* is a multithreaded process, all threads (lightweight processes) in the target process will have their processor assignment changed to what is specified.

MPC_SETPROCESS_FORCE
This call is identical to MPC_SETPROCESS except that the process to processor binding will override the scheduling policy. This call is synchronous. For example, when a processor is ready choose another process/lightweight process) to execute, and the highest priority **SCHED_FIFO** process is bound to a different processor, that process will not be selected to execute on the selecting processor, but instead wait for the specified processor to which it was bound. The selecting processor will then choose a lower priority process to execute on the processor.

Note: This option will not guarantee compliance with POSIX real-time scheduling algorithms.

If the process specified by *pid* is a multithreaded process, all threads (lightweight processes) in the target process will have their processor assignment changed to what is specified.

MPC_SETLWP This call is *advisory*. This request asynchronously assigns thread (lightweight process - LWP) *lwpid* to processor *spu*. This option is only available to change the assignment for threads (LWPs) in the current process. Since the new *spu* assignment is returned, the *spu* MPC_SPUNOCHANGE may be passed to read the current assignment. The *lwpid* MPC_SELFLWPID may be used to refer to the calling thread (LWP). The *spu* MPC_SPUFLOAT may be used to break any specific-processor assignment. This allows the thread (LWP) to float to any processor.

Note: This call is *advisory*. If the scheduling policy for a thread (LWP) conflicts with this processor assignment, the scheduling policy takes precedence. For example, when a processor is ready choose another thread (LWP) to execute, and the highest priority **SCHED_FIFO** thread (LWP) is bound to a different processor, that thread (LWP) will execute on the selecting processor rather than waiting for the specified processor to which it was bound.

MPC_SETLWP_FORCE
This call is identical to MPC_SETLWP except that the thread (LWP) to processor binding will override the scheduling policy. This call is synchronous. For example, when a processor is ready choose another thread (LWP) to execute, and the highest priority **SCHED_FIFO** thread (LWP) is bound to a different processor, that thread (LWP) will not be selected to execute on the selecting processor, but instead wait for the specified processor to which it was bound. The selecting processor will then choose a lower priority thread (LWP) to execute on the processor.

Note: This option will not guarantee compliance with POSIX real-time scheduling algorithms.

MPC_GETPROCESS_BINDINGTYPE
This request returns MPC_ADVISORY or MPC_MANDATORY to indicate the current binding type of the process specified by *pid*. The *spu* argument is ignored.

MPC_GETLWP_BINDINGTYPE
This request returns MPC_ADVISORY or MPC_MANDATORY to indicate the current binding type of the thread (LWP) specified by *lwpid*. The *spu* argument is ignored.

To change the processor assignment of another process, the caller must be a member of a group having PRIV_MPCTL access (or be the super-user).

If the *request* argument specifies MPC_SETPROCESS or MPC_SETPROCESS_FORCE and the process specified by *pid* is a multi-threaded process, the processor binding specified shall be applied for all lightweight processes contained within *pid*.

Each process shall contain a specified processor binding. Each lightweight process shall contain a processor binding. The processor binding for a lightweight process does not have to match the processor binding for the process.

m

When a process creates another process (via **fork()** or **vfork()**) the child process will inherit the parent processes processor binding. The initial lightweight process in the child process shall inherit its processor binding from the child process. Lightweight processes other than the initial lightweight process shall inherit their processor binding from the creating lightweight process.

ERRORS

In general, **mpctl** fails if one or more of the following is true:

[EINVAL] *request* is an illegal number.

[EINVAL] *request* is MPC_GETNEXTSPU and *spu* identifies the last processor.

[ESRCH] *pid* or *lwpid* identifies a process or LWP that does not exist.

[EPERM] *request* is MPC_SETPROCESS or MPC_SETPROCESS_FORCE, *spu* is not MPC_SPUNOCHANGE, *pid* identifies another process, and the caller is not the superuser or a member of a group having PRIV_MPCTL access.

SEE ALSO

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

m

NAME

mprotect - set protection of memory mapping

SYNOPSIS

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

int mprotect(void *addr, size_t len, int prot);
```

DESCRIPTION

The **mprotect()** function changes the access protections on the mappings specified by the range [*addr*, *addr+len*], rounding *len* up to the next multiple of the page size as returned by **sysconf()**, to be that specified by *prot*. Legitimate values for *prot* are the same as those permitted for **mmap()** and are defined in **<sys/mman.h>**:

PROT_READ Page can be read.

PROT_WRITE Page can be written.

PROT_EXEC Page can be executed.

PROT_NONE Page cannot be accessed.

When **mprotect()** fails for reasons other than EINVAL, the protections on some of the pages in the range [*addr*, *addr+len*] may have been changed.

RETURN VALUE

Upon successful completion, **mprotect()** returns 0. Otherwise, it returns -1 and sets **errno** to indicate the error.

ERRORS

The **mprotect()** function will fail if:

[EACCES] The *prot* argument specifies a protection that violates the access permission the process has to the underlying memory object.

[EINVAL] The *addr* argument is not a multiple of the page size as returned by **sysconf()**.

[ENOMEM] Addresses in the range [*addr*, *addr+len*] are invalid for the address space of a process, or specify one or more pages which are not mapped.

The **mprotect()** function may fail if:

[EAGAIN] The *prot* argument specifies **PROT_WRITE** over a **MAP_PRIVATE** mapping and there are insufficient memory resources to reserve for locking the private page.

SEE ALSO

mmap(2), sysconf(2), <sys/mman.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

HP-UX EXTENSIONS

SYNOPSIS

```
int mprotect(
    caddr_t addr,
    size_t len,
    int prot );
```

DESCRIPTION

If the address range does not correspond to one created by a successful call to **mmap()**, **mprotect()** returns an error. *prot* determines whether read, write, execute, or some combination of accesses are permitted to the data being mapped.

If the address range being modified corresponds to a mapped file that was mapped with **MAP_SHARED**, **mprotect()** grants write access permission only if the file descriptor used to map the file was opened for writing. If the address range corresponds to a mapped file that was mapped with the **MAP_PRIVATE** or the **MAP_ANONYMOUS** flag, **mprotect()** grants all requested access permissions.

For example, suppose an error occurs on some page at an *addr2*; **mprotect()** may have modified the protections of all whole pages in the range [*addr*,*addr2*].

ERRORS

[EINVAL] *prot* is invalid, or *addr* is not a multiple of the page size as returned by **sysconf(_SC_PAGE_SIZE)**.

[EFAULT] The range specified by [*addr*, *addr+len*] (from, and including, *addr* to, but not including, *addr+len*) is invalid for a process' address space, or the range specifies one or more unmapped pages.

m

AUTHOR

mprotect() was developed by HP, AT&T, and OSF.

SEE ALSO

mmap(2), sysconf(2).

STANDARDS CONFORMANCE

mprotect(): AES, SVID3

NAME

mq_close - close a message queue descriptor

SYNOPSIS

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

int mq_close(mqd_t mqdes);
```

DESCRIPTION

The **mq_close()** system call removes the association between the message queue descriptor, *mqdes*, and a message queue. Use of this message queue descriptor by the process, after a successful return from this **mq_close()**, and until this descriptor is returned by a subsequent **mq_open()**, will result in the failure of message queue system calls, with **errno** set to **EBADF**.

If the process has a registered notification request with the message queue associated with this *mqdes*, the registration is canceled and the queue becomes available for another process to register a notification request.

If the message queue has been unlinked and *mqdes* is the only existing open descriptor for the queue, the queue is destroyed.

RETURN VALUE

mq_close() returns the following values:

- **0** Successful completion.
- **-1** Failure. **errno** is set to indicate the error.

ERRORS

If **mq_close()** fails, **errno** is set to one of the following values:

- [EBADF] *mqdes* is not a valid message queue descriptor.
- [ENOSYS] **mq_close()** is not supported by the implementation.

m

SEE ALSO

mq_open(2), mq_unlink(2), mq_notify(2).

STANDARDS CONFORMANCE

mq_close(): POSIX 1003.1b

NAME

mq_getattr - get status information and attributes associated with a message queue

SYNOPSIS

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

int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat);
```

DESCRIPTION

The **mq_getattr()** system call collects status information and attributes associated with the message queue specified by *mqdes* which is copied into the **mq_attr** structure referenced by *mqstat*.

Upon a successful return, the *mq_msgsize* and *mq_maxmsg* fields within the *mq_attr* structure contain the maximum size of a message for this queue and the maximum number of messages that can be queued at any time. The *mq_curmsgs* field contains the number of messages currently on the queue. In addition, the *mq_flags* field contains the message queue blocking status associated with this *mqdes*.

RETURN VALUE

mq_getattr() returns the following values:

0 Successful completion.

-1 Failure. **errno** is set to indicate the error.

ERRORS

If **mq_getattr()** fails, **errno** is set to one of the following values:

[EBADF] *mqdes* is not a valid message queue descriptor.

[EINVAL] *mqstat* does not point to a valid *mq_attr* structure.

[ENOSYS] **mq_getattr()** is not supported by the implementation.

SEE ALSO

mq_getattr(2), mq_open(2).

STANDARDS CONFORMANCE

mq_getattr(): POSIX 1003.1b

NAME

mq_notify - register/cancel a notification request with a message queue

SYNOPSIS

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

int mq_notify(mqd_t mqdes, const struct sigevent *notification);
```

DESCRIPTION

If the argument *notification* is not **NULL**, the **mq_notify()** system call registers the calling process to be notified of message arrival at an empty message queue associated with the message queue descriptor *mqdes*. The notification specified by the *notification* argument will be sent to the process when the message queue transitions from the empty state to the non-empty state. At any time, only one process can be registered for notification with a message queue. If the calling process, or any other process has already registered for notification with the specified message queue, subsequent attempts to register with that queue will fail.

If *notification* is **NULL** and the process is currently registered for notification with the specified message queue, the existing registration is canceled.

When the notification is sent to the registered process, its registration is removed. The message queue is then available for registration.

If there is some process blocked in **mq_receive()** waiting to receive a message from a message queue, the arrival of a message on the queue will satisfy the **mq_receive()**, even if the queue has a registered notification request. The resulting behavior is as if the message queue remains empty, and no notification is sent.

RETURN VALUE

mq_notify() returns the following values:

0 Successful completion.

-1 Failure. **errno** is set to indicate the error.

m

ERRORS

If **mq_notify()** fails, **errno** is set to one of the following values:

[EAGAIN] The system lacks sufficient signal queuing resources to honor the request.

[EBADF] The *mqdes* argument is not a valid message queue descriptor.

[EBUSY] A process is already registered for notification with the message queue.

[EINVAL] An attempt was made to cancel a non-existent notification request, or *notification* points to an invalid address.

[ENOSYS] **mq_notify()** is not supported by the implementation.

SEE ALSO

mq_open(2), mq_send(2).

STANDARDS CONFORMANCE

mq_notify() : POSIX 1003.1b

NAME

mq_open - create/open a message queue

SYNOPSIS

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

mqd_t mq_open(const char *name,
              int oflag, ...
              /*
               * [mode_t mode, struct mq_attr]
               */
);
```

Remarks

The ANSI C "**, ...**" construct specifies a variable length argument list whose optional members are given in the associated comment (**/* */**).

DESCRIPTION

The **mq_open()** system call establishes a connection between a process and a message queue. It returns a new message queue descriptor which is used by other message queue system calls to refer to that queue.

The *name* argument points to the message queue name, and must conform to the rules listed in *Message Queue Naming Convention*.

The *oflag* argument is the bitwise inclusive OR of the flags listed in *Read-Write Flags*, and *General Flags* below.

The new message queue descriptor returned, remains open across the **fork()** system call and is inherited by the child process.

Read-Write Flags

The value of *oflag* must be composed by taking the inclusive OR of exactly one of the following flags:

O_RDONLY Open for receiving only.

O_WRONLY Open for sending only.

O_RDWR Open for sending and receiving.

General Flags

Any combination of the following flags may also be used in setting the value of *oflag*.

O_CREAT This flag must be used to create a message queue, and it uses two additional arguments: *mode* which is of type **mode_t**, and *attr* which is a pointer to a **mq_attr** structure. If a message queue with name, *name*, exists, this flag has no effect, except as noted under **O_EXCL**. Otherwise a new message queue is created. The user ID of the queue will be set to the effective user ID of the process, and the group ID of the queue will be set to the effective group ID of the process. The "file permission bits" will be set to the value of *mode*. If *attr* is NULL, the message queue is created with default attributes - **MQ_MAXMSG** and **MQ_MSGSIZE** (defined in **sys/mqueue.h**) If *attr* is non-NULL and the message queue *mq_maxmsg* and *mq_msgsize* attributes are set to the values of the corresponding members in the **mq_attr** structure referred to by *attr*.

O_EXCL If **O_EXCL** and **O_CREAT** are set in *oflag* and the named message queue exists, **mq_open()** will fail. The **O_CREAT** flag is ignored if **O_CREAT** is not set in *oflag*.

O_NONBLOCK This flag is used to specify the blocking status of the message queue descriptor and determines whether a **mq_send()** or a **mq_receive()** will wait for resources or messages respectively, that are not currently available, or fail with **errno** set to [EAGAIN].

Message Queue Naming Convention

A valid message queue name string, must conform to pathname construction rules. In addition it must also meet the following specifications:

a. Begin with a slash character.

m

b. Contain no pathname component consisting of a dot or dot-dot; e.g., **/./tmp** and **/../tmp**.

c. Contain no illegal characters.

d. Contain no pathname components longer that **_POSIX_NAME_MAX**.

e. Entire name should not be longer that **_POSIX_PATH_MAX**.

RETURN VALUE

mq_open() returns the following values:

n Successful completion. *n* is a message queue descriptor for the opened message queue and is greater than or equal to **0**.

-1 Failure. **errno** is set to indicate the error.

ERRORS

If **mq_open()** fails, **errno** is set to one of the following values:

[EACCES] The message queue exists and the permissions specified by *oflag* are denied, or the message queue does not exist and permission to create the queue is denied.

[EEXIST] The **O_CREAT** and **O_EXCL** flags are set in *oflag* and the named message queue exists.

[EINTR] **mq_open()** was interrupted by a signal.

[EINVAL] The argument *name*, does not conform to the *Message Queue Naming Convention*.

O_CREAT has been specified in *oflag*, the value of *attr* is not NULL, and either *mq_maxmsg* or *mq_msgsize* is less than or equal to zero.

[EMFILE] Too many message queue descriptors are currently in use by this process.

[ENAMETOOLONG]
The length of the *name* string exceeds **PATH_MAX** bytes, or the length or a (pathname) component of the *name* string exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.

[ENFILE] Too many message queues are currently open in the system.

[ENOENT] The **O_CREAT** flag is not set in *oflag* and the named message queue does not exist.

[ENOSPC] There are insufficient resources for the creation of the new message queue.

[ENOSYS] **mq_open()** is not supported by the implementation.

SEE ALSO

mq_close(2), mq_unlink(2), mq_send(2), mq_receive(2), mq_setattr(2), mq_getattr(2).

STANDARDS CONFORMANCE

mq_open(): POSIX 1003.1b

m

NAME

mq_receive - receive a message from a message queue

SYNOPSIS

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

ssize_t mq_receive(mqd_t          mqdes,
                   char           *msg_ptr,
                   size_t         msg_len,
                   unsigned int   *msg_prio
);
```

DESCRIPTION

The **mq_receive()** system call receives the oldest of the highest priority message from the message queue specified by *mqdes*. The selected message is removed from the queue and copied to the buffer pointed to by the *msg_ptr* argument. The argument, *msg_len*, specifies the size of the buffer in bytes. The value of *msg_len* should be greater than or equal to the *mq_msgsize* attribute of the message queue, or **mq_receive()** will fail.

If the argument *msg_prio* is not NULL, the priority of the message removed from the queue is stored in the location pointed to by *msg_prio*.

If the specified message queue is empty and the **O_NONBLOCK** flag is not set in the message queue blocking status associated with *mqdes*, **mq_receive()** will block in priority order, until it can receive a message from the queue, or until **mq_receive()** is interrupted by a signal. If the specified message queue is empty and the **O_NONBLOCK** flag is set in the message queue blocking status associated with *mqdes*, **mq_receive()** will not wait for a message to arrive on the queue and will return with an error.

RETURN VALUE

mq_receive() returns the following values:

- *n* Successful completion. *n* is the size of the selected message in bytes and the message is removed from the queue.
- **-1** Failure. **errno** is set to indicate the error and no message is removed from the queue.

ERRORS

If **mq_receive()** fails, **errno** is set to one of the following values:

- [EAGAIN] The **O_NONBLOCK** flag is set in the message queue blocking status associated with *mqdes*, and the message queue is empty.
- [EBADF] *mqdes* is not a valid message queue descriptor open for reading.
- [EINTR] A signal interrupted the call to **mq_receive()**.
- [EINVAL] *msg_ptr* points to an invalid address.
- [EMSGSIZE] The specified message buffer size, *msg_len*, is less than the message size attribute of the message queue.
- [ENOSYS] **mq_receive()** is not supported by the implementation.

SEE ALSO

mq_send(2).

STANDARDS CONFORMANCE

mq_receive(): POSIX 1003.1b

m

NAME

mq_send - send a message to a message queue

SYNOPSIS

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

int mq_send(mqd_t        mqdes,
            const char   *msg_ptr,
            size_t       msg_len,
            unsigned int msg_prio
);
```

DESCRIPTION

The **mq_send()** system call adds a message pointed to by the argument *msg_ptr* to the message queue specified by *mqdes*. The *msg_len* argument specifies the length of the message in bytes. The value of *msg_len* should be less than or equal to the *mq_msgsize* attribute of the message queue, or **mq_send()** will fail.

If the specified message queue is not full, **mq_send()** will insert the message into the queue at the position indicated by the *msg_prio* argument. A message with priority, *msg_prio*, will be inserted behind any other messages with larger or equal priority. The value of *msg_prio* should be less than **MQ_PRIO_MAX**.

If the specified message queue is full and the **O_NONBLOCK** flag is not set in the message queue blocking status associated with *mqdes*, **mq_send()** will block in priority order, until it can send a message on the queue, or until **mq_send()** is interrupted by a signal. If the specified message queue is full and the **O_NONBLOCK** flag is set in the message queue blocking status associated with *mqdes*, the message will not be enqueued, and **mq_send()** will return with an error.

RETURN VALUE

mq_send() returns the following values:

- **0** Successful completion. The message is enqueued.
- **-1** Failure. **errno** is set to indicate the error and the message is not enqueued.

ERRORS

If **mq_send()** fails, **errno** is set to one of the following values:

[EAGAIN] The **O_NONBLOCK** flag is set in the message queue blocking status associated with *mqdes*, and the message queue is full.

[EBADF] *mqdes* is not a valid message queue descriptor open for writing.

[EINTR] A signal interrupted the call to **mq_send()**.

[EINVAL] *msg_ptr* points to an invalid address, or the value of *msg_prio* is outside the valid range.

[EMSGSIZE] The specified message length, *msg_len*, exceeds the message size attribute of the message queue.

[ENOSYS] **mq_send()** is not supported by the implementation.

SEE ALSO

mq_receive(2), mq_setattr(2), mq_getattr(2).

STANDARDS CONFORMANCE

mq_send(): POSIX 1003.1b

NAME

mq_setattr - set the blocking status of a message queue associated with a descriptor

SYNOPSIS

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

int mq_setattr(mqd_t mqdes,
               const struct mq_attr *mqstat,
               struct mq_attr       *omqstat,
);
```

DESCRIPTION

The **mq_setattr()** system call changes the blocking status of a message queue associated with the descriptor, *mqdes*. The blocking status that is modified is per message queue descriptor and another open descriptor for the same message queue can have a different blocking status.

The argument *mqstat*, points to an **mq_attr** structure that specifies the blocking status desired. More specifically, if the **O_NONBLOCK** bit in the *mq_flags* field of the **mq_attr** structure is set, the descriptor is marked as non-blocking. Otherwise it is marked as blocking.

If *omstat* is non-NULL, **mq_setattr()** will store in the **mq_attr** structure referenced by *omqstat*, the previous message queue attributes and the queue blocking status associated with this *mqdes*. The values returned are the same as would be returned by a call to **mq_getattr()**.

RETURN VALUE

mq_setattr() returns the following values:

- **0** Successful completion.
- **-1** Failure. **errno** is set to indicate the error.

ERRORS

If **mq_setattr()** fails, **errno** is set to one of the following values:

- [EBADF] *mqdes* is not a valid message queue descriptor.
- [EINVAL] *mqstat* does not point to a valid **mq_attr** structure, or *omqstat* is non-NULL and does not point to a valid **mq_attr** structure.
- [ENOSYS] **mq_setattr()** is not supported by the implementation.

SEE ALSO

mq_setattr(2), mq_open(2).

STANDARDS CONFORMANCE

mq_setattr(): POSIX 1003.1b

m

NAME

mq_unlink - unlink a message queue

SYNOPSIS

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

int mq_unlink(const char *name);
```

DESCRIPTION

The **mq_unlink()** system call disassociates the queue name, from a message queue specified by the argument, *name*. After a successful call to **mq_unlink()**, attempts to open a message queue with the same name will fail, if the flag **O_CREAT** is not set in *oflags*.

If there are no processes with existing open descriptors for the message queue, the queue is destroyed. If one or more processes have the message queue open, the removal of the queue is postponed until all descriptors for the queue have been closed.

RETURN VALUE

mq_unlink() returns the following values:

0 Successful completion.

-1 Failure, **errno** is set to indicate the error.

ERRORS

If **mq_unlink()** fails, **errno** is set to one of the following values:

[EACCES] Permission to unlink the named message queue is denied.

[EINVAL] The argument *name* is not a valid message queue name.

[ENAMETOOLONG] The length of the *name* string exceeds **PATH_MAX** bytes, or the length of a (pathname) component of the *name* string exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.

[ENOENT] The named message queue does not exist.

[ENOSYS] **mq_unlink()** is not supported by the implementation.

SEE ALSO

mq_open(2), mq_close(2).

STANDARDS CONFORMANCE

mq_unlink(): POSIX 1003.1b

m

NAME

msem_init - initialize a semaphore in a mapped file or anonymous memory region

SYNOPSIS

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

msemaphore *msem_init(msemaphore *sem, int initial_value);
```

DESCRIPTION

msem_init() allocates a new binary semaphore and initializes the state of the new semaphore.

sem points to an **msemaphore** structure in which the state of the semaphore is to be stored.

If *initial_value* is **MSEM_LOCKED**, the new semaphore is initialized in the locked state. If *initial_value* is **MSEM_UNLOCKED**, the new semaphore is initialized in the unlocked state.

The **msemaphore** structure must be located within a mapped file or anonymous memory region created by a successful call to **mmap()** and have both read and write access.

If a semaphore is created in a mapped file region, any reference by a process that has mapped the same file, using a **(struct msemaphore *)** pointer that resolves to the same file offset is interpreted as a reference to the same semaphore. If a semaphore is created in an anonymous memory region, any reference by a process sharing the same region by use of a **(struct msemaphore *)** pointer that resolves to the same offset from the start of the region is interpreted as a reference to the same semaphore.

Any previous semaphore state stored in the **msemaphore** structure will be ignored and overwritten.

Implementation Notes

In order to ensure that an **msemaphore** structure is entirely contained in a single memory page, *sem* must be at an address that is an exact multiple of **sizeof(struct msemaphore)**. The size of the **msemaphore** structure is guaranteed to prevent semaphores that cross page boundaries given the above restriction.

For a memory mapped file region, the system deallocates memory that corresponds to a range of the file that has been truncated with **ftruncate()** or **truncate()**. If a semaphore is located in memory so deallocated, the effect is equivalent to an **msem_remove()** on the semaphore.

RETURN VALUE

msem_init() returns the address of the initialized **msemaphore** structure; otherwise, it returns NULL and sets **errno** to indicate the error. NOTE: This error return value may change to –1 in a future HP-UX release. For portability, applications should check for a zero or negative value for error returns.

ERRORS

msem_init() fails if any of the following conditions are encountered:

[EINVAL] *sem* points to an **msemaphore** structure that is not located in a mapped region created by **mmap()** and with read and write access, or *initial_value* is not valid.

[ENOMEM] A new semaphore could not be created.

[EFAULT] *sem* is an invalid pointer.

AUTHOR

msem_init() was developed by HP and OSF.

SEE ALSO

mmap(2), msem_lock(2), msem_remove(2), msem_unlock(2), mman(5).

STANDARDS CONFORMANCE

msem_init(): AES

m

NAME

msem_lock - lock a semaphore

SYNOPSIS

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

int msem_lock(msemaphore *sem, int condition);
```

DESCRIPTION

msem_lock() attempts to lock a binary semaphore.

sem points to an **msemaphore** structure which specifies the semaphore to be locked.

If the semaphore is not currently locked, it becomes locked and the function returns successfully.

If the semaphore is currently locked, and *condition* is **MSEM_IF_NOWAIT**, then the function returns with an error. If the semaphore is currently locked and *condition* is zero, the function does not return until either the calling process is able to successfully lock the semaphore, or an error condition occurs.

All calls to **msem_lock()** and **msem_unlock()** by multiple processes sharing a common **msemaphore** structure behave as if the calls were serialized.

If the **msemaphore** structure contains any value not resulting from a call to **msem_init()** followed by a (possibly empty) sequence of calls to **msem_lock()** and **msem_unlock()**, the results are undefined. The address of an **msemaphore** uniquely identifies the semaphore. If the **msemaphore** structure contains any value copied from an **msemaphore** structure at a different address, the result is undefined.

IMPLEMENTATION NOTES

If blocked on a locked semaphore, **msem_lock()** suspends the calling process at a priority such that the process can be interrupted by a signal.

The system attempts to ignore or recover from invalid values written to the **msemaphore** structure, but this is not guaranteed for all cases.

m

msem_lock() successfully acquires a semaphore that is locked by a process that has exited.

RETURN VALUE

Upon success, **msem_lock()** returns zero; otherwise, it returns –1 and sets **errno** to indicate the error.

ERRORS

msem_lock() fails if any of the following conditions are encountered:

[EAGAIN]	**MSEM_IF_NOWAIT** was specified and the semaphore was already locked.
[EINVAL]	*sem* points to an **msemaphore** structure that has been removed, or *condition* is invalid.
[EINTR]	**msem_lock()** was interrupted by a signal that was caught.
[EDEADLK]	The semaphore is currently locked, *condition* is zero, and waiting to lock the semaphore would create a deadlock.
[EFAULT]	*sem* is not a properly aligned address or is otherwise an invalid pointer.

AUTHOR

msem_lock() was developed by HP and OSF.

SEE ALSO

msem_init(2), msem_remove(2), msem_unlock(2), mman(5).

STANDARDS CONFORMANCE

msem_lock(): AES

NAME

msem_remove - remove a semaphore in mapped file or anonymous region

SYNOPSIS

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

int *msem_remove(msemaphore *sem);
```

DESCRIPTION

msem_remove() removes a binary semaphore.

sem points to an **msemaphore** structure that specifies the semaphore to be removed. Any subsequent use of the **msemaphore** structure before it is again initialized by calling **msem_init()** produces undefined results.

msem_remove() also causes any process waiting in the **msem_lock()** function on the removed semaphore to return with an error.

If the **msemaphore** structure contains any value not resulting from a call to **msem_init()** followed by a (possibly empty) sequence of calls to **msem_lock()** and **msem_unlock()**, the results are undefined. The address of an **msemaphore** uniquely identifies the semaphore. If the **msemaphore** structure contains any value copied from a **msemaphore** structure at a different address, the result is undefined.

RETURN VALUE

Upon success, **msem_remove()** returns zero; otherwise, it returns –1 and sets **errno** to indicate the error.

ERRORS

msem_remove() fails if any of the following conditions are encountered:

[EINVAL]	*sem* points to an **msemaphore** structure that has been removed.
[EFAULT]	*sem* is an invalid pointer.

AUTHOR

msem_remove() was developed by HP and OSF.

SEE ALSO

msem_init(2), msem_lock(2), msem_remove(2), mman(5).

STANDARDS CONFORMANCE

msem_remove(): AES

NAME

msem_unlock - unlock a semaphore

SYNOPSIS

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

int msem_unlock(msemaphore *sem, int condition);
```

DESCRIPTION

msem_unlock() unlocks a binary semaphore.

sem points to an **msemaphore** structure that specifies the semaphore to be unlocked.

If the *condition* argument is zero, the semaphore will be unlocked, whether or not any other processes are currently attempting to lock it. If the *condition* argument is **MSEM_IF_WAITERS**, and some other process is waiting to lock the semaphore or the implementation cannot reliably determine whether some process is waiting to lock the semaphore, the semaphore is unlocked by the calling process. If the *condition* argument is **MSEM_IF_WAITERS**, and no process is waiting to lock the semaphore, the semaphore is not unlocked and an error is returned.

All calls to **msem_lock()** and **msem_unlock()** by multiple processes sharing a common **msemaphore** structure behave as if the calls were serialized.

If the **msemaphore** structure contains any value not resulting from a call to **msem_init()** followed by a (possibly empty) sequence of calls to **msem_lock()** and **msem_unlock()**, the results are undefined. The address of an **msemaphore** uniquely identifies the semaphore. If the **msemaphore** structure contains any value copied from a **msemaphore** structure at a different address, the result is undefined.

IMPLEMENTATION NOTES

The system attempts to ignore or recover from invalid values placed in the **msemaphore** structure, but this is not guaranteed for all cases.

RETURN VALUE

Upon success, **msem_unlock()** returns zero; otherwise, it returns –1 and sets **errno** to indicate the error.

ERRORS

msem_unlock() fails if any of the following conditions are encountered:

[EAGAIN]	**MSEM_IF_NOWAIT** was specified and there were no waiters.
[EINVAL]	*sem* points to an **msemaphore** structure that has been removed, or *condition* is invalid.
[EFAULT]	*sem* is an invalid pointer.

AUTHOR

msem_unlock() was developed by HP and OSF.

SEE ALSO

msem_init(2), msem_lock(2), msem_remove(2), mman(5).

STANDARDS CONFORMANCE

msem_unlock(): AES

NAME

msgctl - message control operations

SYNOPSIS

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

int msgctl(int msqid, int 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 *glossary*(9).

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*.

m

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

msgctl() fails if one or more of the following is true:

[EINVAL] *msqid* is not a valid message queue identifier.

[EINVAL] *cmd* is not a valid command, or the command contains invalid parameters.

[EACCES] *cmd* is equal to **IPC_STAT** and Read operation permission is denied to the calling process (see *message operation permissions* in *glossary*(9)).

[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 a user who has appropriate privileges 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. Reliable detection of this error is implementation dependent.

SEE ALSO

ftok(3C), ipcrm(1), ipcs(1), msgget(2), msgop(2).

STANDARDS CONFORMANCE

msgctl(): SVID2, SVID3, XPG2, XPG3, XPG4

NAME

msgget - get message queue

SYNOPSIS

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

int msgget(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.

m

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.

ERRORS

msgget() fails if one or more of the following is 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".

SEE ALSO

ipcrm(1), ipcs(1), msgctl(2), msgop(2), stdipc(3C).

STANDARDS CONFORMANCE

msgget(): SVID2, SVID3, XPG2, XPG3, XPG4

NAME

msgsnd, msgrcv - message operations

SYNOPSIS

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

int msgsnd(
    int          msqid,
    const void *msgp,
    size_t       msgsz,
    int          msgflg
);

int msgrcv(
    int     msqid,
    void   *msgp,
    size_t  msgsz,
    long    msgtyp,
    int     msgflg
);
```

DESCRIPTION

The **msgsnd()** system call sends 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 specifies 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 is true:

- The number of bytes already on the queue is equal to **msg_qbytes** (see **message queue identifier** in *glossary*(9)).
- 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 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 to the process ID of the calling process.

msg_stime is set to the current time.

The **msgrcv()** system call 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:

msgtyp = 0 First message on the queue is received.

msgtyp > 0 First message of type *msgtyp* is received.

msgtyp < 0 First message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

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

If (*msgflg* **& IPC_NOWAIT**) is true, the calling process returns immediately with a value of **-1** and **errno** set to [ENOMSG].

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

- A message of the desired type is placed on the queue.
- *msqid* is removed from the system. When this occurs, **errno** is set 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 to the process ID of the calling process.

msg_rtime is set to the current time.

m

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.

ERRORS

If **msgrcv()** fails, **errno** is set to one of the following values.

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

If **msgsnd()** fails, **errno** is set to one of the following values.

[EACCES]	Operation permission is denied to the calling process.
[EAGAIN]	The message cannot be sent for one of the reasons cited above 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]	**msgsnd()** was interrupted by a signal.
[EINVAL]	*msqid* is not a valid message queue identifier.
[EINVAL]	*mtype* is less than 1.
[EINVAL]	*msgsz* is less than zero or greater than the system-imposed limit.

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), stdipc(3C), signal(5).

STANDARDS CONFORMANCE

msgrcv(): SVID2, SVID3, XPG2, XPG3, XPG4

msgsnd(): SVID2, SVID3, XPG2, XPG3, XPG4

NAME

msync - synchronize memory with physical storage

SYNOPSIS

#include <sys/mman.h>

int msync(void *addr,size_t len, int flags);

DESCRIPTION

The **msync()** function writes all modified copies of pages over the range [*addr*, *addr+len*] to the underlying hardware, or invalidates any copies so that further references to the pages will be obtained by the system from their permanent storage locations.

The *flags* argument is one of the following:

MS_ASYNC	perform asynchronous writes
MS_SYNC	perform synchronous writes
MS_INVALIDATE	invalidate mappings

If *flags* is **MS_ASYNC** or **MS_SYNC**, the function synchronizes the file contents to match the current contents of the memory region.

- All write references to the memory region made prior to the call are visible by subsequent read operations on the file.
- It is unspecified whether writes to the same portion of the file prior to the call are visible by read references to the memory region.
- It is unspecified whether unmodified pages in the specified range are also written to the underlying hardware.

m

If flags is **MS_ASYNC**, the function may return immediately once all write operations are scheduled; if *flags* is **MS_SYNC**, the function does not return until all write operations are completed.

If *flags* is **MS_INVALIDATE**, the function synchronizes the contents of the memory region to match the current file contents.

- All writes to the mapped portion of the file made prior to the call are visible by subsequent read references to the mapped memory region.
- It is unspecified whether write references prior to the call, by any process, to memory regions mapped to the same portion of the file using **MAP_SHARED**, are visible by read references to the region.

If **msync()** causes any write to the file, then the file's *st_ctime* and *st_mtime* fields are marked for update.

RETURN VALUE

Upon successful completion, **msync()** returns 0. Otherwise, it returns −1 and sets **errno** to indicate the error.

ERRORS

The **msync()** function will fail if:

[EINVAL]	The *addr* argument is not a multiple of the page size as returned by **sysconf()**.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[ENOMEM]	Some or all the addresses in the range [*addr*, *addr+len*] are invalid for the address space of the process or pages not mapped are specified.

APPLICATION USAGE

The **msync()** function should be used by programs that require a memory object to be in a known state, for example in building transaction facilities.

Normal system activity can cause pages to be written to disk. Therefore, there are no guarantees that **msync()** is the only control over when pages are or are not written to disk.

SEE ALSO

mmap(2), sysconf(2), <sys/mman.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

HP-UX EXTENSIONS

NAME

msync - synchronize a mapped file

SYNOPSIS

int msync(caddr_t addr,size_t len, int flags);

DESCRIPTION

msync controls the caching operations of a mapped file region.

addr and *len* specify the region to be synchronized. If these are not the address and length of a region created by a previous successful call to **mmap()**, **msync()** returns an error. The behavior of **msync()** upon a region created with the **MAP_ANONYMOUS** or **MAP_PRIVATE** flags is undefined.

After a successful call to **msync()** with **MS_SYNC** specified, all previous modifications to the mapped region are visible to processes using **read()**. Previous modifications to the file using **write()** may be lost.

After a successful call to **msync()** with only **MS_INVALIDATE** specified, all previous modifications to the file using **write()** are visible to the mapped region. Previous direct modifications to the mapped region may be lost.

Performance Considerations

The following performance considerations only apply when using the **MS_INVALIDATE** option with **msync()**. These performance constraints do not apply when either **MS_ASYNC** or **MS_SYNC** are exclusively used with **msync()**.

Direct read/write references to portions of a mapped memory region currently undergoing an **msync()** operation (with MS_INVALIDATE specified), may be blocked until all scheduled write operations are completed. This is especially true when performing an **msync()** operation across a relatively large address range that requires many individual write operations to be scheduled out to the underlying hardware. HP-UX will schedule a separate write operation for each contiguous group of modified pages on disk. As more write operations are queued out to the device, the overall suspension time of direct read/write references to the same portions of the memory region will generally increase.

The suspension times of direct read/write references can be reduced by issuing **msync()** requests over smaller portions of the memory region, but issuing them more frequently than a corresponding larger synchronization request. This will serve to more evenly distribute I/O activity across the mapped file, while reducing the number of write operations per **msync()**.

ERRORS

[EINVAL] *addr* is not a multiple of the page size as returned by **sysconf(_SC_PAGE_SIZE)**.

[EINVAL] The address range specified by *addr* and *len* was not created by a successful call to **mmap()**.

AUTHOR

msync() was developed by HP, AT&T, and OSF.

SEE ALSO

mmap(2), sysconf(2).

STANDARDS CONFORMANCE

msync(): AES, SVID3

m

NAME
munlock() - unlock a segment of the process virtual address space

SYNOPSIS
```
#include <sys/mman.h>

int munlock( const void * addr, size_t len) ;
```

DESCRIPTION
The **munlock()** system call allows the calling process to unlock a segment of the process virtual address space that may have been previously locked with **mlock()** or **mlockall()**. Upon successful completion of the **munlock()**, pages within the specified segment are subject to routine paging and/or swapping.

addr must be a valid address in the process virtual address space. *addr* + *len* must also be a valid address in the process virtual address space.

Pages are unlocked at page boundaries that encompass the range from *addr* to *addr* + *len*. If any address within the range is not a valid part of the process virtual address space, an error is returned and no unlocks are performed. However, no error is reported for valid pages within the range that are not already locked, since their state at the completion of the **munlock()** call is as desired.

Regardless of how many times a process locks a page, a single **munlock()** or **munlockall()** will unlock it. An **munlock()** of a page within a range specified in an **mlock()** call results in only the range specified in the **munlock()** being unlocked.

When memory is shared by multiple processes and mlocks are applied to the same physical page by multiple processes, a page remains locked until the last lock is removed from that page.

The effective user ID of the calling process must be a superuser or the user must be a member of a group that has the MLOCK privilege (see *getprivgrp*(2) and *setprivgrp*(1M)).

Although **plock()** and the **mlock()** family of functions may be used together in an application, each may affect the other in unexpected ways. This practice is not recommended.

m

RETURN VALUE
munlock() returns the following values:

- **0** Successful completion.
- **-1** Failure. The requested operation is not performed. **errno** is set to indicate the error.

ERRORS
If **munlock()** fails, **errno** is set to one of the following values:

[ENOMEM] One or more addresses in the specified range is not valid within the process address space.

[EINVAL] The **len** parameter was zero.

[EPERM] The effective user ID of the calling process is not a superuser and the user does not belong to a group that has the **MLOCK** privilege.

EXAMPLES
The following call to **munlock()** unlocks the first 10 pages of the calling process address space:

```
munlock(sbrk(0), 40960);
```

SEE ALSO
setprivgrp(1M), getprivgrp(2), mlock(2), mlockall(2), munlockall(2), plock(2).

STANDARDS CONFORMANCE
munlock(): POSIX Realtime Extensions, IEEE Std 1003.1b

NAME

munlockall() - unlock the entire virtual address space of a process

SYNOPSIS

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

int munlockall() ;
```

DESCRIPTION

The **munlockall()** system call allows the calling process to unlock any portions of its virtual address space that have previously been locked into memory with **mlock()** or **mlockall()**, including any portions locked due to the **MCL_FUTURE** option of **mlockall()**. Upon successful completion of the **munlockall()**, all pages within the process virtual address space are subject to routine paging and/or swapping and the **MCL_FUTURE** option will no longer be in effect for the process.

Regardless of how many times a process locks a page, a single **munlockall()** will unlock it. When memory is shared by multiple processes and mlocks are applied to the same physical page by multiple processes, a page remains locked until the last lock is removed from that page.

The effective user ID of the calling process must be a superuser or the user must be a member of a group that has the MLOCK privilege (see *getprivgrp*(2) and *setprivgrp*(1M)).

Although **plock()** and the **mlock()** family of functions may be used together in an application, each may affect the other in unexpected ways. This practice is not recommended.

RETURN VALUE

munlockall() returns the following values:

- **0** Successful completion.
- **-1** Failure. The requested operation is not performed. **errno** is set to indicate the error.

m

ERRORS

If **munlockall()** fails, **errno** is set to the following value:

[EPERM] The effective user ID of the calling process is not a superuser and the user does not belong to a group that has the **MLOCK** privilege.

EXAMPLES

The following call to **munlockall()** unlocks the process virtual address space:

```
munlockall();
```

SEE ALSO

setprivgrp(1M), getprivgrp(2), mlock(2), mlockall(2), munlock(2), plock(2).

STANDARDS CONFORMANCE

munlockall(): POSIX Realtime Extensions, IEEE Std 1003.1b

NAME

munmap - unmap pages of memory

SYNOPSIS

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

int munmap(void *addr, size_t len);
```

DESCRIPTION

The **munmap()** function removes the mappings for pages in the range [*addr*, *addr+len*], rounding the *len* argument up to the next multiple of the page size as returned by **sysconf()**. If *addr* is not the address of a mapping established by a prior call to **mmap()**, the behaviour is undefined. After a successful call to **munmap()** and before any subsequent mapping of the unmapped pages, further references to these pages will result in the delivery of a **SIGBUS** or **SIGSEGV** signal to the process.

RETURN VALUE

Upon successful completion, **munmap()** returns 0. Otherwise, it returns –1 and sets **errno** to indicate the error.

ERRORS

The **munmap()** function will fails if:

[EINVAL] The *addr* argument is not a multiple of the page size as returned by **sysconf()**.

[EINVAL] Addresses in the range [*addr*, *addr+len*], are outside the valid range for the address space of a process.

[EINVAL] The *len* argument is 0.

SEE ALSO

mmap(2), sysconf(2), <signal.h>, <sys/mman.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

HP-UX EXTENSIONS

SYNOPSIS

`int munmap(caddr_t addr, size_t len);`

DESCRIPTION

`munmap()` unmaps a mapped file or anonymous memory region.

If the address range specified by *addr* and *len* was not created by a successful call to **`mmap()`**, **`munmap()`** returns an error.

If the specified address range was created by multiple calls to **`mmap()`**, **`munmap()`** succeeds in unmapping all of the specified regions, provided they form a contiguous address range.

If the region was created with the **`MAP_PRIVATE`** option, any modifications made to the region are discarded.

ERRORS

[EINVAL] *addr* is not a multiple of the page size as returned by **`sysconf(_SC_PAGE_SIZE)`**.

[EINVAL] The address range specified by *addr* and *len* was not created by a successful call to **`mmap()`**.

AUTHOR

`munmap()` was developed by HP, AT&T, and OSF.

SEE ALSO

mmap(2), sysconf(2).

m

STANDARDS CONFORMANCE

`munmap()`: AES, SVID3

NAME

nanosleep() - high resolution sleep

SYNOPSIS

```
#include <time.h>

int nanosleep(
     const struct timespec *rqtp,
     struct timespec *rmtp
);
```

DESCRIPTION

The **nanosleep()** function causes the current process to be suspended from execution until either the time interval specified by the **rqtp** argument has elapsed, or a signal is delivered to the calling process and its action is to invoke a signal-catching function or to terminate the process. The suspension time may be longer than that requested because the argument value is rounded up to an integer multiple of the sleep resolution or because of the scheduling of other activity by the system. But, except for the case of being interrupted by a signal, the suspension time will not be less than the time specified by **rqtp**, as measured by the system clock, **CLOCK_REALTIME**.

The use of the **nanosleep()** function has no effect on the action or blockage of any signal.

RETURN VALUE

If the **nanosleep()** function returns because the requested time has elapsed, its return value is zero.

If the **nanosleep()** function returns because it has been interrupted by a signal, the function returns a value of –1 and sets **errno** to indicate the interruption. If the **rmtp** argument is non-NULL, the *timespec* structure referenced by it is updated to contain the amount of time remaining in the interval (the requested time minus the time actually slept). If the **rmtp** argument is NULL, the remaining time is not returned.

If **nanosleep()** fails, it returns a value of –1 and sets **errno** to indicate the error.

ERRORS

If any of the following conditions occur, the **nanosleep()** function returns –1 and sets **errno** (see *errno*(2)) to the corresponding value:

[EINTR] **nanosleep()** was interrupted by a signal.

[EINVAL] The **rqtp** argument specified a nanosecond value less than zero or greater than or equal to 1000 million.

[ENOSYS] The function **nanosleep()** is not supported by this implementation.

[EFAULT] The **rqtp** or **rmtp** arguments specify an invalid address.

EXAMPLES

Suspend execution of the current process for half a second:

```
#include <time.h>
#include <errno.h>

struct timespec interval, remainder;

interval.tv_sec = 0;
interval.tv_nsec = 500000000;
if (nanosleep(&interval, &remainder) == -1) {
    if (errno == EINTR) {
        (void)printf("nanosleep interrupted\n");
        (void)printf("Remaining secs: %d\n", remainder.tv_sec);
        (void)printf("Remaining nsecs: %d\n", remainder.tv_nsec);
    }
    else perror("nanosleep");
}
```

AUTHOR

nanosleep was derived from the proposed IEEE POSIX P1003.4 Standard, Draft 14.

n

SEE ALSO

clocks(2), timers(2), sleep(3C).

STANDARDS CONFORMANCE

nanosleep() : POSIX.4

n

NAME

nice - change priority of a process

SYNOPSIS

```
#include <unistd.h>

int nice(int priority_change);
```

DESCRIPTION

nice() adds the value of *priority_change* 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.

If the calling process contains more than one thread or lightweight process (i.e., the process is multi-threaded) this function shall apply to all threads or lightweight processes in the calling process.

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 a user having appropriate privileges has changed the user process priority value to something less than 20, certain values for *priority_change* can cause **nice()** to return −1, which is indistinguishable from an error return.

ERRORS

[EPERM] **nice()** fails and does not change the nice value if *priority_change* is negative or greater than 40, and the effective user ID of the calling process is not a user having appropriate privileges.

SEE ALSO

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

STANDARDS CONFORMANCE

nice(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

NAME

open() - open file for reading or writing

SYNOPSIS

```
#include <fcntl.h>

int open(const char *path, int oflag, ... /* [mode_t mode] */ );
```

Remarks

The ANSI C "**, ...**" construct specifies a variable length argument list whose optional member is given in the associated comment (**/* */**).

DESCRIPTION

The **open()** system call opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*.

If the system call is made in 64 bit mode, the O_LARGEFILE status flag is automatically set in addition to the value of *oflag* (see *fcntl*(5)).

The *path* argument points to a path name naming a file, and must not exceed **PATH_MAX** bytes in length.

The *oflag* argument is a value that is the bitwise inclusive OR of flags listed in "Read-Write Flags," "General Flags," and "Synchronized I/O Flags" below.

The optional *mode* argument is only effective when the **O_CREAT** flag is specified.

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).

Read-Write Flags

Exactly one of the **O_RDONLY**, **O_WRONLY**, or **O_RDWR** flags must be used in composing the value of *oflag*. If none or more than one is used, the behavior is undefined.

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O

General Flags

Several of the flags listed below can be changed with the **fcntl()** system call while the file is open. See *fcntl*(2) and *fcntl*(5) for details.

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 new 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 mode is cleared (see *umask*(2)).
- The "save text image after execution" bit of the new file 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_EXCL If **O_EXCL** and **O_CREAT** are set and the file exists, **open()** fails.

O_LARGEFILE

When the filesystem is mounted as large files enabled and the file is opened with the **O_LARGEFILE** option, the file can grow over 2 GB.

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:

A read-only **open()** returns without delay.

A write-only **open()** returns an error if no process currently has the file open for reading.

If **O_NDELAY** is clear:

A read-only **open()** does not return until a process opens the file for writing.

A write-only **open()** 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_NOCTTY If set, and *path* identifies a terminal device, **open()** does not cause the terminal to become the controlling terminal for the process.

O_NONBLOCK Same effect as **O_NDELAY** for *open*(2), but slightly different effect in *read*(2) and *write*(2). If both **O_NONBLOCK** and **O_NDELAY** are specified, **O_NONBLOCK** takes precedence.

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

Synchronized I/O Flags

Together, the **O_DSYNC**, **O_RSYNC**, and **O_SYNC** flags constitute support for Synchronized I/O. These flags are ignored for files other than ordinary files and block special files on those systems that permit I/O to block special devices (see *pathconf*(2)). If both the **O_DSYNC** and **O_SYNC** flags are set, the effect is as if only the **O_SYNC** flag was set. The **O_RSYNC** flag is ignored if it is not set along with the **O_DSYNC** or **O_SYNC** flag.

O_DSYNC

If a file is opened with **O_DSYNC** or that flag is set with the **F_SETFL** option of **fcntl()**, writes to that file by the process block until the data specified in the write request and all file attributes required to retrieve the data are written to the disk. File attributes that are not necessary for data retrieval (access time, modification time, status change time) are not necessarily written to the disk prior to returning to the calling process.

O_SYNC

Identical to **O_DSYNC**, with the addition that all file attributes changed by the write operation (including access time, modification time, and status change time) are also written to the disk prior to returning to the calling process.

O_RSYNC|O_DSYNC (specified together)

Identical to **O_DSYNC** for file system writes.

For file system reads, the calling process blocks until the data being read and all file attributes required to retrieve the data are the same as their image on disk. Writes pending on the data to be read are executed prior to returning to the calling process.

O_RSYNC|O_SYNC (specified together)

Identical to **O_SYNC** for file system writes.

Identical to **O_RSYNC|O_DSYNC** for file system reads, with the addition that all file attributes changed by the read operation (including access time, modification time, and status change time) too are the same as their image on disk.

RETURN VALUE

open() returns the following values:

n Successful completion. *n* is a file descriptor for the opened file.
-1 Failure. **errno** is set to indicate the error.

ERRORS

If **open()** fails, **errno** is set to one of the following values.

- [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.
- [EACCES] **O_TRUNC** is specified and write permission is denied.
- [EAGAIN] The file exists, enforcement mode file/record locking is set (see *chmod*(2)), there are outstanding record locks on the file with the **lockf()** or **fcntl()** system calls, and **O_TRUNC** is set.
- [EDQUOT] User's disk quota block or inode limit has been reached for this file system.
- [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 either does not exist, or the driver for this device has not been configured into the kernel.
- [ENOSPC] **O_CREAT** is specified, the file does not already exist, and the directory that would contain the file cannot be extended.
- [EOVERFLOW] The named file is a regular file and the size of the file cannot be represented correctly in an object of size **off_t**.
- [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 can be kept open for a long time, or indefinitely under some circumstances.

O

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 entry.

```
int infd;
infd = 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 entry. For an example of writing to **outputfile**, see the *write*(2) manual entry.

```
int outfd;
outfd = open ("outputfile", O_WRONLY);
```

The following call opens file **iofile** for synchronized I/O file integrity for reads and writes.

```
int iofd;
iofd = open ("iofile", O_RDWR|O_SYNC|O_RSYNC);
```

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), creat64(2), pathconf(2), read(2), select(2), umask(2), write(2), setacl(2), acl(5), fcntl(5), signal(5), unistd(5).

STANDARDS CONFORMANCE

open(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, POSIX.4

O

NAME

pathconf(), fpathconf() - get configurable path name variables

SYNOPSIS

```
#include <unistd.h>

long pathconf(const char *path, int name);

long fpathconf(int fildes, int 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
	_PC_FILESIZEBITS	3, 4, 10
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_SYNC_IO	_PC_SYNC_IO	9
_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 path name 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 file names 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 if *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.

9. **_POSIX_SYNC_IO**, when defined, determines whether synchronized IO operations may be performed for the associated file (see *open*(2)). If *path* or *fildes* refers to a directory, it is unspecified whether or not the implementation supports an association of the variable name with the specified file.

10. For file systems that are not large file enabled, the **_PC_FILESIZEBITS** return value will be less than or equal to 32. For file systems that are large file enabled, the **_PC_FILESIZEBITS** return value will be between 33 and 63.

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()** fail if any of the following conditions are encountered:

[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.

p

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*. The first determines whether a file name longer than **NAME_MAX** bytes will be truncated to **NAME_MAX** bytes in the **/tmp** directory. If so, the second call is made 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

The following error can occur:

[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_PIPE_BUF**, **_PC_PATH_MAX**, **_PC_NO_TRUNC**, and **_PC_CHOWN_RESTRICTED**, cannot be determined for an NFS file.

AUTHOR

pathconf() and **fpathconf()** were developed by HP.

SEE ALSO

chown(2), errno(2), limits(5), unistd(5), termio(7).

STANDARDS CONFORMANCE

pathconf(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1, POSIX.2, POSIX.4

fpathconf(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1, POSIX.2, POSIX.4

p

NAME

pause - suspend process until signal

SYNOPSIS

```
#include <unistd.h>

int pause(void);
```

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()** does 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()** can affect the behavior described on this page.

APPLICATION USAGE

Threads Considerations

Signal dispositions (such as catch/default/ignore) are shared by all threads in the process and blocked signal masks are maintained by each thread. Therefore, the signals being waited for should not be ignored by the process or blocked by the calling thread.

pause() will suspend only the calling thread until it receives a signal.

If other threads in the process do not block the signal, the signal may be delivered to another thread in the process and the thread in **pause()** may continue waiting. For this reason, the use of **sigwait()** is recommended instead of **pause()** for multi-threaded applications.

For more information regarding signals and threads, refer to *signal*(5).

SEE ALSO

alarm(2), kill(2), sigvector(2), sigwait(2), wait(2), signal(5).

STANDARDS CONFORMANCE

pause(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

pipe - create an interprocess channel

SYNOPSIS

```
int pipe(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).

By default, HP-UX pipes are not STREAMS-based. It is possible to generate the kernel so that all pipes created on a system are STREAMS-based. This can only be done for HP-UX releases 10.0 and later. STREAMS-based FIFOs (created by **mknod** or **mkfifo**) are not supported on HP-UX.

To generate a kernel that supports STREAMS-based pipes:

- STREAMS/UX must be installed.
- The module **pipemod** and the driver **pipedev** must be included in the **/stand/system** file. (When STREAMS/UX is installed, **pipemod** and **pipedev** are automatically added to the system file.)
- The tunable parameter "streampipes" must be set to 1 in the **/stand/system** file. (This is not automatically done when STREAMS/UX is installed.)
- The kernel must be generated and the system rebooted. Once this is done, all pipes created by **pipe()** will be STREAMS-based.

For more information, see *STREAMS/UX for the HP 9000 Reference Manual*.

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) /* check process id of child process */ {
     close(1);  /* close stdout */
     dup (pipefd[1]); /* points pipefd at file descriptor */
     close (pipefd[0]);
     execlp ( ls", ls , (char *)0);
}
else if (pid > (pid_t)0) {
     close(0);  /* close stdin  */
     dup (pipefd[0]);
     /* point the child's standard output to parent's standard input */
     close (pipefd[1]);
     execlp ("sort", "sort", (char *)0); /* parent process does sort */
}
```

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.

p

[ENOSPC] The file system lacks sufficient space to create the pipe.

[ENOSR] Could not allocate resources for both Stream heads (STREAMS-based pipes only).

SEE ALSO

sh(1), read(2), write(2), popen(3S), streamio(7).

STANDARDS CONFORMANCE

pipe(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

plock() - lock process, text, data, stack, or shared library in memory

SYNOPSIS

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

int plock(int op);
```

DESCRIPTION

The **plock()** system call 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. Stack segments are also locked when data segments are locked. Shared library text and shared library data segments (shlib lock) can also be locked. Locked segments are immune to all routine swapping. **plock()** also allows these segments to be unlocked.

The effective user ID of the calling process must be a superuser or the user must be a member of a group that has the MLOCK privilege (see *getprivgrp*(2) and *setprivgrp*(1M)).

op must be one of 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
SHLIBLOCK	Lock shared library text and shared library data segments (shared library lock)
PROCSHLIBLOCK	Lock text, data and shared library text and shared library data segments into memory (process and shared library lock)
TXTSHLIBLOCK	Lock text, shared library text and shared library data segments into memory (text and shared library lock)
DATSHLIBLOCK	Lock data, shared library text and shared library data segments into memory (data and shared library lock)

RETURN VALUE

plock() returns the following values:

p

- **0** Successful completion.
- **-1** Failure. The requested operation is not performed. **errno** is set to indicate the error.

ERRORS

If **plock()** fails, **errno** is set to one of the following values.

[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 process lock already exists on the calling process.
[EINVAL]	*op* is equal to **DATLOCK** and a data lock, or 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 equal to **SHLIBLOCK** and there are no unlocked shared library segments in the calling process.
[EINVAL]	*op* is equal to **PROCSHLIBLOCK** and a process lock, a text lock, or a data lock already exists on the calling process.
[EINVAL]	*op* is equal to **TXTSHLIBLOCK** and a text lock or process lock already exists on the calling process.
[EINVAL]	*op* is equal to **DATSHLIBLOCK** and a data lock, or process lock already exists on the calling process.
[EINVAL]	*op* is not equal to one of the values specified in DESCRIPTION.

[EINVAL] **plock()** is not allowed in a [**vfork**,**exec**] window. See *vfork*(2).

[ENOMEM] There is not enough lockable memory in the system to satisfy the locking request.

[EPERM] The effective user ID of the calling process is not a superuser and the user does not belong to a group that has the **MLOCK** privilege.

EXAMPLES

The following call to **plock()** locks the calling process in memory:

```
plock(PROCLOCK);
```

SEE ALSO

setprivgrp(1M), exec(2), exit(2), fork(2), getprivgrp(2), vfork(2).

STANDARDS CONFORMANCE

plock(): SVID2, SVID3, XPG2

p

NAME

poll - monitor I/O conditions on multiple file descriptors

SYNOPSIS

```
#include <poll.h>

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

DESCRIPTION

poll() provides a general mechanism for reporting I/O conditions associated with a set of file descriptors and for waiting until one or more specified conditions becomes true. Specified conditions include the ability to read or write data without blocking, and error conditions.

Arguments

fds Points to an array of **pollfd** structures, one for each file descriptor of interest.

nfds Specifies the number of **pollfd** structures in the *fds* array.

timeout Specifies the maximum length of time (in milliseconds) to wait for at least one of the specified conditions to occur.

Each **pollfd** structure includes the following members:

int fd	File descriptor
short events	Requested conditions
short revents	Reported conditions

The **fd** member of each **pollfd** structure specifies an open file descriptor. The **poll()** function uses the **events** member to determine what conditions to report for this file descriptor. If one or more of these conditions is true, **poll()** sets the associated **revents** member.

poll() ignores any **pollfd** structure whose **fd** member is negative. If the **fd** member of all **pollfd** structures is negative, **poll()** returns 0 and has no other results.

The **events** and **revents** members of the **pollfd** structure are bit masks. The calling process sets the **events** bit mask, and **poll()** sets the **revents** bit masks. These bit masks contain ORed combinations of condition flags. The following condition flags are defined:

POLLIN Data can be read without blocking. For streams, this flag means that a message that is not high priority is at the front of the stream head read queue. This message can be of zero length.

POLLNORM Synonym for **POLLIN**

POLLPRI A high priority message is available. For streams, this message can be of zero length.

POLLOUT Data can be written without blocking. For streams, this flag specifies that normal data (not high priority or priority band > 0) can be written without being blocked by flow control. This flag is not used for high priority data, because it can be written even if the stream is flow controlled.

POLLERR An error has occurred on the file descriptor.

POLLHUP The device has been disconnected. For streams, this flag in **revents** is mutually exclusive with **POLLOUT**, since a stream cannot be written to after a hangup occurs. This flag and **POLLIN**, **POLLPRI**, **POLLRDNORM**, **POLLRDBAND**, and **POLLMSG** are not mutually exclusive.

POLLNVAL **fd** is not a valid file descriptor.

POLLRDNORM A non-priority message is available. For streams, this flag means that a normal message (not high priority or priority band > 0) is at the front of the stream head read queue. This message can be of zero length.

POLLRDBAND A priority message (priority band > 0) is at the front of the stream head read queue. This message can be read without blocking. The message can be of zero length.

POLLWRNORM Same as **POLLOUT**

POLLWRBAND Priority data (priority band > 0) can be written without being blocked by flow control. Only previously written bands are checked.

POLLMSG A **M_SIG** or **M_PCSIG** message specifying **SIGPOLL** has reached the front of the stream head read queue.

The conditions indicated by **POLLNORM** and **POLLOUT** are true if and only if at least one byte of data can be read or written without blocking. The exception is regular files, which always poll true for **POLLNORM** and **POLLOUT**. Also, streams return **POLLNORM** in **revents** even if the available message is of zero length.

The condition flags **POLLERR**, **POLLHUP**, and **POLLNVAL** are always set in **revents** if the conditions they indicate are true for the specified file descriptor, whether or not these flags are set in **events**.

For each call to **poll()**, the set of reportable conditions for each file descriptor consists of those conditions that are always reported, together with any further conditions for which flags are set in **events**. If any reportable condition is true for any file descriptor, **poll()** returns with flags set in **revents** for each true condition for that file descriptor.

If no reportable condition is true for any of the file descriptors, **poll()** waits up to *timeout* milliseconds for a reportable condition to become true. If, in that time interval, a reportable condition becomes true for any of the file descriptors, **poll()** reports the condition in the file descriptor's associated **revents** member and returns. If no reportable condition becomes true, **poll()** returns without setting any **revents** bit masks.

If the *timeout* parameter is a value of −1, **poll()** does not return until at least one specified event has occurred. If the value of the *timeout* parameter is 0, **poll()** does not wait for an event to occur but returns immediately, even if no specified event has occurred. The behavior of **poll()** is not affected by whether the **O_NONBLOCK** flag is set on any of the specified file descriptors.

RETURN VALUES

Upon successful completion, **poll()** returns a nonnegative value. If the call returns 0, **poll()** has timed out and has not set any of the **revents** bit masks. A positive value indicates the number of file descriptors for which **poll()** has set the **revents** bit mask. If **poll()** fails, it returns −1 and sets **errno** to indicate the error.

ERRORS

poll() fails if any of the following conditions are encountered:

[EAGAIN] Allocation of internal data structures failed. A later call to **poll()** may complete successfully.

[EINTR] A signal was delivered before any of the selected for conditions occurred or before the time limit expired.

[EINVAL] *timeout* is a negative number other than −1.

[EFAULT] The *fds* parameter in conjunction with the *nfds* parameter addresses a location outside of the allocated address space of the process. Reliable detection of this error is implementation-dependent.

p

EXAMPLES

Wait for input on file descriptor 0:

```
#include <poll.h>
struct pollfd fds;

fds.fd = 0;
fds.events = POLLNORM;
poll(&fds, 1, -1);
```

Wait for input on **ifd1** and **ifd2**, output on **ofd**, giving up after 10 seconds:

```
#include <poll.h>
struct pollfd fds[3];
int ifd1, ifd2, ofd, count;

fds[0].fd = ifd1;
fds[0].events = POLLNORM;
fds[1].fd = ifd2;
fds[1].events = POLLNORM;
fds[2].fd = ofd;
```

```
fds[2].events = POLLOUT;
count = poll(fds, 3, 10000);
if (count == -1) {
        perror("poll failed");
        exit(1);
}
if (count==0)
        printf("No data for reading or writing\n");
if (fds[0].revents & POLLNORM)
        printf("There is data for reading fd %d\n", fds[0].fd);
if (fds[1].revents & POLLNORM)
        printf("There is data for reading fd %d\n", fds[1].fd);
if (fds[2].revents & POLLOUT)
        printf("There is room to write on fd %d\n", fds[2].fd);
```

Check for input or output on file descriptor 5 without waiting:

```
#include <poll.h>
struct pollfd fds;

fds.fd = 5;
fds.events = POLLNORM|POLLOUT;
poll(&fds, 1, 0);
if (fds.revents & POLLNORM)
        printf("There is data available on fd %d\n", fds.fd);
if (fds.revents & POLLOUT)
        printf("There is room to write on fd %d\n", fds.fd);
```

Wait 3.5 seconds:

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

poll((struct pollfd *) NULL, 0, 3500);
```

Wait for a high priority, priority, or normal message on streams file descriptor 0:

```
#include <poll.h>
struct pollfd fds;

fds.fd = 0;
fds.events = POLLIN|POLLPRI;
poll(&fds, 1, -1);
```

SEE ALSO

read(2), write(2), select(2), getmsg(2), putmsg(2), streamio(7).

STANDARDS CONFORMANCE

poll(): AES, SVID2, SVID3

NAME

prealloc - preallocate fast disk storage

SYNOPSIS

```
#include <unistd.h>

int prealloc(int fildes, off_t size);
```

DESCRIPTION

prealloc() is used to preallocate space on a disk for faster storage operations.

fildes is a file descriptor obtained from a **creat()**, **open()**, **dup()**, or **fcntl()** system call for an ordinary file of zero length. It must be opened writable, because 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 allocated in an implementation-dependent fashion for fast sequential reads and writes. The EOF in an extended file is 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** or **tar** (see *cp*(1) and *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.

RETURN VALUE

Upon successful completion, **prealloc()** returns 0; otherwise, it returns −1 and sets **errno** to indicate the error.

ERRORS

prealloc() fails and no disk space is allocated if any of the following conditions are encountered:

[EBADF]	*fildes* is not a valid open file descriptor opened for writing.
[EDQUOT]	User's disk quota block limit has been reached for this file system.
[EFBIG]	*size* exceeds the maximum file size or the process's file size limit. See *ulimit*(2).
[ENOSPC]	Not enough space is left on the device to allocate the requested amount; no space was allocated.
[ENOTEMPTY]	*fildes* not associated with an ordinary file of zero length.

p

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);
```

WARNINGS

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.

AUTHOR

prealloc() was developed by HP.

SEE ALSO

prealloc(1), creat(2), dup(2), fcntl(2), open(2), prealloc64(2), read(2), ulimit(2), write(2).

NAME

profil - execution time profile

SYNOPSIS

```
#include <time.h>

void profil(
     unsigned short int *buff,
     size_t bufsiz,
     size_t offset,
     unsigned int scale
);
```

DESCRIPTION

profil() controls profiling, by which the system maintains estimates of the amount of time the calling program spends executing at various places in its address space.

The *buff* argument must point to an area of memory whose length (in bytes) is given by *bufsiz*. When profiling is on, the process's program counter (pc) is examined each clock tick (**CLK_TCK** times per second), *offset* is subtracted from the pc value, and the result is multiplied by *scale*. If the resulting number corresponds to an element inside the array of **unsigned short int**s to which *buff* points, that element is incremented.

The number of samples per second for a given implementation is given by **CLK_TCK**, which is defined in <**time.h**>.

The scale is interpreted as an unsigned, sixteen bit, fixed-point fraction with binary point at the left: 0177777 (octal) gives a one-to-one 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 one of the **exec()** functions is executed, but remains on in child and parent both after a **fork()**. Profiling is turned off if an update in *buff* would cause a memory fault.

RETURN VALUE

No value is returned.

SEE ALSO

prof(1), monitor(3C).

STANDARDS CONFORMANCE

profil(): SVID2, SVID3, XPG2

NAME

pstat_getstatic(), pstat_getdynamic(), pstat_getproc(), pstat_getlwp(), pstat_getprocvm(), pstat_getprocessor(), pstat_getvminfo(), pstat_getdisk(), pstat_getlv(), pstat_getswap(), pstat_getfile(), pstat_getipc(), pstat_getsem(), pstat_getmsg(), pstat_getshm(), pstat_getstable(), pstat_getcrashinfo(), pstat_getcrashdev(), pstat() - get system information

SYNOPSIS

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

int pstat_getstatic(
     struct pst_static *buf, size_t elemsize, size_t elemcount,
     int index
);

int pstat_getdynamic(
     struct pst_dynamic *buf, size_t elemsize, size_t elemcount,
     int index
);

int pstat_getvminfo(
     struct pst_vminfo *buf, size_t elemsize, size_t elemcount,
     int index
);

int pstat_getipc(
     struct pst_ipcinfo *buf, size_t elemsize, size_t elemcount,
     int index
);

int pstat_getprocessor(
     struct pst_processor *buf, size_t elemsize, size_t elemcount,
     int index
);

int pstat_getproc(
     struct pst_status *buf, size_t elemsize, size_t elemcount,
     int index
);

int pstat_getlwp(
     struct lwp_status *buf, size_t elemsize, size_t elemcount,
     int index, pid_t pid
);

int pstat_getprocvm(
     struct pst_vm_status *buf, size_t elemsize, size_t elemcount,
     int index
);

int pstat_getdisk(
     struct pst_diskinfo *buf, size_t elemsize, size_t elemcount,
     int index
);

int pstat_getlv(
     struct pst_lv *buf, size_t elemsize, size_t elemcount,
     int index
);

int pstat_getswap(
     struct pst_swapinfo *buf, size_t elemsize, size_t elemcount,
     int index
);

int pstat_getsem(
     struct pst_seminfo *buf, size_t elemsize, size_t elemcount,
     int index
```

```
);
int pstat_getmsg(
     struct pst_msginfo *buf, size_t elemsize, size_t elemcount,
     int index
);
int pstat_getshm(
     struct pst_shminfo *buf, size_t elemsize, size_t elemcount,
     int index
);
int pstat_getfile(
     struct pst_fileinfo *buf, size_t elemsize, size_t elemcount,
     int index
);
int pstat_getstable(
     struct pst_stable *buf, size_t elemsize, size_t elemcount,
     int index
);
int pstat_getcrashinfo(
     struct pst_crashinfo *buf, size_t elemsize, size_t elemcount,
     int index
);
int pstat_getcrashdev(
     struct pst_crashdev *buf, size_t elemsize, size_t elemcount,
     int index
);
int pstat(
     int, union pstun, size_t, size_t, int
);
```

Remarks

The underlying function **pstat()** is provided for backward compatibility. Use of the **pstat_get*()** wrapper functions (for example, **pstat_getproc()**) is recommended to avoid the polymorphic typing of the **union pstun** parameter.

p

DESCRIPTION

The **pstat** functions return information about various system contexts. The contents of the various contexts' associated data structures, structs **pst_static**, **pst_dynamic**, **pst_vminfo**, **pst_ipcinfo**, **pst_processor**, **pst_diskinfo**, **pst_swapinfo**, **pst_status**, **pst_vm_status**, **pst_lvinfo**, **pst_seminfo**, **pst_msginfo**, **pst_shminfo**, **pst_fileinfo**, **pst_stable**, **pst_crashinfo**, and **pst_crashdev**, are declared in the header file **<sys/pstat.h>**. The header contains descriptions of the fields of each of the context data structures.

Summary of Available Contexts

The **pstat** routines support the following contexts of information. Detailed descriptions of each routine follow.

Context	Struct	Routine	Instances	Short Cut
Static	**pst_static**	**pstat_getstatic()**	1	
Dynamic	**pst_dynamic**	**pstat_getdynamic()**	1	
VM	**pst_vminfo**	**pstat_getvminfo()**	1	
IPC	**pst_ipcinfo**	**pstat_getipc()**	1	
Stable Store	**pst_stable**	**pstat_getstable()**	1	
Crash Dumps	**pst_crashinfo**	**pstat_getcrashinfo()**	1	
Processor	**pst_processor**	**pstat_getprocessor()**	1 per processor	
Disk	**pst_diskinfo**	**pstat_getdisk()**	1 per disk	
Swap	**pst_swapinfo**	**pstat_getswap()**	1 per swap area	
Dump Areas	**pst_crashdev**	**pstat_getcrashdev()**	1 per dump area	
Process	**pst_status**	**pstat_getproc()**	1 per process	yes
LW Process	**lwp_status**	**pstat_getlwp()**	1 per lwp/thread	yes
Process VM	**pst_vm_status**	**pstat_getprocvm()**	1 per process region	yes
LVM Vol	**pst_lvinfo**	**pstat_getlv()**	1 per lvol	yes
Sema Set	**pst_seminfo**	**pstat_getsem()**	1 per sem set	yes
Msg Queue	**pst_msginfo**	**pstat_getmsg()**	1 per msg queue	yes
Shared Mem	**pst_shminfo**	**pstat_getshm()**	1 per shm seg	yes
Open File	**pst_fileinfo**	**pstat_getfile()**	1 per file	yes

Wrapper Function Descriptions

pstat_getstatic()

Returns static information about the system. This data does not vary while the system is running. There is one global instance of this context. Data, up to a maximum of **elemsize** bytes, are returned in the **struct pst_static** pointed to by **buf**. The **elemcount** parameter must be 1. The **index** parameter must be 0.

pstat_getdynamic()

Returns dynamic information about the system. There is one global instance of this context. Data, up to a maximum of **elemsize** bytes, are returned in the **struct pst_dynamic** pointed to by **buf**. The **elemcount** parameter must be 1. The **index** parameter must be 0.

pstat_getvminfo()

Returns information about the virtual memory subsystem. There is one global instance of this context. Data, up to a maximum of **elemsize** bytes, are returned in the **struct pst_vminfo** pointed to by **buf**. The **elemcount** parameter must be 1. The **index** parameter must be 0.

pstat_getipc()

Returns information about System V IPC subsystem. There is one global instance of this context. This data does not vary while the system is running. Data, up to a maximum of **elemsize** bytes, are returned in the **struct pst_ipcinfo** pointed to by **buf**. The **elemcount** parameter must be 1. The **index** parameter must be 0.

pstat_getcrashinfo()

Returns information about the system's crash dump configuration. Data, up to a maximum of **elemsize** bytes, are returned in the **struct pst_crashinfo** pointed to by **buf**. The **elemcount** parameter must be 1. The **index** parameter must be 0.

pstat_getprocessor()

Returns information specific to a particular processor (the only processor on a uniprocessor system). There is one instance of this context for each processor on the system. For each instance requested, data, up to a maximum of **elemsize** bytes, are returned in the **struct**s **pst_processor** pointed to by **buf**. The **elemcount** parameter specifies the number of **struct**s **pst_processor** that are available at **buf** to be filled in. The **index** parameter specifies the starting index within the context of processors.

pstat_getdisk()

Returns information specific to a particular disk. There is one instance of this context for each disk configured into the system. For each instance requested, data, up to a maximum of **elemsize** bytes, are returned in the **struct**s **pst_diskinfo** pointed to by **buf**. The **elemcount** parameter specifies the number of **struct**s **pst_diskinfo** that are available at **buf** to be filled in. The **index** parameter specifies the starting index within the context of disks.

pstat_getswap()
Returns information specific to a particular swap area. There is one instance of this context for each swap area (block or filesystem) configured into the system. For each instance requested, data, up to a maximum of **elemsize** bytes, are returned in the **struct**s **pst_swapinfo** pointed to by **buf**. The **elemcount** parameter specifies the number of **struct**s **pst_swapinfo** that are available at **buf** to be filled in. The **index** parameter specifies the starting index within the context of swap areas.

pstat_getcrashdev()
Returns information specific to a particular crash dump device. There is one instance of this context for each crash dump device configured on the system. For each instance requested, data, up to a maximum of **elemsize** bytes, are returned in the **struct**s **pst_crashdev** pointed to by **buf**. The **elemcount** parameter specifies the number of **struct**s **pst_crashdev** that are available at **buf** to be filled in. The **index** parameter specifies the starting index within the context of crash dump devices.

pstat_getproc()
Returns information specific to a particular process. There is one instance of this context for each active process on the system. For each instance requested, data, up to a maximum of **elemsize** bytes, are returned in the **struct**s **pst_status** pointed to by **buf**. The **elemcount** parameter specifies the number of **struct**s **pst_status** that are available at **buf** to be filled in. The **index** parameter specifies the starting index within the context of processes. As a shortcut, information for a single process may be obtained by setting **elemcount** to zero and setting **index** to the PID of that process.

pstat_getlwp()
Returns information specific to a particular thread or LWP (Lightweight Process) in a process. There is one instance of this context for each LWP in a process on the system. For each instance requested, data, up to a maximum of **elemsize** bytes, are returned in the **struct lwp_status** pointed to by **buf**. The **elemcount** parameter specifies the number of **struct lwp_status** that are available at **buf** to be filled in. The **index** parameter specifies the starting index within the context of LWPs in a process.

If **pid** is set to -1 and **elemcount** is greater than 0, **elemcount** entries of system LWP information are returned to the caller program.

p

If **pid** is greater than or equal to 0 and **elemcount** is greater than 0, **elemcount** entries of LWP info within the process specified by **pid** are returned.

As a shortcut, information about a single LWP can be obtained by setting **elemcount** to zero and setting **index** to the TID (Thread ID) of that LWP within its process.

pstat_getprocvm()
Returns information specific to a particular process' address space. There is one instance of this context for each process region contained in the process' address space. For each instance requested, data, up to a maximum of **elemsize** bytes, are returned in the **struct pst_vm_status** pointed to by **buf**. Only at most one instance (process region) is returned for each call to **pstat_getprocvm()**. The **elemcount** parameter identifies the process for which address space information is to be returned. An **elemcount** parameter of zero indicates that address space information for the currently executing process should be returned. The **index** parameter specifies the starting index (beginning with 0) within the context of process regions for the indicated process. For example, an **index** of 3 indicates the 4th process region within the indicated process' address space. As a shortcut, information for a specific process (other than the currently executing one) may be obtained by setting **elemcount** to the PID of that process.

pstat_getlv()
Returns information specific to a particular logical volume. There is one instance of this context for each logical volume configured into the system. For each instance requested, data, up to a maximum of **elemsize** bytes, are returned in the **struct**s **pst_lvinfo** pointed to by **buf**. The **elemcount** parameter specifies the number of **struct**s **pst_lvinfo** that are available at **buf** to be filled in. The **index** parameter specifies the starting index within the context of logical volumes. As a shortcut, information for a single logical volume may be obtained by setting **elemcount** to zero and setting **index**

to the dev_t of that logical volume.

pstat_getsem()

Returns information specific to a particular System V semaphore set. There is one instance of this context for each System V semaphore set on the system. For each instance requested, data, up to a maximum of **elemsize** bytes, are returned in the **struct**s **pst_seminfo** pointed to by **buf**. The **elemcount** parameter specifies the number of **struct**s **pst_seminfo** that are available at **buf** to be filled in. The **index** parameter specifies the starting index within the context of System V semaphore sets. As a shortcut, information for a single semaphore set may be obtained by setting **elemcount** to zero and setting **index** to the semid of that semaphore set.

pstat_getmsg()

Returns information specific to a particular System V message queue. There is one instance of this context for each System V message queue on the system. For each instance requested, data, up to a maximum of **elemsize** bytes, are returned in the **struct**s **pst_msginfo** pointed to by **buf**. The **elemcount** parameter specifies the number of **struct**s **pst_msginfo** that are available at **buf** to be filled in. The **index** parameter specifies the starting index within the context of System V message queues. As a shortcut, information for a single message queue may be obtained by setting **elemcount** to zero and setting index to the msqid of that message queue.

pstat_getshm()

Returns information specific to a particular System V shared memory segment. There is one instance of this context for each System V shared memory segment on the system. For each instance requested, data, up to a maximum of **elemsize** bytes, are returned in the **struct**s **pst_shminfo** pointed to by **buf**. The **elemcount** parameter specifies the number of **struct**s **pst_shminfo** that are available at **buf** to be filled in. The **index** parameter specifies the starting index within the context of System V shared memory segments. As a shortcut, information for a single shared memory segment may be obtained by setting **elemcount** to zero and setting **index** to the shmid of that shared memory segment.

pstat_getfile()

Returns information specific to a particular open file for a specified process. For the specified process, there is one instance of this context for each open file descriptor. For each instance requested, data, up to a maximum of **elemsize** bytes, are returned in the **struct**s **pst_fileinfo** pointed to by **buf**. The **elemcount** parameter specifies the number of **struct**s **pst_fileinfo** that are available at **buf** to be filled in. The **index** parameter specifies the starting index within the context of open files for the specified process: it is a 32-bit quantity constructed of the **pst_idx** field of the 'owning' process, obtained via **pstat_getproc()**, described above, as the most significant 16 bits, and the index of open files within the process as the least significant 16 bits. Example:

index = ((pst_idx << 16) | (file_index & 0xffff)); As a shortcut, information for a single file within the specified process may be obtained by setting **elemcount** to zero and setting the least significant 16 bits to the file descriptor number (the most significant 16 bits are still set to the **pst_idx** field from the **pst_status** structure for the process).

The **pst_fileinfo** structure contains both a **psf_offset** and **psf_offset64** element. The **psf_offset** element can correctly store a 32-bit value, whereas the **psf_offset64** element can store a 64-bit value. **pstat_getfile()** will fill in both **psf_offset** and **psf_offset64** if the value can be correctly stored in both elements. If the offset is too large to be correctly stored in **psf_offset** , then **psf_offset** will contain a -1. No error will be set in this case.

pstat_getstable()

Returns information contained in the system's stable storage area. There is one global instance of this context. Data, up to a maximum of **elemsize** bytes, are returned in the **struct pst_stable** pointed to by **buf**. The **elemcount** parameter must be 1. The **index** parameter must be 0.

Notes

A wide (64 bit) version of the **pstat** interfaces are available for narrow (32 bit) applications to use. A narrow application could use the flag **-D_PSTAT64** at compile time to switch to the wide interfaces. Using

p

this compiler flag in a narrow application is equivalent to using the default interfaces on a wide system.

Refer to the **pstat** header file to see how the various structures would look like when the -D_PSTAT64 flag is used.

The **pstat_getlwp**, **pstat_getcrashinfo**, and **pstat_getcrashdev** interfaces are available only in the wide mode and for applications written in standard C and extended ANSI.

RETURN VALUE

Upon successful completion, **pstat()** and the various wrapper routines (for example, **pstat_getprocessor()**) return the number of instances filled in at the address **buf**. Otherwise, a value of −1 is returned and **errno** is set to indicate the error.

ERRORS

The **pstat** functions fail if any of the following conditions are encountered:

[EFAULT] **buf** points to an invalid address.

[ESRCH] For the **pstat_getproc()**, **pstat_getprocvm()**, **pstat_getlv()**, **pstat_getsem()**, **pstat_getmsg()**, **pstat_getshm()** or **pstat_getfile()** calls, **elemcount** was 0, specifying the single-item short-cut, and no item matched the selection criteria in **index** (for example, PID for **pstat_getproc()**).

[EINVAL] For the **pstat_getproc()**, **pstat_getprocvm()**, **pstat_getlv()**, **pstat_getsem()**, **pstat_getmsg()**, **pstat_getshm()** or **pstat_getfile()** calls, **elemcount** was not zero, and **index** was less than zero.

[EINVAL] **elemsize** is less than or equal to zero or **elemsize** is larger than the size of the associated data structure (for example, **elemsize**>**sizeof**(**struct pst_processor**) for the **pstat_getprocessor()** call).

[EINVAL] **elemcount** is not 1 or **index** is not zero for the **pstat_getstatic()**, **pstat_getdynamic()**, **pstat_getvminfo()**, **pstat_getipc()**, **pstat_getstable()**, or **pstat_getcrashinfo()** calls.

[EINVAL] **elemcount** is not greater than or equal to 1 or **index** is not greater than or equal to zero for the **pstat_getprocessor()**, **pstat_getdisk()**, **pstat_getswap()**, or **pstat_getcrashdev()** calls.

[EOVERFLOW] Offset element is too large to store into the structure pointed to by the **buf** argument.

BACKWARD COMPATIBILITY

The specific calling convention of passing the expected data structure size is used in order to allow for future expansion of the interface, while preserving backwards source and object compatibility for programs written using the pstat interfaces. Three rules are followed to allow existing applications to continue to execute from release to release of the operating system.

- New data for a context are added to the end of that context's data structure.
- Old, obsolete data members are **NOT** deleted from the data structure.
- The operating system honors the **elemsize** parameter of the call and only returns the first **elemsize** bytes of the context data, even if the actual data structure has since been enlarged.

In this way, an application which passes its compile-time size of the context's data structure (for example, **sizeof(struct pst_processor)**) for the per-process context) as the **elemsize** parameter will continue to execute on future operating system releases without recompilation, even those that have larger context data structures. If the program is recompiled, it will also continue to execute on that and future releases. Note that the reverse is not true: a program using the **pstat** interfaces compiled on, say, HP-UX release 10.0 will not work on HP-UX release 9.0.

The code examples, below, demonstrate the calling conventions described above.

EXAMPLES

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

p

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

/*
 * Example 1: get static global information
 */
{
     struct pst_static pst;

     if (pstat_getstatic(&pst, sizeof(pst), (size_t)1, 0) != -1)
          (void)printf("page size is %d bytes\n", pst.page_size);
      else
          perror("pstat_getstatic");
}

/*
 * Example 2: get information about all processors, first obtaining
 * number of processor context instances
 */
{
     struct pst_dynamic psd;
     struct pst_processor *psp;

     if (pstat_getdynamic(&psd, sizeof(psd), (size_t)1, 0) != -1) {
          size_t nspu = psd.psd_proc_cnt;
          psp = (struct pst_processor *)malloc(nspu * sizeof(*psp));
          if (pstat_getprocessor(psp, sizeof(*psp), nspu, 0) != -1) {
               int i;
               int total_execs = 0;
               for (i = 0; i < nspu; i++) {
                    int execs = psp[i].psp_sysexec;
                    total_execs += execs;
                    (void)printf("%d exec()s on processor #%d\n",
                                 execs, i);
               }

               (void)printf("total execs for the system were %d\n",
                            total_execs);
          }
          else
               perror("pstat_getdynamic");
     }
     else
          perror("pstat_getdynamic");
}

/*
 * Example 3: get information about all per-process -- 10 at a time
 * done this way since current count of active processes unknown
 */
{
#define BURST ((size_t)10)

     struct pst_status pst[BURST];
     int i, count;
     int idx = 0; /* index within the context */

     /* loop until count == 0, will occur all have been returned */
     while ((count=pstat_getproc(pst, sizeof(pst[0]),BURST,idx))>0) {
          /* got count (max of BURST) this time.  process them */
          for (i = 0; i < count; i++) {
               (void)printf("pid is %d, command is %s\n",
                             pst[i].pst_pid, pst[i].pst_ucomm);
          }
```

p

```
        /*
         * now go back and do it again, using the next index after
         * the current 'burst'
         */
        idx = pst[count-1].pst_idx + 1;
    }

    if (count == -1)
        perror("pstat_getproc()");

#undef BURST
}

/*
 * Example 4: Get a particular process' information
 */
{
    struct pst_status pst;
    int target = (int)getppid();

    if (pstat_getproc(&pst, sizeof(pst), (size_t)0, target) != -1)
        (void)printf("Parent started at %s", ctime(&pst.pst_start));
    else
        perror("pstat_getproc");
}

/*
 * Example 5: get information about all shared memory segments
 */
{
    struct pst_ipcinfo psi;
    struct pst_shminfo *pss;

    if (pstat_getipc(&psi, sizeof(psi), (size_t)1, 0) != -1) {
        size_t num_shm = psi.psi_shmmni;
        pss = (struct pst_shminfo *)malloc(num_shm * sizeof(*pss));
        if (pstat_getshm(pss, sizeof(*pss), num_shm, 0) != -1) {
            int i;
            (void)printf("owner\tkey\tsize\n");
            for (i = 0; i < num_shm; i++) {
                /* skip inactive segments */
                if (!(pss[i].psh_flags & PS_SHM_ALLOC))
                    continue;
                (void)printf("%d\t%#x\t%d\n",
                             pss[i].psh_uid, pss[i].psh_key,
                             pss[i].psh_segsz);
            }
        }
        else
            perror("pstat_getshm");
    }
    else
        perror("pstat_getipc");
}

/*
 * Example 6: List all the open files for a process
 */
{
    struct pst_status pst;
    int target = (int)getppid();

    /*
     * First get the desired process to get its 'index'.
     * This will be used when retrieving the file data.
     */
```

p

```
     if (pstat_getproc(&pst, sizeof(pst), (size_t)0, target) != -1) {
          int pidx = pst.pst_idx;
#define BURST ((size_t)10)
          struct pst_fileinfo psf[BURST];
          int i, count;
          int idx = 0; /* index within the context */

          (void)printf("Open files for process PID %d\n", pst.pst_pid);

          /*
           * Construct the index into the per-process file context:
           * Most significant 16 bits are the process' index (above).
           * Least significant 16 bits are the file's index.
           * For a given process, the file index starts at 0.
           */
          idx = (pidx << 16) | (0 & 0xffff);

          /* loop until all fetched */
          while (count = pstat_getfile(psf, sizeof(psf[0]),
                 BURST, idx) > 0) {
               /* process them (max of BURST) at a time */
               for (i = 0; i < count; i++) {
                    (void)printf("fd #%x\tFSid %x:%x\tfileid %d\n",
                           psf[i].psf_fd,
                           psf[i].psf_id.psf_fsid.psfs_id,
                           psf[i].psf_id.psf_fsid.psfs_type,
                           psf[i].psf_id.psf_fileid);
               }

               /*
                * Now go back and do it again, using the
                * next index after the current 'burst'
                */
                idx = psf[count-1].psf_idx + 1;
          }
          if (count == -1)
               perror("pstat_getfile()");

#undef BURST
     }
     else
          perror("pstat_getproc");
}

/*
 * Example 7: Acquire information about a specific LWP
 */
{
     struct lwp_status lwpbuf;

     /*
      * get information for LWP whose lwpid is 121 within
      * a process whose pid is 1234.
      */

     count = pstat_getlwp(buf, sizeof(struct lwp_status),
          0, 4321, 1234)

     if (count == -1)
          perror("pstat_getlwp()");
     else
          ...
}
```

p

WARNINGS

Some parts of the program status may not get updated when a process becomes a zombie. An example is that the cpu percentage is not updated because the process is not expected to be scheduled to run after entering the zombie state.

AUTHOR

The `pstat` routines were developed by HP.

FILES

`/usr/include/sys/pstat.h`
Contains detailed descriptions of context data structures and fields.

SEE ALSO

ps(1), top(1), vmstat(1), iostat(1), fuser(1), vgdisplay(1), lvdisplay(1), crashconf(1M), stat(2), sysconf(2), semctl(2), msgctl(2), shmctl(2), crashconf(2), fileno(3S).

NAME

ptrace() - process trace

SYNOPSIS

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

int ptrace(
     int request,
     pid_t pid,
     int addr,
     int data,
     int addr2
);
```

Remarks

Much of the functionality of **ptrace()** 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

The **ptrace()** system call provides a means by which a process can 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()** (see *wait*(2)).

A traced process may also enter the stopped state without encountering a signal. This can happen if the traced process stops in response to specific *events* that it encounters during the course of its execution. To make this happen, the tracing process has to set specific event flags in the context of the traced process. This mechanism will be described later in greater detail.

When the traced process is in the stopped state, the tracing process can use **ptrace()** to examine and modify the "core image". Also, the tracing process can cause the traced process to either terminate or continue, with the possibility of ignoring the signal that caused it to stop.

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

The *request* argument determines the precise action to be taken by **ptrace()**. It is one of the values described in the rest of this section.

The following request is used by the child process that will be traced.

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, which 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 occur 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. The responsibility of ensuring that the traced process is in a stopped state before a request is issued, lies with the tracing process.

PT_RDUSER
PT_RIUSER 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 produces equivalent 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.

p

This request fails if *addr* is not the start address of a word or is outside the user area, in which case a value of −1 is returned to the tracing process and its **errno** is set to [EIO].

PT_WDUSER
PT_WIUSER With these requests, the value given by the *data* argument is written into the address space of the traced process at location *addr*. **PT_WIUSER** writes a word into I space, and **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 This request is not supported. Therefore, it returns −1, sets **errno** to [EIO] and does not affect the user area of the traced process.

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_RDDATA
PT_RDTEXT These requests are identical to **PT_RDUSER** and **PT_RIUSER**, 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_WRDATA
PT_WRTEXT These requests are identical to **PT_WDUSER** and **PT_WIUSER**, 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.

p

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 *addr2* argument is ignored.

If the *addr* argument 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.

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()**. 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. It 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 user ID of process *pid* unless the effective user ID of the tracing process is superuser. The calling process can use the **wait()** 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**.

If the *addr* argument is not 1, the Instruction Address Offset Queue (program counter) is loaded with the values *addr* and *addr2*.

PT_CONTIN1 This request causes the traced process to resume execution with all its pending signals intact. If the *data* argument is 0, the signal that caused the traced process to stop is canceled before the traced process resumes execution. If the *data* argument is a valid signal number, the traced process resumes execution as if it had received that signal. 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_SINGLE1 This request causes a flag to be set so that an interrupt occurs upon the completion of one machine instruction. It then executes the same steps as listed above for request **PT_CONTIN1**. 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.

As noted earlier, a tracing process can set event flags in the context of the traced process to make it respond to specific events, during its execution. These events are:

PTRACE_SIGNAL

This event flag indicates that, when processing signals, the traced process needs to examine signal mask bits set in its context by the tracing process. See the *ptrace_event* structure description under **PT_SET_EVENT_MASK** for further details.

If the signal being processed has its signal mask bit set, signal processing continues as though the process were not traced. The traced process is not stopped and the tracing process is not notified of the signal. If the signal mask bit is not set for the signal being processed, the traced process is stopped and the tracing process is notified via **wait()** (see *wait*(2)).

Note that the **SIGKILL** signal is an exception to this rule in that it can never be unmasked; that is, it behaves as though its mask bit were always set, regardless of whether or not its mask bit is in fact set. Consequently, a **SIGKILL** signal cannot be used to stop a traced process.

In this respect, a **SIGTRAP** signal is also special in that it is specifically used to stop traced processes. A **SIGTRAP** signal should therefore never be masked. Setting a mask bit for **SIGTRAP** will result in unexpected system behavior.

PTRACE_FORK

This event flag indicates that the traced process needs to take special action when it invokes **fork().** When set, both the parent and child processes stop (the child after marking itself as a traced process and adopting its parent's debugger). Both processes log the fact that they stopped in response to a **PTRACE_FORK** event. Further, the child's *pid* is logged in the parent's context, and the parent's *pid* is logged in the child's context. The child does not inherit its parent's event flags. See the *ptrace_state* structure description under **PT_GET_PROCESS_STATE** for further details.

PTRACE_VFORK

This event flag indicates that the traced process needs to take special action when it invokes **vfork()**. When set, the child process stops after marking itself as a traced process and adopting its parent's debugger. The fact that a **PTRACE_VFORK** event was responded to is logged in the context of both the parent and child processes. Further, the child's *pid* is logged in the parent's context, and the parent's *pid* is logged in the child's context. The child does not inherit its parent's event flags. See the *ptrace_state* structure description under **PT_GET_PROCESS_STATE** for further

p

details. It is important to note that the warnings with respect to **vfork()** (see *vfork*(2)), continue to apply here. In particular, it needs to be remembered that, when the child process stops, its parent process is suspended, and 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()** or abnormally (see *exec*(2) and *exit*(2))).

PTRACE_EXEC

This event flag indicates that the traced process needs to take special action when it invokes **exec*()**. When set, the traced process stops after logging the fact that it stopped in response to a **PTRACE_EXEC** event. It also logs information pertaining to the path or file argument of **exec*()**. This includes a pointer to the path name string and the length of the path name string. See the *ptrace_state* structure description under **PT_GET_PROCESS_STATE** for further details.

PTRACE_EXIT

This event flag indicates that the traced process needs to take special action when it invokes **exit()**. When set, the traced process stops after logging the fact that it stopped in response to a **PTRACE_EXIT** event.

PT_SET_EVENT_MASK

This request is used by the calling process to specify event flags and signal mask values that it wants the traced process to respond to. It does so by writing the contents of the *ptrace_event* data structure in the user space pointed to by *addr* into the context of the traced process. The *data* argument specifies the number of bytes to be transferred. The *addr2* argument is ignored.

The request fails if the number of bytes specified is less than zero or greater than the size of the *ptrace_event* structure, and its **errno** is set to [EIO].

```
typedef struct ptrace_event{
     sigset_t  pe_signals;
     events_t  pe_set_event;
} ptrace_event_t;
```

Event flags are set in the *pe_set_event* member of the *ptrace_event* data structure. An event flag is set when the tracing process wants the traced process to respond to a particular event. As detailed earlier, the event flags defined are **PTRACE_EXEC**, **PTRACE_EXIT**, **PTRACE_FORK**, **PTRACE_SIGNAL**, and **PTRACE_VFORK**. See the definition of *events_t* in **<sys/ptrace.h>** for more details.

Signal mask values are set in the *pe_signals* member of the *ptrace_event* structure. This field is qualified by a **PTRACE_SIGNAL** event flag being set in the *pe_set_event* member. Mask values set in the *pe_signals* member correspond to signals that need to be masked from the tracing process when received by the traced process; that is, these are signals received by the traced process that the tracing process does not want to be informed about. The *pe_signals* member is described by the type definition *sigset_t*, which is defined in **<signal.h>**.

PT_GET_EVENT_MASK

This request is used by the calling process to determine the event flags and signal mask values that have been set in the traced process's context by the last **PT_SET_EVENT_MASK** request. The *data* argument specifies the number bytes to be read from the context of the traced process into the *ptrace_event* data structure in user space pointed to by *addr* . The *addr2* argument is ignored.

The request fails if the number of bytes requested is less than zero or greater than the size of the *ptrace_event* structure, and its **errno** is set to [EIO].

PT_GET_PROCESS_STATE

This request is used by the calling process to access state information logged by the traced process after it (the traced process) has responded to an event. The request reads *data* bytes of data from the traced process's context into the *ptrace_state* data structure in user space pointed to by *addr* . The *addr2* argument is ignored.

The *ptrace_state* data structure is described in **<sys/ptrace.h>** and has the following members:

```
typedef struct ptrace_state{
     events_t  pe_report_event;
     int       pe_path_len;
     pid_t     pe_other_pid;
} ptrace_state_t;
```

The event that the traced process responded to and stopped is logged in the *pe_report_event* member. One of **PTRACE_EXEC**, **PTRACE_EXIT**, **PTRACE_FORK**, **PTRACE_SIGNAL**, or **PTRACE_VFORK** is logged here. See the definition of *events_t* in **<sys/ptrace.h>** for more details.

If the event that the traced process responded to was **PTRACE_EXEC**, then the *pe_path_len* member provides the length of the path name string (which is the path name of the executable file) not including the null terminating character.

If the event that the traced process responded to was **PTRACE_FORK** or **PTRACE_VFORK**, then the *pe_other_pid* member provides the parent's *pid* when accessed from the child's context, and the child's *pid* when accessed from the parent's context.

The request fails if the number of bytes requested is less than zero or greater than the size of the *ptrace_event* structure and its **errno** is set to [EIO].

PT_GET_PROCESS_PATHNAME

If the event that the traced process responded to and stopped was **PTRACE_EXEC**, then this request is used by the calling process to access the path name of the executable file provided as a *path* or *file* argument to **exec*()**. The request reads *data* bytes of data of the path name string from the traced process's context into the data buffer in user space pointed to by *addr* . The *addr2* argument is ignored. In the typical case, *data* is equal to the value of the *pe_path_len* member of the *ptrace_state* structure returned via the **PT_GET_PROCESS_STATE** request.

If the number of bytes requested is greater than zero but less than the length of the path name string, then the number of bytes requested is returned. If the number of bytes requested is greater than the length of the path name string, then the full path name string (including the null terminating character) is returned.

The request fails if the number of bytes requested is less than zero, and its **errno** is set to [EIO].

p

EXAMPLES

The following example illustrates the use of some of the **ptrace()** requests by a tracing process.

```
#include <stdio.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/ptrace.h>
#define BUFSIZ  1024
#define MAXPATH 1024

pid_t           npid, cpid, pid;
int             status, errors=0, pathlength;
ptrace_event_t  *event_addr;
ptrace_state_t  *state_addr;
char            *buf_addr;
size_t          event_len, state_len;
int             filed[2];

child()
{
    int n, bar;

    close(filed[1]);
    /* Wait for parent to write to pipe */
    while ((n = read(filed[0], &bar, BUFSIZ)) == 0);

    /* Now the child can exec. */
    if (execlp("ls", "ls", (char *)0) < 0)   /* error during exec */
```

```
                printf("Child: exec failed\n");
        exit(0);
}
parent()
{
        close(filed[0]);
        /* Before child does an exec, attach it and set its event flag. */
        if (ptrace(PT_ATTACH,pid))  /* failed to attach process */
            printf("Parent: Failed to attach child\n");
        if (pid != wait(&status))   /* wait failed */
            printf("Parent: attach failed with wrong wait status\n");
        if (!WIFSTOPPED(status) || (WSTOPSIG(status) != SIGTRAP))
            printf("Parent: SIGTRAP didn't stop child\n");
        /*
         * The child process has now stopped.  Set its event flag indicating
         * that it needs to trigger on a PTRACE_EXEC event.
         */
        event_addr->pe_set_event = PTRACE_EXEC;
        if (ptrace(PT_SET_EVENT_MASK, pid, event_addr, event_len))
            printf("Parent: PT_SET_EVENT_MASK ptrace request failed\n");
        if (pid != wait(&status))   /* wait failed */
            printf("Parent: wait() failed with wrong wait status\n");
        /*
         * Send the child a message so it can break out of the while loop.
         * Get it running so it can exec.
         */
        write(filed[1], "now run", 7);
        if (ptrace(PT_CONTIN, pid, 1, 0) != 0)
            printf("Parent: failed to get child process running\n");
        /*
         * Wait for the traced child to stop after the exec system call in
         * response to an exec event set in its ptrace_event structure.
         */
        if (pid != (npid = wait(&status)))   /* wait failed */
            printf("Parent: wait() failed with wrong status\n");
        if (!WIFSTOPPED(status))
            printf("Parent: invalid wait() completion\n");
        /*
         * Child has stopped; fetch its process state and examine state
         * information.
         */
        if (ptrace(PT_GET_PROCESS_STATE, pid, state_addr, state_len) < 0)
            printf("Parent: PT_GET_PROCESS_STATE ptrace request failed\n");
        if (pid != wait(&status))   /* wait failed */
            printf("Parent: wait() failed with wrong wait status\n");
        /* Check if the pathlength value returned is non-zero */
        if ((pathlength = state_addr->pe_path_len) == 0)
            printf("Parent: zero length pathname returned\n");
        /* Fetch exec'd file pathname and store it in the buffer. */
        if (ptrace(PT_GET_PROCESS_PATHNAME, pid, buf_addr, (pathlength+1))
            < 0){
            printf("Parent: Failed to get exec pathname\n");
        } else {
            printf("Parent: the exec pathname is %s\n", buf_addr);
            if (pid != wait(&status))   /* wait failed */
                printf("Parent: wait() failed with wrong status\n");
        }
}
```

p

```
main()
{
    event_len = sizeof(ptrace_event_t);
    state_len = sizeof(ptrace_state_t);
    event_addr = calloc(event_len, 1);
    state_addr = calloc(state_len, 1);
    buf_addr = calloc(MAXPATH, 1);
    pipe(filed);
    switch (pid = fork()) {
        case -1:
            exit(1);
        case 0:
            child();
            break;
        default:
            parent();
            break;
    }
}
```

ERRORS

If **ptrace()** fails, **errno** is set to one of the following values.

[EACCES] The executable image of the process being attached resides across an interruptible NFS mount.

[EIO] *request* is an illegal number.

[EIO] The **PT_SETTRC** request is used with a *data* argument that is less than zero or not a multiple of four, or *data* is not word-aligned.

[EIO] Attempting to write to a memory segment of the traced process that is not writeable, or attempting to write to page 0, or the *request* argument is out of range.

[EIO] The **PT_CONTIN** request is being used with an invalid *data* argument (signal number).

[EIO] Attempting to write to the user area via the **PT_WUAREA** request.

[EPERM] The specified process cannot be attached for tracing.

[EPERM] The process *pid* is already being traced or *pid* refers to the calling process itself.

[ESRCH] *pid* identifies a process to be traced that does not exist or has not executed a **ptrace()** with request **PT_SETTRC**.

SEE ALSO

adb(1), exec(2), exit(2), signal(2), wait(2).

STANDARDS CONFORMANCE

ptrace(): SVID2, SVID3, XPG2

NAME

putmsg, putpmsg - send a message on a stream

SYNOPSIS

```
#include <stropts.h>

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

int putpmsg(
     int fildes,
     struct strbuf *ctlptr,
     struct strbuf *dataptr,
     int band,
     int flags
);
```

DESCRIPTION

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

The **putpmsg()** function does the same things as **putmsg()**, but the process can send messages in different priority bands. Except where noted, all requirements on **putmsg()** also pertain to **putpmsg()**.

The *fildes* argument specifies a file descriptor referencing an open stream. The *ctlptr* and *dataptr* arguments each point to a **strbuf** structure.

The *ctlptr* argument points to the structure describing the control part, if any, to be included in the message. The *buf* member in the **strbuf** structure points to the buffer where the control information resides, and the *len* member indicates the number of bytes sent. The *maxlen* member is not used by **putmsg()**. In a similar manner, the argument *dataptr* specifies the data, if any, to be included in the message. The *flags* argument indicates what type of message should be sent and is described further below.

p

To send the data part of a message, *dataptr* must not be a null pointer and the *len* member of *dataptr* must be 0 or greater. To send the control part of a message, the corresponding values must be set for *ctlptr*. No data (control) part will be sent if either *dataptr* (*ctlptr*) is a null pointer or the *len* member of *dataptr* (*ctlptr*) is set to −1.

For **putmsg()**, if a control part is specified and *flags* is set to **RS_HIPRI**, a high priority message is sent. If no control part is specified, and *flags* is set to **RS_HIPRI**, **putmsg()** fails and sets **errno** to [EINVAL]. If *flags* is set to 0, a normal message (priority band equal to 0) is sent. If a control part and data part are not specified and *flags* is set to 0, no message is sent and 0 is returned.

The stream head guarantees that the control part of a message generated by **putmsg()** is at least 64 bytes in length.

For **putpmsg()**, the flags are different. The *flags* argument is a bitmask with the following mutually-exclusive flags defined: **MSG_HIPRI** and **MSG_BAND**. If *flags* is set to 0, **putpmsg()** fails and sets **errno** to [EINVAL]. If a control part is specified and *flags* is set to **MSG_HIPRI** and *band* is set to 0, a high-priority message is sent. If *flags* is set to **MSG_HIPRI** and either no control part is specified or *band* is set to a non-zero value, **putpmsg()** fails and set **errno** to [EINVAL]. If *flags* is set to **MSG_BAND**, then a message is sent in the priority band specified by *band*. If a control part and data part are not specified and *flags* is set to **MSG_BAND**, no message is sent and 0 is returned.

The **putmsg()** function blocks if the stream write queue is full due to internal flow control conditions. For high-priority messages, **putmsg()** does not block on this condition. For other messages, **putmsg()** does not block when the write queue is full and **O_NONBLOCK** is set.

The **putmsg()** function also blocks, unless prevented by lack of internal resources, while for the availability of message blocks in the stream, regardless of priority of whether **O_NONBLOCK** has been specified. No partial message is sent.

MULTITHREAD USAGE

The **putmsg()** and **putpmsg()** functions are safe to be called by multithreaded applications, and they are thread-safe for both POSIX Threads and DCE User Threads. The **putmsg()** and **putpmsg()** functions have cancellation points. They are async-signal safe and fork-safe. They are not async-cancel safe.

RETURN VALUE

Upon successful completion, **putmsg()** and **putpmsg()** return 0. Otherwise, they return -1 and set **errno** to indicate the error.

ERRORS

[EAGAIN] A non-priority message was specified, the **O_NONBLOCK** flag is set, and the stream write queue is full due to internal flow control conditions, or buffers could not be allocated for the message that was to be created.

[EBADF] *fildes* is not a valid file descriptor open for writing.

[EINTR] A signal was caught during **putmsg()** or **putpmsg()**.

[EINVAL] An undefined value is specified in *flags*, or *flags* is set to **RS_HIPRI** or **MSG_HIPRI** and no control part is supplied, or the stream or multiplexor referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexor, or *flags* is set to **MSG_HIPRI** and *band* is non-zero (for **putpmsg()** only).

[ENOSTR] A stream is not associated with *fildes*.

[ENXIO] A hangup condition was generated downstream for the specified stream.

[EPIPE] or [EIO]
The *fildes* argument refers to a STREAMS-based pipe and the other end of the pipe is closed. A **SIGPIPE** signal is generated for the calling process.

[ERANGE] The size of the data part of the message does not fall within the range specified by the maximum and minimum packet sizes of the topmost STREAMS module. This value is also returned if the control part of the message is larger than the maximum configured size of the control part of a message, or if the data part of the message is larger than the maximum configured size of the data part of a message.

In addition, **putmsg()** and **putpmsg()** will fail if the stream head had processed an asynchronous error before the call. In this case, the value of **errno** does not reflect the result of **putmsg()** or **putpmsg()** but reflects the prior error.

p

SEE ALSO

getmsg(2), poll(2), read(2), write(2), <stropts.h>, streamio(7).

NAME

quotactl - manipulate disk quotas

SYNOPSIS

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

int quotactl(int cmd, const char *special, uid_t uid, void *addr);
```

DESCRIPTION

quotactl() manipulates disk quotas. *cmd* indicates a command to be applied to the user ID *uid*. Parameter *special* is a pointer to a null-terminated string containing the path name of the block special device for the file system being manipulated. The block special device must be mounted. The parameter *addr* is the address of an optional, command-specific, data structure which is copied in or out of the system. The interpretation of *addr* is explained with each command below:

Q_QUOTAON Turn on quotas for a file system. The parameter *addr* points to the path name of file containing the quotas for the file system. The quota file must exist; it is normally created using the **quotacheck** command (see *quotacheck*(1M)). The *uid* parameter is ignored. This call is restricted to users having appropriate privileges.

Q_QUOTAOFF Turn off quotas for a file system. The *addr* and *uid* parameters are ignored. This call is restricted to the user with appropriate privileges.

Q_GETQUOTA Get disk quota limits and current usage for user *uid*. *addr* is a pointer to a **dqblk** structure (defined in **<sys/quota.h>**). Only users having appropriate privileges can get the quotas of a user other than himself.

Q_SETQUOTA Set disk quota limits and current usage of files and blocks for user *uid*. Note **vxfs** does not allow the current usage fields to be changed *addr* is a pointer to a **dqblk** structure (defined in **<sys/quota.h>**). This call is restricted to users with appropriate privileges.

Q_SETQLIM Set disk quota limits for user *uid*. The parameter *addr* is a pointer to a **dqblk** structure (defined in **<sys/quota.h>**). This call is restricted to users with appropriate privileges.

Q_SYNC Update the on-disk copy of quota usages for a file system. If *special* is null, all file systems with active quotas are synced. The parameters *addr* and *uid* are ignored.

RETURN VALUE

Upon successful completion, **quotactl()** returns 0; otherwise, it returns –1 and sets **errno** to indicate the error.

ERRORS

quotactl() fails when any of the following occurs:

[ENOSYS] The kernel has not been configured with the disk quota subsystem.

[EINVAL] The parameters *cmd* and/or *uid* are invalid.

[ESRCH] No disc quota is found for the indicated user or quotas have not been turned on for this file system.

[EPERM] The call is privileged and the calling process does not have appropriate privileges.

[ENODEV] The parameter *special* contains a type of file system that does not support quotas. Currently, quotas are supported on HFS and VxFS file systems.

[ENOTBLK] The parameter *special* is not a block device.

[EACCES] (**Q_QUOTAON**) The quota file pointed to by *addr* exists but is either not a regular file or is not on the file system pointed to by *special*.

[EBUSY] **Q_QUOTAON** attempted while another **Q_QUOTAON** or **Q_QUOTAOFF** is in progress.

[ENOENT] The file specified by *special* or *addr* does not exist.

[EFAULT] The *addr* or *special* parameter points to an invalid address. Reliable detection of this error is implementation-dependent.

[EDQUOT] User's disk quota block limit has been reached for this file system.

WARNINGS

The **quotactl()** system call is incompatible with the 4.2/4.3BSD implementation of Melbourne quotas which uses a different system call interface and on-disk data structure.

AUTHOR

quotactl() was developed by HP and Sun Microsystems, Inc.

SEE ALSO

quota(1), edquota(1M), rquotad(1M), quotacheck(1M), quotaon(1M), mount(2), quota(5), quota(5).

NAME

read, readv - read from file

SYNOPSIS

```
#include <unistd.h>
ssize_t read(int fildes, void *buf, size_t nbyte);
#include <sys/uio.h>
ssize_t readv(int fildes, const struct iovec *iov, int iovcnt);
```

DESCRIPTION

The **read()** function attempts to read *nbyte* bytes from the file associated with the open file descriptor, *fildes*, into the buffer pointed to by *buf*.

If *nbyte* is 0, **read()** will return 0 and have no other results.

On files that support seeking (for example, a regular file), the **read()** starts at a position in the file given by the file offset associated with *fildes*. The file offset is incremented by the number of bytes actually read.

Files that do not support seeking, for example, terminals, always read from the current position. The value of a file offset associated with such a file is undefined.

No data transfer will occur past the current end- of-file. If the starting position is at or after the end-of-file, 0 will be returned. If the file refers to a device special file, the result of subsequent **read()** requests is implementation-dependent.

If the value of *nbyte* is greater than **{SSIZE_MAX}** the result is implementation-dependent.

When attempting to read from an empty pipe or FIFO:

- If no process has the pipe open for writing, **read()** will return 0 to indicate end-of-file.
- If some process has the pipe open for writing and **O_NONBLOCK** is set, **read()** will return −1 and set **errno** to **EAGAIN**.
- If some process has the pipe open for writing and **O_NONBLOCK** is clear, **read()** will block until some data is written or the pipe is closed by all processes that had the pipe open for writing.

When attempting to read a file (other than a pipe or FIFO) that supports non-blocking reads and has no data currently available:

- If **O_NONBLOCK** is set, **read()** will return a −1 and set **errno** to **EAGAIN**.
- If **O_NONBLOCK** is clear, **read()** will block until some data becomes available.
- The use of the **O_NONBLOCK** flag has no effect if there is some data available.

r

The **read()** function reads data previously written to a file. If any portion of a regular file prior to the end-of-file has not been written, **read()** returns bytes with value 0. For example, **lseek()** allows the file offset to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads in the gap between the previous end of data and the newly written data will return bytes with value 0 until data is written into the gap.

Upon successful completion, where *nbyte* is greater than 0, **read()** will mark for update the *st_atime* field of the file, and return the number of bytes read. This number will never be greater than *nbyte*. The value returned may be less than *nbyte* if the number of bytes left in the file is less than *nbyte*, if the **read()** request was interrupted by a signal, or if the file is a pipe or FIFO or special file and has fewer than *nbyte* bytes immediately available for reading. For example, a **read()** from a file associated with a terminal may return one typed line of data.

If a **read()** is interrupted by a signal before it reads any data, it will return −1 with **errno** set to **[EINTR]**.

If a **read()** is interrupted by a signal after it has successfully read some data, it will return the number of bytes read.

A **read()** from a **STREAMS** file can read data in three different modes: byte-stream mode, message-ondiscard mode, and message-discard mode. The default is byte-stream mode. This can be changed using the **I_SRDOPT ioctl()** request, and can be tested with the **I_GRDOPT ioctl()**. In byte-stream mode, **read()** retrieves data from the **STREAM** until as many bytes as were requested are transferred, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In **STREAMS** message-nondiscard mode, **read()** retrieves data until as many bytes as were requested are transferred, or until a message boundary is reached. If **read()** does not retrieve all the data in a message, the remaining data is left on the **STREAM ,** and can be retrieved by the next **read()** call. Message-discard mode also retrieves data until as many bytes as were requested are transferred, or a message boundary is reached. However, unread data remaining in a message after the **read()** returns is discarded, and is not available for a subsequent **read()**, **readv()**, or **getmsg()** call.

How **read()** handles zero-byte **STREAMS** messages is determined by the current read mode setting. In byte-stream mode, **read()** accepts data until it has read *nbyte* bytes, or until there is no more data to read, or until a zero-byte message block is encountered. The **read()** function then returns the number of bytes read, and places the zero-byte message back on the **STREAM** to be retrieved by the next **read()**, **readv()**, or **getmsg()**. In message-nondiscard mode or message-discard mode, a zero-byte message returns 0 and the message is removed from the **STREAM**. When a zero-byte message is read as the first message on a **STREAM**, the message is removed from the **STREAM** and 0 is returned, regardless of the read mode.

A **read()** from a **STREAMS** file returns the data in the message at the front of the **STREAM** head read queue, regardless of the priority band of the message.

By default, **STREAM**s are in control-normal mode, in which a **read()** from a **STREAMS** file can only process messages that contain a data part but do not contain a control part. The **read()** fails if a message containing a control part is encountered at the **STREAM** head. This default action can be changed by placing the **STREAM** in either control-data mode or control-discard mode with the **I_SRDOPT ioctl()** command. In control-data mode, **read()** converts any control part to data and passes it to the application before passing any data part originally present in the same message. In control-discard mode, **read()** discards message control parts but returns to the process any data part in the message.

In addition, **read()** and **readv()** will fail if the **STREAM** head had processed an asynchronous error before the call. In this case, the value of **errno** does not reflect the result of **read()** or **readv()** but reflects the prior error. If a hangup occurs on the **STREAM** being read, **read()** continues to operate normally until the **STREAM** head read queue is empty. Thereafter, it returns 0.

The **readv()** function is equivalent to **read()**, but places the input data into the *iovcnt* buffers specified by the members of the *iov* array: *iov*[*0*], *iov*[*1*], **...**, *iov*[*iovcnt*−1]. The *iovcnt* argument is valid if greater than 0 and less than or equal to **{IOV_MAX}**.

Each *iovec* entry specifies the base address and length of an area in memory where data should be placed. The **readv()** function always fills an area completely before proceeding to the next.

Upon successful completion, **readv()** marks for update the *st_atime* field of the file.

RETURN VALUE

Upon successful completion, **read()** and **readv()** return a non-negative integer indicating the number of bytes actually read. Otherwise, the functions return −1 and set **errno** to indicate the error.

r

ERRORS

The **read()** and **readv()** functions will fail if:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor and the process would be delayed in **read()** or **readv()**.

[EBADF] The *fildes* argument is not a valid file descriptor open for reading.

[EBADMSG] The file is a **STREAM** file that is set to control-normal mode and the message waiting to be read includes a control part.

[EINTR] The read operation was terminated due to the receipt of a signal, and no data was transferred.

[EINVAL] The **STREAM** or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.

[EIO] A physical I/O error has occurred.

[EIO] The process is a member of a background process attempting to read from its controlling terminal, the process is ignoring or blocking the **SIGTTIN** signal or the process group is orphaned. This error may also be generated for implementation-dependent reasons.

[EISDIR] The *fildes* argument refers to a directory and the implementation does not allow the directory to be read using **read()** or **readv()**. The **readdir()** function should be used instead.

The **readv()** function will fail if:

[EINVAL] The sum of the *iov_len* values in the *iov* array overflowed an *ssize_t*.

The **read()** and **readv()** functions may fail if:

[ENXIO] A request was made of a non-existent device, or the request was outside the capabilities of the device.

The **readv()** function may fail if:

[EINVAL] The **iovcnt** argument was less than or equal to 0, or greater than **{IOV_MAX}**.

SEE ALSO

fcntl(2), ioctl(2), lseek(2), open(2), pipe(2), <stropts.h>, <sys/uio.h>, <unistd.h>, XBD Specification, Chapter 9, General Terminal Interface.

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 4

The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- The type of the argument *buf* is changed from *char* * ®.IR void* , and the type of the argument *nbyte* is changed from unsigned to *size_t*.
- The DESCRIPTION section now states that the result is implementation-dependent if *nbyte* is greater than **{SSIZE_MAX}**. This limit was defined by the constant **{INT_MAX}** in Issue 3.

The following change is incorporated for alignment with the FIPS requirements:

- The last paragraph of the DESCRIPTION section now states that if **read()** is interrupted by a signal after it has successfully read some data, it will return the number of bytes read. In Issue 3 it was optional whether **read()** returned the number of bytes read, or whether it returned −1 with **errno** set to **EINTR**.

Other changes are incorporated as follows:

- The header **<unistd.h>** is added to the SYNOPSIS section.
- The DESCRIPTION section is rearranged for clarity and to align more closely with the ISO POSIX-1 standard. No functional changes are made other than as noted elsewhere in this CHANGE HISTORY section.
- In the ERRORS section in previous issues, generation of the **EIO** error depended on whether or not an implementation supported Job Control. This functionality is now defined as mandatory.
- The **ENXIO** error is marked as an extension.
- The APPLICATION USAGE section is removed.
- The description of **EINTR** is amended.

r

Issue 4, Version 2

The following changes are incorporated for X/OPEN UNIX conformance:

- The **readv()** function is added to the SYNOPSIS.
- The DESCRIPTION is updated to describe the reading of data from **STREAMS** files. An operational description of the **readv()** function is also added.
- References to the **readv()** function are added to the RETURN VALUE and ERRORS sections in appropriate places.
- The ERRORS section has been restructured to describe errors that apply generally (that is, to both **read()** and **readv()**), and to describe those that apply to **readv()** specifically. The **EBADMSG**, **EINVAL**, and **EISDIR** errors are also added.

HP-UX EXTENSIONS

DESCRIPTION

For **readv()**, the **iovec** structure is defined in **/usr/include/sys/uio.h**.

For ordinary files, if the **O_RSYNC**|**O_DSYNC** file status flag is set, the calling process blocks until the data being read and all file attributes required to retrieve the data are the same as their image on disk. Writes pending on the data to be read are executed before returning to the calling process. If the **O_RSYNC**|**O_SYNC** file status flag is set, the behavior is identical to that for **O_RSYNC**|**O_DSYNC** with this addition: all file attributes changed by the read operation (including access time, modification time and status change time) must also be the same as their image on disk. For block special files, if either the **O_RSYNC**|**O_DSYNC** or **O_RSYNC**|**O_SYNC** status flag is set, the calling process blocks until the data being read is an image of the data on the disk. Writes pending on the data to be read are executed before returning to the calling process.

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, **read()** returns −1 and **errno** is set to [EAGAIN].
- If **O_NDELAY** and **O_NONBLOCK** are clear, **read()** does not return until the blocking write lock is removed.

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

- If no process has the pipe open for writing, the read returns a 0.
- If some process has the pipe open for writing and **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 some process has the pipe open for writing and **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_NDELAY** is set, the read returns 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.
- **read()** was interrupted by a signal after it had successfully read some, but not all of the data requested.

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.

r

ERRORS

read() fails if any of the following conditions are encountered:

[EBADF]	*fildes* is not a valid file descriptor open for reading.
[EINTR]	A signal was caught before any data was transferred (see *sigvector*(2)).
[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. Reliable detection of this error is implementation dependent.

[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.

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)).

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()** 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), directory(3C), tty(7).

STANDARDS CONFORMANCE

read(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, POSIX.4

NAME

readlink() - read the contents of a symbolic link

SYNOPSIS

```
#include <unistd.h>

int readlink(
      const char *path,
      char *buf,
      size_t bufsiz
);
```

DESCRIPTION

The **readlink()** function places the contents of the symbolic link referred to by path in the buffer *buf* which has size *bufsiz*. If the number of bytes in the symbolic link is less than *bufsiz*, the contents of the remainder of *buf* are unspecified.

RETURN VALUE

Upon successful completion, **readlink()** returns the count of bytes placed in the buffer. Otherwise, it returns a value of −1, leaves the buffer unchanged, and sets **errno** to indicate the error.

ERRORS

The **readlink()** function will fail if:

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

[EINVAL] The path argument names a file that is not a symbolic link.

[EIO] An I/O error occurred while reading from the file system.

[ENOENT] A component of path does not name an existing file or path is an empty string.

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

[ENAMETOOLONG]
The length of path exceeds **PATH_MAX**, or a pathname component is longer than **NAME_MAX**.

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

The **readlink()** function may fail if:

[EACCES] Read permission is denied for the directory.

[ENAMETOOLONG]
Pathname resolution of a symbolic link produced an intermediate result whose length exceeds **PATH_MAX.**

r

APPLICATION USAGE

Portable applications should not assume that the returned contents of the symbolic link are null- terminated.

SEE ALSO

stat(2), symlink(2), <unistd.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

HP-UX EXTENSIONS

SYNOPSIS

#include <symlink.h>

DESCRIPTION

If the length of the path name string is less than *bufsiz*, the string will be null-terminated when returned. If the length of the path name string is exactly *bufsiz*, the string will not be null-terminated when returned.

ERRORS

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

[EFAULT] *buf* or *path* points outside the process's allocated address space. Reliable detection of this error is implementation-dependent.

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

AUTHOR

readlink() was developed by the University of California, Berkeley.

SEE ALSO

stat(2), symlink(2), symlink(4).

STANDARDS CONFORMANCE

readlink(): AES, SVID3

r

NAME

reboot - boot the system

SYNOPSIS

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

int reboot (int howto);
```

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.

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 file systems when the system reboots.

Only users with appropriate privileges can reboot a machine.

RETURN VALUE

If successful, this call never returns. Otherwise, a –1 is returned and **errno** is set to indicate the error.

ERRORS

reboot() fails if this condition is encountered:

- [EPERM] — The effective user ID of the caller is not a user with appropriate privileges.

DEPENDENCIES

The default file and device for **RB_AUTOBOOT** is **/stand/vmunix** on the current root device.

AUTHOR

reboot() was developed by HP and the University of California, Berkeley.

SEE ALSO

reboot(1M).

NAME

recv, recvfrom, recvmsg - receive a message from a socket

SYNOPSIS

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

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

int recvfrom(
    int      s,
    void    *buf,
    int      len,
    int      flags,
    void    *from,
    int     *fromlen
);

int recvmsg(int s, struct msghdr msg[], int flags);
```

_XOPEN_SOURCE_EXTENDED Only (UNIX 98)

```
ssize_t recvfrom(
        int               s,
        void             *buf,
        size_t            len,
        int               flags,
        struct sockaddr  *from,
        socklen_t        *fromlen
);

ssize_t recvmsg(int s, struct msghdr *msg, int flags);
```

Obsolescent _XOPEN_SOURCE_EXTENDED Only (UNIX 95)

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

r

DESCRIPTION

The **recv()**, **recvfrom()**, and **recvmsg()** system calls are used to receive messages from a socket.

s is a socket descriptor from which messages are received.

buf is a pointer to the buffer into which the messages are placed.

len is the maximum number of bytes that can fit in the buffer referenced by *buf*.

If the socket uses connection-based communications, such as a SOCK_STREAM socket, these calls can only be used after the connection has been established (see *connect*(2)). For connectionless sockets such as SOCK_DGRAM, these calls can be used whether a connection has been specified or not.

recvfrom() operates in the same manner as **recv()** except that it is able to return the address of the socket from which the message was sent. For connected datagram sockets, **recvfrom()** simply returns the same address as **getpeername()** (see *getpeername*(2)). For stream sockets, **recvfrom()** retrieves data in the same manner as **recv()**, but does not return the socket address of the sender. If *from* is nonzero, the source address of the message is placed in the socket address structure pointed to by *from*. *fromlen* is a value-result parameter, initialized to the size of the structure associated with *from*, and modified on return to indicate the actual size of the address stored there. If the memory pointed to by *from* is not large enough to contain the entire address, only the first *fromlen* bytes of the address are returned.

For message-based sockets such as SOCK_DGRAM, the entire message must be read in a single operation. If a message is too long to fit in the supplied buffer, the excess bytes are discarded. For stream-based sockets such as SOCK_STREAM, there is no concept of message boundaries. In this case, data is returned to the user as soon as it becomes available, and no data is discarded. See the AF_CCITT Only subsection below for a list of the exceptions to this behavior for connections in the address family AF_CCITT.

recvmsg() performs the same action as **recv()**, but scatters the read data into the buffers specified in the **msghdr** structure (see **_XOPEN_SOURCE_EXTENDED Only** below). This structure is defined in **<sys/socket.h>** and has the following form (**HP-UX BSD Sockets Only**):

```
struct msghdr {
    caddr_t   msg_name;            /* optional address */
    int       msg_namelen;         /* size of address  */
    struct    iovec *msg_iov;      /* scatter array for data */
    int       msg_iovlen;          /* # of elements in msg_iov */
    caddr_t   msg_accrights;       /* access rights */
    int       msg_accrightslen;    /* size of msg_accrights */
}
```

msg_name points to a **sockaddr** structure in which the address of the sending socket is to be stored, if the socket is connectionless; *msg_name* may be a null pointer if no name is specified. *msg_iov* specifies the locations of the character arrays for storing the incoming data. *msg_accrights* specifies a buffer to receive any access rights sent along with the message. Access rights are limited to file descriptors of size *int*. If access rights are not being transferred, set the *msg_accrights* field to NULL. Access rights are supported only for AF_UNIX.

If no data is available to be received, **recv()** waits for a message to arrive unless nonblocking mode is enabled. There are three ways to enable nonblocking mode:

- With the **FIOSNBIO ioctl()** request
- With the **O_NONBLOCK fcntl()** flag
- With the **O_NDELAY fcntl()** flag

Although the use of **FIONBIO** is not recommended, if nonblocking I/O is enabled using **FIOSNBIO** or the equivalent **FIONBIO** request (defined in **<sys/ioctl.h>** and explained in *ioctl*(2), *ioctl*(5) and *socket*(7)), the **recv()** request completes in one of three ways:

- If there is enough data available to satisfy the entire request, **recv()** completes successfully, having read all of the data, and returns the number of bytes read.
- If there is not enough data available to satisfy the entire request, **recv()** complete successfully, having read as much data as possible, and returns the number of bytes it was able to read.
- If there is no data available, **recv()** fails and **errno** is set to [**EWOULDBLOCK**].

If nonblocking I/O is disabled using **FIOSNBIO**, **recv()** always executes completely (blocking as necessary) and returns the number of bytes read.

If the **O_NONBLOCK** flag is set using **fcntl()** (defined in **<sys/fcntl.h>** and explained in *fcntl*(2) and *fcntl*(5)), POSIX-style nonblocking I/O is enabled. In this case, the **recv()** request completes in one of three ways:

- If there is enough data available to satisfy the entire request, **recv()** completes successfully, having read all the data, and returns the number of bytes read.
- If there is not enough data available to satisfy the entire request, **recv()** completes successfully, having read as much data as possible, and returns the number of bytes it was able to read.
- If there is no data available, **recv()** completes, having read no data, and returns –1 with **errno** set to [EAGAIN].

If the **O_NDELAY** flag is set using **fcntl()** (defined in **<sys/fcntl.h>** and explained in *fcntl*(2) and *fcntl*(5)), nonblocking I/O is enabled. In this case, the **recv()** request completes in one of three ways:

- If there is enough data available to satisfy the entire request, **recv()** completes successfully, having read all the data, and returns the number of bytes read.
- If there is not enough data available to satisfy the entire request, **recv()** completes successfully, having read as much data as possible, and returns the number of bytes it was able to read.
- If there is no data available, **recv()** completes successfully, having read no data, and returns 0.

If the **O_NONBLOCK** or **O_NDELAY** flag is cleared using **fcntl()**, the corresponding style of nonblocking I/O, if previously enabled, is disabled. In this case, **recv()** always executes completely (blocking as necessary) and returns the number of bytes read.

Since both the **fcntl()** **O_NONBLOCK** and **O_NDELAY** flags and **ioctl()** **FIOSNBIO** request are supported, some clarification on how these features interact is necessary. If the **O_NONBLOCK** or **O_NDELAY** flag has been set, **recv()** requests behave accordingly, regardless of any **FIOSNBIO** requests. If neither the **O_NONBLOCK** flag nor the **O_NDELAY** flag has been set, **FIOSNBIO** requests control the the behavior of **recv()**.

By default nonblocking I/O is disabled.

select() can be used to determine when more data arrives by selecting the socket for reading.

The *flags* parameter can be set to **MSG_PEEK**, **MSG_OOB**, both, or zero. If it is set to **MSG_PEEK**, any data returned to the user still is treated as if it had not been read. The next **recv()** rereads the same data. The **MSG_OOB** flag is used to receive out-of-band data. For TCP SOCK_STREAM sockets, both the **MSG_PEEK** and **MSG_OOB** flags can be set at the same time. The **MSG_OOB** flag value is supported for TCP SOCK_STREAM sockets only. **MSG_OOB** is not supported for AF_UNIX or AF_VME_LINK sockets.

A **read()** call made to a socket behaves in exactly the same way as a **recv()** with *flags* set to zero.

AF_CCITT Only

Connections in the address family AF_CCITT support message-based sockets only. Although the user specifies connection-based communications (SOCK_STREAM), the X.25 subsystem communicates via messages. This address family does not support SOCK_DGRAM socket types.

Normally, each **recv()** returns one complete X.25 message. If the socket is in nonblocking mode, **recv()** behaves as described above. Note that if the user specifies *len* less than the actual X.25 message size, the excess data is discarded and no error indication is returned. The size of the next available message as well as the state of MDTF, D, and Q bits can be obtained with **ioctl(X25_NEXT_MSG_STAT)**.

Connections of the address family AF_CCITT receive data in the same way as message-based connections described above, with the following additions and exceptions:

- **recvfrom()** is supported; however, the *from* and *fromlen* parameters are ignored (that is, it works in the same manner as **recv()**).
- To receive a message in fragments of the complete X.25 message, use **ioctl(X25_SET_FRAGMENT_SIZE)**. The state of the MDTF bit is 1 for all except the last fragment of the message.
- The **MSG_OOB** flag is supported.
- The **MSG_PEEK** flag is supported; the two flags can be combined.
- If a message is received that is larger than the user-controlled maximum message size (see *af_ccitt*(7F)), the X.25 subsystem RESETs the circuit, discards the data, and sends the out-of-band event **OOB_VC_MESSAGE_TOO_BIG** to the socket.

r

_XOPEN_SOURCE_EXTENDED Only

For **X/Open Sockets**, the **msghdr** structure has the following form:

(UNIX 98)

```
struct msghdr {
    void       *msg_name;           /* optional address */
    socklen_t   msg_namelen;        /* size of address  */
    struct      iovec *msg_iov;     /* scatter array for data */
    int         msg_iovlen;         /* # of elements in msg_iov */
    void       *msg_control;        /* ancillary data, see below */
    socklen_t   msg_controllen;     /* ancillary data buffer len */
    int         msg_flags;          /* flags on received message */
}
```

Obsolescent (UNIX 95)

```
struct msghdr {
    void       *msg_name;           /* optional address */
    size_t      msg_namelen;        /* size of address  */
    struct      iovec *msg_iov;     /* scatter array for data */
    int         msg_iovlen;         /* # of elements in msg_iov */
    void       *msg_control;        /* ancillary data, see below */
```

```
        size_t      msg_controllen;     /* ancillary data buffer len */
        int         msg_flags;          /* flags on received message */
    }
```

msg_control specifies a buffer to receive any ancillary data sent along with the message. Ancillary data consists of a sequence of pairs, each consisting of a **cmsghdr** structure followed by a data array. The data array contains the ancillary data message, and the **cmsghdr** structure contains descriptive information that allows an application to correctly parse the data. **cmsghdr** has the following structure:

```
    (UNIX 98)

    struct cmsghdr {
        socklen_t  cmsg_len;      /* data byte count, including hdr*/
        int        cmsg_level;    /* originating protocol */
        int        cmsg_type;     /* protocol-specific type */
    }

    Obsolescent (UNIX 95)

    struct cmsghdr {
        size_t     cmsg_len;      /* data byte count, including hdr*/
        int        cmsg_level;    /* originating protocol */
        int        cmsg_type;     /* protocol-specific type */
    }
```

The supported value for cmsg_level is SOL_SOCKET, and the supported value for cmsg_type is SCM_RIGHTS. Together they indicate that the data array contains the access rights to be received. Access rights are supported only for AF_UNIX. Access rights are limited to file descriptors of size *int*. If ancillary data are not being transferred, set the *msg_control* field to NULL, and set the *msg_controllen* field to 0.

The *flags* parameter accepts a new value, **MSG_WAITALL**, which requests that the function block until the full amount of data requested can be returned. The function may return a smaller amount of data if a signal is caught, the connection is terminated, or an error is pending for the socket.

On successful completion of **recvmsg()**, the *msg_flags* member of the message header is the bitwise-inclusive OR of all of the following flags that indicate conditions detected for the received message.

MSG_EOR	End of record was received (if supported by the protocol).
MSG_OOB	Out-of-band data was received.
MSG_TRUNC	Normal data was truncated.
MSG_CTRUNC	Control data was truncated.

r

DEPENDENCIES

AF_CCITT

recvfrom() is supported; however, the *from* and *fromlen* parameters are ignored (i.e., it works in the same manner as **recv()**).

The **O_NDELAY fcntl()** call is not supported over X.25 links. Use the **FIOSNBIO ioctl()** call instead to enable nonblocking I/0.

RETURN VALUE

recv(), **recvfrom()**, and **recvmsg()** return the following values:

- *n* Successful completion. *n* is the number of bytes received.
- **0** The socket is blocking and the transport connection to the remote node failed.
- **-1** Failure. **errno** is set to indicate the error.

ERRORS

If **recv()**, **recvfrom()**, or **recvmsg()** fails, **errno** is set to one of the following values.

[EAGAIN] Non-blocking I/O is enabled using **O_NONBLOCK** flag with **fcntl()** and the receive operation would block, or the socket has an error that was set asynchronously. An asynchronous error can be caused by a gateway failing to forward a datagram because the datagram exceeds the MTU of the next-hop network and the "Don't Fragment" (DF) bit in the datagram is set. (See **SO_PMTU** in *getsockopt*(2).)

- [EBADF] The argument *s* is an invalid descriptor.
- [ECONNRESET] A connection was forcibly closed by a peer.
- [EFAULT] An invalid pointer was specified in the *buf* , *from* , or *fromlen* parameter, or in the msghdr structure.
- [EINTR] The receive was interrupted by delivery of a signal before any data was available for the receive.
- [EINVAL] The *len* parameter or a length in the msghdr structure is invalid; or no data is available on receive of out of band data.
- [EMSGSIZE] A length in the msghdr structure is invalid.
- [ENOBUFS] Insufficient resources were available in the system to perform the operation.
- [ENOTCONN] Receive on a SOCK_STREAM socket that is not yet connected.
- [ENOTSOCK] The argument *s* is a valid file descriptor, but it is not a socket.
- [EOPNOTSUPP] The **MSG_OOB** flag was set for a UDP SOCK_DGRAM message-based socket, or **MSG_OOB** or **MSG_PEEK** was set for any AF_UNIX socket. The **MSG_OOB** flag is supported only for stream-based TCP SOCK_STREAM sockets. Neither **MSG_PEEK** nor **MSG_OOB** is supported for AF_UNIX sockets.

  AF_CCITT only: **recv()** was issued on a **listen()** socket.
- [ETIMEDOUT] The connection timed out during connection establishment, or due to a transmission timeout on active connection.
- [EWOULDBLOCK] Non-blocking I/O is enabled using **ioctl() FIOSNBIO** request, and the requested operation would block.

OBSOLESCENCE

Currently, the **socklen_t** and **size_t** types are the same size. This is compatible with both the UNIX 95 and UNIX 98 profiles. However, in a future release, **socklen_t** might be a different size. In that case, passing a **size_t** pointer will evoke compile-time warnings, which must be corrected in order for the application to behave correctly. Also, the size of the **msghdr** and **cmsghdr** structures and the relative position of their members will be different, which might affect application behavior. Applications that use **socklen_t** now, where appropriate, will avoid such migration problems. On the other hand, applications that need to be portable to the UNIX 95 profile should follow the X/Open specification (see *xopen_networking*(7)).

FUTURE DIRECTION

Currently, the default behavior is the **HP-UX BSD Sockets**; however, it might be changed to **X/Open Sockets** in a future release. At that time, any **HP-UX BSD Sockets** behavior that is incompatible with **X/Open Sockets** might be obsoleted. Applications that conform to the X/Open specification now will avoid migration problems (see *xopen_networking*(7)).

MULTITHREAD USAGE

The **recv()**, **recvmsg()**, and **recvfrom()** system calls are thread-safe. They each have a cancellation point; and they are async-cancel safe, async-signal safe, and fork-safe.

AUTHOR

recv(), **recvmsg()**, and **recvfrom()** were developed by HP and the University of California, Berkeley.

SEE ALSO

getsockopt(2), read(2), select(2), send(2), socket(2), af_ccitt(7F), af_vme_link(7F), inet(7F), socket(7), socketx25(7), tcp(7P), udp(7P), unix(7P), xopen_networking(7).

STANDARDS CONFORMANCE

recv(): XPG4

NAME

rename - change the name of a file

SYNOPSIS

```
#include <stdio.h>

int rename(const char *source, const char *target);
```

DESCRIPTION

The **rename()** system call causes the *source* file to be renamed to *target*. If *target* exists, it is first removed. Both *source* and *target* must be of the same type (that is, either directories or nondirectories), 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

rename() returns the following values:

- **0** Successful completion.
- **-1** Failure. Neither file is affected. **errno** is set to indicate the error.

ERRORS

If **rename()** fails, **errno** is set to one of the following values.

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

[EACCES] The requested link requires writing to a directory without write permission.

[EBUSY] *target* or *source* is an existing directory that is the mount point for a mounted file system.

[EDQUOT] User's disk quota block or inode limit has been reached for this file system.

[EEXIST] *target* is a directory and is not empty.

[EFAULT] *source* or *target* points outside the allocated address space of the process. Reliable detection of this error is implementation dependent.

[EINVAL] *source* is a parent directory of *target*, or an attempt is made to rename the **.** or **..** directory.

[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.

[EPERM] The directory containing *source* has the sticky bit set, and neither the containing directory nor the *source* are owned by the effective user ID.

[EPERM] The *target* file exists, the directory containing *target* has the sticky bit set, and neither the containing directory nor the *target* are owned by the effective user ID.

[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.

SEE ALSO

open(2).

STANDARDS CONFORMANCE

rename(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

rmdir() - remove a directory file

SYNOPSIS

```
#include <unistd.h>

int rmdir(const char *path);
```

DESCRIPTION

The **rmdir()** system call removes a directory file whose name is given by *path*. The directory must be empty (except for the files **.** and **..**) before it can be removed.

RETURN VALUE

rmdir() returns the following values:

- **0** Successful completion.
- **-1** Failure. **errno** is set to indicate the error.

ERRORS

If **rmdir()** fails, **errno** is set to one of the following values.

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

[EACCES] Write permission is denied on the directory containing the link to be removed.

[EACCES] The process does not have read/write access permission to the parent directory.

[EBUSY] The directory to be removed is the mount point for a mounted file system.

[EBUSY] The path is the current working directory.

[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.

[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.

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

[EPERM] The directory containing the directory to be removed has the sticky bit set and neither the containing directory nor the directory to be removed are owned by the effective user ID.

[EROFS] The directory entry to be removed resides on a read-only file system.

AUTHOR

rmdir() was developed by the University of California, Berkeley and HP.

SEE ALSO

mkdir(2), unlink(2), remove(3C).

STANDARDS CONFORMANCE

rmdir(): AES, SVID2, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

rtprio - change or read real-time priority

SYNOPSIS

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

int rtprio(pid_t pid, int prio);
```

DESCRIPTION

The **rtprio()** system call sets or reads the real-time priority of a process.

If *pid* is zero, it specifies the calling process; otherwise, it specifies the process ID of a process.

If the process *pid* contains more than one thread or a lightweight process (that is, the process is multi-threaded), this function shall only change the process scheduling policy and priority. Individual threads or lightweight processes in the target process shall not have their scheduling policies and priorities modified. Note that if the target process is multi-threaded, this process scheduling policy and priority change will only affect a child process that is created later and inherits its parent's scheduling policy and priority. The priority returned is the value of the target's old priority, though individual threads or lightweight processes may have a different value if some other interface is used to change an individual thread or lightweight processes priority.

When setting the real-time 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 a user having appropriate privileges. 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 a user having appropriate privileges.

Simply reading real-time priorities requires no special privilege.

Real-time scheduling policies differ from normal timesharing policies in that the real-time priority is used to absolutely order all real-time processes. This priority is not degraded over time. All real-time processes are of higher priority than normal user and system processes, although some system processes may run at real-time priorities. If there are several eligible processes at the same priority level, they are run in a round robin fashion as long as no process with a higher priority intervenes. A real-time process receives CPU service until it either voluntarily gives up the CPU or is preempted by a process of equal or higher priority. Interrupts can also preempt a real-time process.

Valid real-time priorities run from zero to 127. Zero is the highest (most important) priority. This real-time priority is inherited across forks (see *fork*(2)) and execs (see *exec*(2)).

prio can have the following values:

0 to **127** — Set the process to this real-time priority.

RTPRIO_NOCHG — Do not change the real-time priority. This is used to read the process real-time priority.

RTPRIO_RTOFF — Set the process to no longer have a real-time priority. It resumes a normal timesharing priority.

Any process, regardless of privilege, is allowed to turn off its own real-time priority using a *pid* of zero.

RETURN VALUE

rtprio() returns the following values:

0 to **127** — The process was a real-time process. The value is the process's former (before the call) real-time priority.

RTPRIO_RTOFF — The process was not a real-time process.

-1 — An error occurred. **errno** is set to indicate the error.

ERRORS

If **rtprio()** fails, **errno** is set to one of the following values:

[EINVAL] — *prio* is not **RTPRIO_NOCHG**, **RTPRIO_RTOFF**, or in the range 0 to 127.

[EPERM] — The calling process is not a user having appropriate privileges, and neither its real nor 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*.

EXAMPLES

The following call to **rtprio()** sets the calling process to a real-time priority of 90:

```
rtprio(0, 90);
```

WARNINGS

Normally, compute-bound programs should not be run at real-time priorities, because all timesharing work on the CPU would come to a complete halt.

DEPENDENCIES

Series 800

Because processes executing at real-time priorities get scheduling preference over a system process executing at a lower priority, unexpected system behavior can occur after a power failure on systems that support power-fail recovery. For example, when **init** (see *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, the running of **init** is delayed. This condition temporarily prevents terminal input to any process, including real-time shells of higher priority than the eligible real-time process. To avoid this situation, a real-time 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).

NAME

rtsched: sched_get_priority_max(), sched_get_priority_min(), sched_getparam(), sched_getscheduler(), sched_rr_get_interval(), sched_setparam(), sched_setscheduler(), sched_yield(), PRI_HPUX_TO_POSIX(), PRI_POSIX_TO_HPUX() - real-time scheduling operations

SYNOPSIS

```
#include <sched.h>

int sched_setparam(
    pid_t pid,
    const struct sched_param *param
);

int sched_getparam(
    pid_t pid,
    struct sched_param *param
);

int sched_setscheduler(
    pid_t pid,
    int policy,
    const struct sched_param *param
);

int sched_getscheduler(
    pid_t pid
);

int sched_yield();

int sched_get_priority_max(
    int policy
);

int sched_get_priority_min(
    int policy
);

int sched_rr_get_interval(
    pid_t pid,
    struct timespec *interval
);

int PRI_POSIX_TO_HPUX(
    const struct sched_param *param
);

int PRI_HPUX_TO_POSIX(
    int pri,
    struct sched_param *param
);
```

DESCRIPTION

Summary

sched_get_priority_max()	Get maximum scheduling policy
sched_get_priority_min()	Get minimum scheduling policy
sched_getparam()	Get scheduling parameters of process
sched_getscheduler()	Get scheduling policy of process
sched_rr_get_interval()	Update execution time limit for a process
sched_setparam()	Set scheduling parameters of process
sched_setscheduler()	Set scheduling policy and parameters of process
sched_yield()	Requeue current process in process list
PRI_HPUX_TO_POSIX()	Convert HP-UX priority to POSIX
PRI_POSIX_TO_HPUX()	Convert POSIX priority to HP-UX

sched_setparam()

The **sched_setparam()** function sets the scheduling parameters of the process specified by *pid* to the values specified by the *sched_param* structure pointed to by *param*. The value of the *sched_priority* member in the *param* structure is any integer within the inclusive priority range for the current scheduling policy of the process specified by *pid*.

Higher numerical values for the priority represent higher (stronger) priorities. Note that this is different from the **SCHED_HPUX**, **SCHED_TIMESHARE**, and **SCHED_RTPRIO** scheduling policies, where higher numerical values represent lower (weaker) priorities. See the **PRI_HPUX_TO_POSIX()** and **PRI_POSIX_TO_HPUX()** functions, and **SCHED_RTPRIO** and **SCHED_OTHER** in "Scheduling Policies" below.

If a process described by *pid* exists and if the calling process has permission, the scheduling parameters are set for the process whose process ID is equal to *pid*.

If *pid* is zero, the scheduling parameters are set for the calling process.

If the process *pid* contains more than one thread or lightweight process (that is, the process is multi-threaded), this function shall only change the process scheduling policy and priority. Individual threads or lightweight processes in the target process shall not have their scheduling policies and priorities modified. Note that if the target process is multi-threaded, this process scheduling policy and priority change will only affect a child process that is created later and inherits its parent's scheduling policy and priority. The priority returned is the old priority of the target process, though individual threads or lightweight processes may have a different value if some other interface is used to change an individual thread or lightweight processes priority.

Only a superuser may change the scheduling parameters of another process.

The calling process must have the appropriate privileges or be a member of a group having **PRIV_RTSCHED** access to successfully call **sched_setparam()**.

The target process, whether it is running or not running, will resume execution after all other runnable processes of equal or greater priority have been scheduled to run.

If the priority of the process specified by the *pid* argument is set higher than that of the lowest priority running process, and if the specified process is ready to run, the process specified by the *pid* argument will preempt a lowest-priority running process. Similarly, if the process calling **sched_setparam()** sets its own priority lower than that of one or more other nonempty process lists, then the process that is the head of the highest priority list will also preempt the calling process. Thus, in either case, the originating process may not receive notification of the completion of the requested priority change until the higher priority process has executed.

sched_getparam()

The **sched_getparam()** function returns the scheduling parameters of a process specified by *pid* in the *sched_param* structure pointed to by *param*.

If a process described by *pid* exists, the scheduling parameters are returned for the process whose process ID is equal to *pid*.

If the process *pid* contains more than one thread or lightweight process (that is, the process is multi-threaded), this function shall only return the process scheduling policy and priority. Individual threads or lightweight processes in the target process will have their own scheduling policies and priorities which may be different from the scheduling policy and priority of their process.

If *pid* is zero, the scheduling parameters are returned for the calling process.

sched_setscheduler()

The **sched_setscheduler()** function sets the scheduling policy and scheduling parameters of the process specified by *pid* to *policy* and the parameters specified in the *sched_param* structure pointed to by *param*, respectively. The value of the *sched_priority* member in the *param* structure can be any integer within the inclusive priority range for the scheduling policy specified by *policy*.

The possible values for the *policy* parameter are defined in the header file **<sched.h>**, and mentioned below.

If a process described by *pid* exists, the scheduling policy and scheduling parameters are set for the process whose process ID is equal to *pid*.

If *pid* is zero, the scheduling policy and scheduling parameters are set for the calling process.

If the process *pid* contains more than one thread or lightweight process (that is, the process is multi-threaded), this function shall only change the process scheduling policy and priority. Individual threads or lightweight processes in the target process shall not have their scheduling policies and priorities modified. Note: If the target process is multi-threaded, this change will only affect a child process that is created later and inherits its parent's scheduling policy and priority. The priority returned is the old priority of the target process, though individual threads or lightweight processes may have a different value if some other interface is used to change an individual thread or lightweight process' priority.

Appropriate privileges are required to change the scheduling parameters of another process.

The calling process must have appropriate privileges or be a member of a group having **PRIV_RTSCHED** access to successfully call **sched_setscheduler()**.

The **sched_setscheduler()** function is considered successful if it succeeds in setting the scheduling policy and scheduling parameters of the process specified by *pid* to the values specified by *policy* and the structure *param*, respectively.

sched_getscheduler()

The **sched_getscheduler()** function returns the scheduling policy of the process specified by *pid*.

The values that can be returned by **sched_getscheduler()** are defined in the header file **<sched.h>** (see **sched_setscheduler()**).

If a process described by *pid* exists, the scheduling policy is returned for the process whose process ID is equal to *pid*.

If *pid* is zero, the scheduling policy is returned for the calling process.

If the process *pid* contains more than one thread or lightweight process (that is, the process is multi-threaded), this function shall only return the process scheduling policy and priority. Individual threads or lightweight processes in the target process will have their own scheduling policies and priorities which may be different from the scheduling policy and priority of their process.

sched_yield()

The **sched_yield()** function forces the running process to relinquish the processor until it again becomes the head of its process list. It takes no arguments.

sched_get_priority_max()
sched_get_priority_min()

The **sched_get_priority_max()** and **sched_get_priority_min()** functions return the appropriate maximum or minimum, respectively, for the scheduling policy specified by *policy*.

The value of *policy* must be one of the scheduling policy values defined in **<sched.h>**.

sched_rr_get_interval()

The **sched_rr_get_interval()** function updates the *timespec* structure referenced by the *interval* argument to contain the current execution time limit (that is, time quantum) for the process indicated by *pid* under the **SCHED_RR** policy, at which a scheduling decision will be made when another process at the same priority is ready to execute. If *pid* is zero, the current execution time limit for the calling process is returned.

PRI_HPUX_TO_POSIX()
PRI_POSIX_TO_HPUX()

These two functions serve to map (translate) the range of HP-UX priorities into the POSIX.4 model. These translations are necessary because the POSIX.4 standard chose *larger* numbers to represent stronger priorities and the existing HP-UX behavior, which must be maintained for backward compatibility, uses *smaller* numbers for stronger priorities.

The **PRI_HPUX_TO_POSIX()** function returns, in the *sched_param* structure pointed to by *param*, the POSIX.4 scheduling priority corresponding to the HP-UX priority passed in the argument *pri*. The argument *pri* must contain a valid HP-UX priority.

The **PRI_POSIX_TO_HPUX()** function returns an HP-UX process priority corresponding to the *sched_priority* member in the *sched_param* structure specified. The value of the *sched_priority* member can be any integer within the inclusive priority range for the **SCHED_HPUX** scheduling policy. The HP-UX priority returned is comparable to the values returned by **getpriority()** (see *getpriority*(2)).

r

Scheduling Policies

The scheduling policies described are defined in terms of a conceptual model, which contains a set of process lists. There is, conceptually, one process list for each priority. Any runnable process may be in any process list. Multiple scheduling policies are provided. Each nonempty list is ordered, and contains a head as one end of its order, and a tail as the other. The purpose of a scheduling policy is to define the allowable operations on this set of lists (for example, moving processes between and within lists).

Each process will be controlled by an associated scheduling policy and priority. These parameters may be specified by explicit application execution of the **sched_setscheduler()** or **sched_setparam()** functions.

Associated with each policy is a priority range. The priority ranges for each policy can (but need not) overlap the priority ranges of other policies.

When a process is to be selected to run, the process that is at the head of the highest priority nonempty process list is chosen. It is then removed from its process list.

The following scheduling policies are defined:

SCHED_FIFO First in-first out (FIFO) scheduling policy.

Processes scheduled under this policy are chosen from a process list that is ordered by the time its processes have been in the list without being executed. Generally, the head of the list is the process that has been in the list the longest time, and the tail is the process that has been in the list the shortest time.

Under the **SCHED_FIFO** policy, the modification of the definitional process lists is as follows:

- When a running process becomes a preempted process, it becomes the head of the process list for its priority.
- When a blocked process becomes a runnable process, it becomes the tail of the process list for its priority.
- When a running process calls the **sched_setscheduler()** function, the process specified in the function call is modified to the policy and priority specified by the *param* argument. If the process whose policy and priority has been modified is a running process or is runnable, it then becomes the tail of the process list for its new priority.
- When a running process calls the **sched_setparam()** function, the priority of the process specified in the function call is modified to the priority specified by the *param* argument. If the process whose priority has been modified is a running process or is runnable, it then becomes the tail of the process list for its new priority.
- When a running process issues the **sched_yield()** function, the process becomes the tail of the process list for its priority.
- At no other time is the position of a process with this scheduling policy within the process lists affected.

For this policy, valid priorities are within the range returned by the functions **sched_get_priority_max()** and **sched_get_priority_min()** when **SCHED_FIFO** is provided as the parameter. The priority range for this policy contains at least 32 priorities.

SCHED_RR Round-robin scheduling policy, with a per-system time slice (time quantum).

This policy is identical to the **SCHED_FIFO** policy with the additional condition that when the implementation detects that a running process has been executing as a running process for a time period of length returned by the function **sched_rr_get_interval()**, or longer, the process becomes the tail of its process list, and the head of that process list is removed and made a running process.

The effect of this policy is to ensure that if there are multiple **SCHED_RR** processes at the same priority, one of them will not monopolize the processor. An application should not rely only on the use of **SCHED_RR** to ensure application progress among multiple processes if the application includes processes using the **SCHED_FIFO** policy at the same or higher priority levels, or **SCHED_RR** processes at a higher priority

r

level.

A process under this policy that is preempted and subsequently resumes execution as a running process completes the unexpired portion of its round-robin interval time period.

For this policy, valid priorities are within the range returned by the functions **sched_get_priority_max()** and **sched_get_priority_min()** when **SCHED_RR** is provided as the parameter. The priority range for this policy contains at least 32 priorities.

SCHED_RR2 Round-robin scheduling policy, with a per-priority time slice (time quantum).

This policy is identical to the **SCHED_RR** policy, except that the round-robin time slice interval returned by **sched_rr_get_interval()** depends upon the priority of the specified process.

For this policy, valid priorities are within the range returned by the functions **sched_get_priority_max()** and **sched_get_priority_min()** when **SCHED_RR** is provided as the parameter. The priority range for this policy contains at least 32 priorities.

SCHED_RTPRIO

Real-time scheduling policy with nondecaying priorities (like **SCHED_FIFO** and **SCHED_RR**) with a priority range between the POSIX real-time policies and the HP-UX policies, described below (see *rtprio*(2)).

For processes executing under this policy, the implementation must use only priorities within the range returned by the functions **sched_get_priority_max()** and **sched_get_priority_min()** when **SCHED_RTPRIO** is provided as the parameter. Note that, for the **SCHED_RTPRIO** scheduling policy, *smaller* numbers represent higher (stronger) priorities, which is the opposite of the POSIX scheduling policies. This is done to provide continuing support for existing applications that depend on this priority ordering. However, it is guaranteed that the priority range for the **SCHED_OTHER** scheduling policy is properly disjoint from the priority ranges of all of the other scheduling policies described and the strongest priority in the priority range for **SCHED_RTPRIO** is weaker than the weakest priority in the priority ranges for any of the POSIX policies, **SCHED_FIFO**, **SCHED_RR**, and **SCHED_RR2**.

SCHED_OTHER (**SCHED_HPUX**, **SCHED_TIMESHARE**)

Another scheduling policy.

The **SCHED_OTHER** policy, also known as **SCHED_HPUX** and **SCHED_TIMESHARE**, provides a way for applications to indicate, in a portable way, that they no longer need a real-time scheduling policy.

For processes executing under this policy, the implementation can use only priorities within the range returned by the functions **sched_get_priority_max()** and **sched_get_priority_min()** when **SCHED_OTHER** is provided as the parameter. Note that for the **SCHED_OTHER** scheduling policy, like **SCHED_RTPRIO**, *smaller* numbers represent higher (stronger) priorities, which is the opposite of the POSIX scheduling policies. This is done to provide continuing support for existing applications that depend on this priority ordering. However, it is guaranteed that the priority range for the **SCHED_OTHER** scheduling policy is properly disjoint from the priority ranges of all of the other scheduling policies described and the strongest priority in the priority range for **SCHED_OTHER** is weaker than the weakest priority in the priority ranges for any of the other policies, **SCHED_FIFO**, **SCHED_RR**, and **SCHED_RR2**.

RETURN VALUE

The functions return the following values:

sched_getparam()
sched_rr_get_interval()
sched_setparam()

sched_yield()
PRI_HPUX_TO_POSIX()

- **0** Successful completion.
- **-1** Failure. **errno** is set to indicate the error.

sched_setscheduler()

- *n* Successful completion. *n* is the former scheduling policy of the specified process.
- **-1** Failure. The policy and scheduling parameters remain unchanged. **errno** is set to indicate the error.

sched_getscheduler()

- *n* Successful completion. *n* is the scheduling policy of the specified process.
- **-1** Failure. **errno** is set to indicate the error.

sched_get_priority_max()
sched_get_priority_min()

- *n* Successful completion. *n* is the maximum or minimum value, respectively.
- **-1** Failure. **errno** is set to indicate the error.

PRI_POSIX_TO_HPUX()

- *n* Successful completion. *n* is the the HP-UX priority corresponding to the *sched_priority* member in the *param* structure.
- **-1** Failure. **errno** is set to indicate the error.

ERRORS

If the functions fail, **errno** is set to one of the following values.

sched_setparam()

- [EFAULT] The *param* argument points to an invalid address.
- [EINVAL] One or more of the requested scheduling parameters is outside the range defined for the scheduling policy of the specified *pid*.
- [ENOSYS] The function is not supported by this implementation.
- [EPERM] The requesting process does not have permission to set the scheduling parameters for the specified process, or does not have the appropriate privilege to invoke **sched_setparam()**.
- [ESRCH] No process can be found corresponding to that specified by *pid*.

sched_getparam()

- [EFAULT] The *param* argument points to an invalid address.
- [ENOSYS] The function is not supported by this implementation.
- [ESRCH] No process can be found corresponding to that specified by *pid*.

sched_setscheduler()

- [EFAULT] The *param* argument points to an invalid address.
- [EINVAL] The value of the *policy* parameter is invalid, or one or more of the parameters contained in *param* is outside the valid range for the specified scheduling policy.
- [ENOSYS] The function is not supported by this implementation.
- [EPERM] The requesting process does not have permission to set the scheduling policy of the specified process.
- [ESRCH] No process can be found corresponding to that specified by *pid*.

sched_getscheduler()

- [ENOSYS] The function is not supported by this implementation.
- [ESRCH] No process can be found corresponding to that specified by *pid*.

r

sched_yield()

[ENOSYS] The function is not supported by this implementation.

sched_get_priority_max()
sched_get_priority_min()

[EINVAL] The value of the *policy* parameter does not represent a defined scheduling policy.

[ENOSYS] The function is not supported by this implementation.

sched_rr_get_interval()

[ENOSYS] The function is not supported by this implementation.

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

PRI_POSIX_TO_HPUX()

[EINVAL] The priority specified in the *sched_priority* member of the *param* argument is outside the range defined for the **SCHED_HPUX** scheduling policy.

[ENOSYS] The function is not supported by this implementation.

PRI_HPUX_TO_POSIX()

[EINVAL] The priority specified in the *pri* argument is not a valid HP-UX priority.

[ENOSYS] The function is not supported by this implementation.

EXAMPLES

Change the calling process to use the strongest **FIFO** priority:

```
#include <sched.h>

struct sched_param param;
int maxpri;

maxpri = sched_get_priority_max(SCHED_FIFO);
if (maxpri == -1) {
    perror("sched_get_priority_max() failed");
    exit(1);
}
param.sched_priority = maxpri;
if (sched_setscheduler(getpid(), SCHED_FIFO, &param) == -1) {
    perror("sched_setscheduler() failed");
    exit(1);
}
```

r

AUTHOR

The **sched_*()** functions were derived from the proposed IEEE POSIX P1003.4 standard, draft 14.

PRI_HPUX_TO_POSIX() and **PRI_POSIX_TO_HPUX()** were developed by HP.

SEE ALSO

rtsched(1), rtprio(2).

STANDARDS CONFORMANCE

sched_get_priority_max(): POSIX.4

sched_get_priority_min(): POSIX.4

sched_getparam(): POSIX.4

sched_getscheduler(): POSIX.4

sched_rr_getinterval(): POSIX.4

sched_setparam(): POSIX.4

sched_setscheduler(): POSIX.4

sched_yield(): POSIX.4

NAME

select - synchronous I/O multiplexing

SYNOPSIS

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

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set
*errorfds, struct timeval *timeout);

void FD_CLR(int fd, fd_set *fdset);

int FD_ISSET(int fd, fd_set *fdset);

void FD_SET(int fd, fd_set *fdset);

void FD_ZERO(fd_set *fdset);
```

DESCRIPTION

The **select()** function indicates which of the specified file descriptors is ready for reading, ready for writing, or has an error condition pending. If the specified condition is false for all of the specified file descriptors, **select()** blocks, up to the specified timeout interval, until the specified condition is true for at least one of the specified file descriptors.

The **select()** function supports regular files, terminal and pseudo-terminal devices, STREAMS-based files, FIFOs and pipes. The behaviour of **select()** on file descriptors that refer to other types of file is unspecified.

The *nfds* argument specifies the range of file descriptors to be tested. The **select()** function tests file descriptors in the range of 0 to *nfds* −1.

If the *readfds* argument is not a null pointer, it points to an object of type *fd_set* that on input specifies the file descriptors to be checked for being ready to read, and on output indicates which file descriptors are ready to read.

If the *writefds* argument is not a null pointer, it points to an object of type *fd_set* that on input specifies the file descriptors to be checked for being ready to write, and on output indicates which file descriptors are ready to write.

If the *errorfds* argument is not a null pointer, it points to an object of type *fd_set* that on input specifies the file descriptors to be checked for error conditions pending, and on output indicates which file descriptors have error conditions pending.

On successful completion, the objects pointed to by the *readfds*, *writefds*, and *errorfds* arguments are modified to indicate which file descriptors are ready for reading, ready for writing, or have an error condition pending, respectively. For each file descriptor less than *nfds*, the corresponding bit will be set on successful completion if it was set on input and the associated condition is true for that file descriptor.

If the *timeout* argument is not a null pointer, it points to an object of type *struct* timeval that specifies a maximum interval to wait for the selection to complete. If the *timeout* argument points to an object of type *struct* timeval whose members are 0, **select()** does not block. If the *timeout* argument is a null pointer, **select()** blocks until an event causes one of the masks to be returned with a valid (non-zero) value. If the time limit expires before any event occurs that would cause one of the masks to be set to a non-zero value, **select()** completes successfully and returns 0.

Implementations may place limitations on the maximum timeout interval supported. On all implementations, the maximum timeout interval supported will be at least 31 days. If the timeout argument specifies a timeout interval greater than the implementation- dependent maximum value, the maximum value will be used as the actual timeout value. Implementations may also place limitations on the granularity of timeout intervals. If the requested timeout interval requires a finer granularity than the implementation supports, the actual timeout interval will be rounded up to the next supported value.

If the *readfds*, *writefds*, and *errorfds* arguments are all null pointers and the timeout argument is not a null pointer, **select()** blocks for the time specified, or until interrupted by a signal. If the *readfds*, *writefds*, and *errorfds* arguments are all null pointers and the *timeout* argument is a null pointer, **select()** blocks until interrupted by a signal.

File descriptors associated with regular files always select true for ready to read, ready to write, and error conditions.

On failure, the objects pointed to by the *readfds*, *writefds*, and *errorfds* arguments are not modified. If the timeout interval expires without the specified condition being true for any of the specified file descriptors, the objects pointed to by the *readfds*, *writefds*, and *errorfds* arguments have all bits set to 0.

File descriptor masks of type *fd_set* can be initialised and tested with **FD_CLR()**, **FD_ISSET()**, **FD_SET()**, and **FD_ZERO()**. It is unspecified whether each of these is a macro or a function. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with any of these names, the behaviour is undefined.

FD_CLR(fd, &fdset) Clears the bit for the file descriptor *fd* in the file descriptor set *fdset*.

FD_ISSET(fd, &fdset) Returns a non-zero value if the bit for the file descriptor *fd* is set in the file descriptor set pointed to by *fdset*, and 0 otherwise.

FD_SET(fd, &fdset) Sets the bit for the file descriptor *fd* in the file descriptor set *fdset*.

FD_ZERO(&fdset) Initialises the file descriptor set *fdset* to have zero bits for all file descriptors. The behaviour of these macros is undefined if the *fd* argument is less than 0 or greater than or equal to **FD_SETSIZE**.

RETURN VALUE

FD_CLR(), **FD_SET()**, and **FD_ZERO()** return no value. **FD_ISSET()** returns a non-zero value if the bit for the file descriptor *fd* is set in the file descriptor set pointed to by *fdset*, and 0 otherwise.

On successful completion, **select()** returns the total number of bits set in the bit masks. Otherwise, −1 is returned, and **errno** is set to indicate the error.

ERRORS

Under the following conditions, **select()** fails and sets **errno** to:

[EBADF] One or more of the file descriptor sets specified a file descriptor that is not a valid open file descriptor. This could happen either if the file descriptor sets are not initialised or *nfds* argument is greater than **FD_SETSIZE**.

[EINTR] The **select()** function was interrupted before any of the selected events occurred and before the *timeout* interval expired. If **SA_RESTART** has been set for the interrupting signal, it is implementation-dependent whether **select()** restarts or returns with **EINTR**.

[EINVAL] An invalid timeout interval was specified.

[EINVAL] The *nfds* argument is less than 0, or is greater than or equal to the value of the kernel parameter **MAXFUPLIM**, which specifies the absolute maximum number of files a process can have open at one time.

[EINVAL] One of the specified file descriptors refers o a STREAM or multiplexer that is linked (directly or indirectly) downstream from a multiplexer.

S

APPLICATION USAGE

The use of a timeout does not affect any pending timers set up by **alarm()**, **ualarm()**, or **setitimer()**.

On successful completion, the object pointed to by the *timeout* argument may be modified.

The **FD_SETSIZE** is used in the definition of **fd_set** structure. It is set to a value of 2048 to accommodate 2048 file descriptors. Any user code that uses **FD_SETSIZE** or the structure **fd_set** should redefine **FD_SETSIZE** to a smaller value (greater than or equal to the number of open files the process will have) in order to save space. Similarly, any user code that wants to test more than 2048 file descriptors should redefine **FD_SETSIZE** to the required higher value.

The user can also allocate the space for **fd_set** structure dynamically, depending upon the number of file descriptors to be tested. The following code segment illustrates the basic concepts.

```
int num_of_fds,s;
struct fd_set *f;

/*
 * Set num_of_fds to the required value.
 * User can set it to the maximum possible value the kernel is
 * configured for, by using sysconf(_SC_OPEN_MAX).
```

```
 * Note that, if you are not using these many files, you are
 * wasting too much space.
 */
num_of_fds = sysconf(_SC_OPEN_MAX);
s = sizeof(long);
/*
 * howmany is a macro defined in sys/types.h
 */
f = (struct fd_set *)malloc(s*howmany(num_of_fds, s*8);
/*
 * Use f wherever struct fd_set * is used.
 * It can be used to test num_of_fds file descriptors.
 */
```

SEE ALSO

fcntl(2), poll(2), read(2), write(2), <sys/time.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

S

HP-UX EXTENSIONS

SYNOPSIS

```
#include <time.h>

int select(
     size_t nfds,
     int *readfds,
     int *writefds,
     int *exceptfds,
     const struct timeval *timeout
);
```

DESCRIPTION

This **select()** function prototype is provided for backward compatibility only. For this prototype to be used, <time.h>, instead of <sys/time.h>, must be included and the symbol _XOPEN_SOURCE_EXTENDED must not be defined in the compilation time. Otherwise, the X/Open compliant version will be used.

select() examines the files or devices associated with 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))

Ttys and sockets are ready for reading or writing, respectively, if a **read()** or **write()** would not block for one or more of the following reasons:

- input data is available.
- output data can be accepted.
- an error condition exists, such as a broken pipe, no carrier, or a lost connection.

Sockets select true on reads and/or exceptions if out-of-band data is available.

Pipes are ready for reading if there is any data in the pipe, or if there are no writers left for the pipe. Pipes are ready for writing if there is room for more data in the pipe AND there are one or more readers for the pipe, OR there are no readers left for the pipe. **select()** returns the same results for a pipe whether a file descriptor associated with the read-only end or the write-only end of the pipe is used, since both file descriptors refer to the same underlying pipe. So a **select()** of a read-only file descriptor that is associated with a pipe can return ready to write, even though that particular file descriptor cannot be written to.

ERRORS

[EFAULT] One or more of the pointers was invalid. The reliable detection of this error is implementation dependent.

S

EXAMPLES

The following call to **select()** checks if any of 4 terminals are ready for reading. **select()** times 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 works 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");
```

S

```
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);
```

WARNINGS

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

The file descriptor masks are always modified on return, even if the call returns as the result of a timeout.

DEPENDENCIES

select() supports the following devices and file types:

- pipes
- fifo special files (named pipes)
- all serial devices
- All ITEs (internal terminal emulators) and HP-HIL input devices
- *hpib*(7) special files
- *lan*(7) special files
- *pty*(7) special files
- sockets

AUTHOR

select() was developed by HP and the University of California, Berkeley.

SEE ALSO

fcntl(2), read(2), write(2).

S

NAME

sem_close - close a named POSIX semaphore

SYNOPSIS

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

int sem_close(sem_t *sem);
```

DESCRIPTION

sem_close() is used to close a named semaphore. A successful call to **sem_close()** will do the following: Remove the process's descriptor for the semaphore referenced by the specified **sem_t** structure *sem*. Remove the semaphore referenced by the specified **sem_t** structure *sem*, if the semaphore is marked for removal by a call to **sem_unlink()** and there are no other descriptors referencing this semaphore.

When the process's descriptor for the semaphore referenced by *sem* is removed, subsequent use of this semaphore by this process will fail. Descriptors for named semaphores are also removed by processes on exit. Calling **sem_close()** does not affect other processes referencing the same semaphore.

To use this function, link in the realtime library by specifying **-lrt** on the compiler or linker command line.

EXAMPLES

The following call to **sem_close()** will close a named semaphore referred to by *sem* by removing the process's descriptor to the semaphore and removing the semaphore if it is marked for removal by a previous **sem_unlink()** and there are no descriptors referencing it.

```
sem_close(sem);
```

RETURN VALUE

If the semaphore was closed and the descriptors referencing it were removed, **sem_close()** returns 0 to the caller.

If the semaphore could not be closed, the call returns -1 and sets **errno** to indicate the error.

ERRORS

sem_close() fails and does not perform the requested operation if the following condition is encountered:

[EINVAL] The argument *sem* is not a valid named semaphore.

SEE ALSO

sem_init(2), sem_open(2), sem_unlink(2), <semaphore.h>.

STANDARDS CONFORMANCE

sem_close(): POSIX

S

NAME

sem_destroy - destroy an unnamed POSIX semaphore

SYNOPSIS

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

int sem_destroy(sem_t *sem);
```

DESCRIPTION

sem_destroy() is used to destroy an unnamed semaphore. A successful call to **sem_destroy()** will invalidate the unnamed semaphore referred to by *sem* and removes all descriptors referencing it. The semaphore should have been created by a previous call to **sem_init()** and there should not be any processes blocked on it.

To use this function, link in the realtime library by specifying **-lrt** on the compiler or linker command line.

EXAMPLES

The following call to **sem_destroy()** will destroy an unnamed semaphore referred to by *sem* and remove all descriptors referencing it.

```
sem_destroy(sem);
```

RETURN VALUE

If the semaphore was destroyed and the descriptors referencing it were removed, **sem_destroy()** returns 0 to the caller.

If the semaphore could not be destroyed, the call returns -1 and sets **errno** to indicate the error.

ERRORS

sem_destroy() fails and does not perform the requested operation if any of the following conditions are encountered:

[EBUSY] There are threads currently blocked on the semaphore or there are outstanding locks held on the semaphore.

[EINVAL] The argument *sem* is not a valid unnamed semaphore.

SEE ALSO

sem_init(2), sem_open(2), <semaphore.h>.

STANDARDS CONFORMANCE

sem_destroy(): POSIX

S

NAME

sem_getvalue - get the value of a POSIX semaphore

SYNOPSIS

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

int sem_getvalue(sem_t *sem, int *sval);
```

DESCRIPTION

sem_getvalue() is used to read the value of the semaphore. The value of the semaphore specified by *sem* is read, at some unspecified time during the call, and then stored into *sval*. If the semaphore value is <= 0, at that time, the semaphore is considered unavailable. If the semaphore value is > 0, at that time, the semaphore is considered available.

If *sval* is positive, it is equal to the number of locks available on the semaphore, at the time the semaphore value was read. If *sval* is negative, its absolute value is equal to the number of blocked threads waiting for the semaphore to become available, at the time the semaphore value was read.

If the specified semaphore referred to by *sem* is a named semaphore, then this semaphore must have been opened by the calling process with **sem_open()** and the process must have read permission on this semaphore.

To use this function, link in the realtime library by specifying **-lrt** on the compiler or linker command line.

EXAMPLES

The following call to **sem_getvalue()** will read the value of the semaphore *sem* and store it in *sval*.

```
sem_getvalue(sem,sval);
```

RETURN VALUE

A successful call to **sem_getvalue()** will return 0. Otherwise, the call to **sem_getvalue()** will return -1 with errno set to the appropriate value of the error condition.

ERRORS

sem_getvalue() fails and does not perform the requested operation if any of the following conditions are encountered:

- [EPERM] The calling process does not have the privileges necessary to read the semaphore.
- [EINVAL] The argument *sem* does not refer to a valid semaphore.

SEE ALSO

sem_open(2), <semaphore.h>.

STANDARDS CONFORMANCE

sem_getvalue(): POSIX

S

NAME

sem_init - initialize an unnamed POSIX semaphore

SYNOPSIS

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

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

DESCRIPTION

sem_init() is used to initialize an unnamed semaphore. A successful call to **sem_init()** will create a new unnamed semaphore referred to by *sem*, if one does not exist, initialize the unnamed semaphore descriptor, referred to by *sem*, to the non-negative value specified by *value*. If the unnamed semaphore already exists, i.e. created by a previous call to **sem_init()**, it is re-initialized only if its current value is equal to its initial value (set by the last successful call to **sem_init()**). If so, the initial value of the unnamed semaphore is re-initialized to the value argument. Otherwise, the call fails.

The argument *pshared* specifies if the unnamed semaphore is sharable with other processes. If *pshared* is equal to 0, the unnamed semaphore is not shared with other processes. If *pshared* is non-zero, the unnamed semaphore is sharable with any processes that can access *sem*. The access mode specified for the unnamed semaphore allows read and write permissions to all processes. If the calling process may attach to the shared **sem_t** structure, it is assumed it may operate on the semaphore.

To use this function, link in the realtime library by specifying **-lrt** on the compiler or linker command line.

EXAMPLES

The following call to **sem_init()** will create a new unnamed semaphore referred to by *sem*, if one does not exist, initialize the unnamed semaphore descriptor, referred to by *sem*, to the non-negative value specified by *value*.

```
sem_init(sem, pshared, value);
```

RETURN VALUE

If the semaphore was created and initialized, **sem_init()** returns 0 to the caller.

If the semaphore could not be created/initialized, the call returns -1 and sets **errno** to indicate the error.

ERRORS

sem_init() fails and does not perform the requested operation if any of the following conditions are encountered:

[EPERM]	The calling process does not have the privileges necessary to initialize the semaphore.
[EBUSY]	There are threads currently blocked on the semaphore or there are outstanding locks held on the semaphore.
[EINVAL]	The argument *value* is greater than **{_POSIX_SEM_VALUE_MAX}**.
[ENOSPC]	There are insufficient resources to perform the operation or the upper limit on the number of semaphores is reached.

S

SEE ALSO

sem_destroy(2), sem_post(2), sem_trywait(2), sem_wait(2), <semaphore.h>.

STANDARDS CONFORMANCE

sem_init(): POSIX

NAME

sem_open - create/open a named POSIX semaphore

SYNOPSIS

#include <sys/semaphore.h>

sem_t * sem_open(const char *name, int oflag, mode_t mode, unsigned int value);

DESCRIPTION

sem_open() is used to open or create a named semaphore. A successful call to **sem_open()** will create a descriptor for the semaphore specified by *name*. The pointer to the semaphore returned by **sem_open()** can be used to access the semaphore associated with *name* in subsequent operations. The *name* argument points to a string referring to a semaphore. It should begin with a "/" and shall conform to pathname rules except that no path component should be "." or "..".

The *oflag* argument specifies whether a semaphore is to be created or not. The following bits in it may be set:

O_CREAT If this flag is set, a new semaphore is created if it does not already exist. If this flag is not set, the semaphore should already exist.

O_EXCL If this flag is set, the call fails if the semaphore already exists. This flag is valid only when **O_CREAT** is also set; otherwise, it is ignored.

The *mode* and *value* arguments are provided to supply the permissions and the initial value information necessary for creating a new semaphore.

To use this function, link in the realtime library by specifying **-lrt** on the compiler or linker command line.

EXAMPLES

The following call to **sem_open()** will create a new named semaphore if one does not exist, which depends on the flags specified in *oflag*, has the permissions specified in *mode* and has an initial value of *value*.

```
sem_open(name, oflag, mode, value);
```

RETURN VALUE

If the semaphore was created and initialized, **sem_open()** returns a pointer to a **sem_t** structure containing the index of the new descriptor.

If the semaphore could not be created/initialized, the call returns -1 and sets **errno** to indicate the error. If the named semaphore is already opened by the calling process, a descriptor and a **sem_t** structure for the named semaphore already exists for the calling process. A new descriptor is not created and a pointer to the existing **sem_t** structure is returned for this call.

S

ERRORS

sem_open() fails and does not perform the requested operation if any of the following conditions are encountered:

[EACCES] The named semaphore exists and the process does not have the permissions to open the semaphore as described by *oflag*, or the named semaphore does not exist and the process does not have the permission to open it.

[EEXIST] The flags **O_CREAT** and **O_EXCL** are set in *oflag* and the named semaphore exists.

[EINTR] A signal interrupted the **sem_open()** operation.

[EINVAL] The argument *value* is greater than **{_POSIX_SEM_VALUE_MAX}** and the **O_CREAT** flag was specified in *oflag*.

[EINVAL] The *name* argument does not begin with "/" or contains "." or ".." as a pathname component.

[EMFILE] Too many semaphore descriptors are currently in use by this process.

[ENAMETOOLONG]
The name string is longer than **{PATH_MAX}**.

[ENFILE]	There are too many semaphores in the system.
[ENOENT]	The flag **O_CREAT** is not set in *oflag* and the named semaphore does not exist.
[ENOSPC]	There are insufficient resources for the creation of a new named semaphore.

SEE ALSO

sem_close(2), sem_post(2), sem_wait(2), sem_unlink(2), <semaphore.h>.

STANDARDS CONFORMANCE

sem_open(): POSIX

S

NAME

sem_post - unlock a POSIX semaphore

SYNOPSIS

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

int sem_post(sem_t *sem);
```

DESCRIPTION

sem_post() is used to post the semaphore referenced by *sem*. The calling thread will not return from its call to **sem_post()** unless it can either: increment the semaphore value, if there are no blocked threads on this semaphore; give the semaphore to a blocked thread, if there are any blocked threads on this semaphore; or have an error condition.

If the semaphore value is < 0, the semaphore has blocked threads, waiting for it to become available (the absolute value of the semaphore's value indicates the number of waiters at that moment). If the semaphore value is >= 0, the semaphore has no waiters.

If the semaphore has no waiters at the time its value is checked, the semaphore's value will be atomically incremented, with respect to the checking of its value, up to its maximum value as specified by **{_POSIX_SEM_VALUE_MAX}**. If the semaphore has waiters at the time its value is checked, the semaphore value is not changed. Instead, the calling thread will attempt to wake up a waiter. If the semaphore has waiters having realtime priorities, the thread must wake up the highest priority waiter. Otherwise the thread at the head of the channel queue is woken up.

When a waiter is successfully woken, the semaphore being posted will be given to the woken waiter. In other words, the state of the semaphore remains unchanged. Instead, the semaphore being posted will be inherited by the waiter being woken from this call to **sem_post()**.

If the specified semaphore referred to by *sem* is a named semaphore, then this semaphore must have been opened by the calling process with **sem_open()**. The calling process must have both read and write permissions on the semaphore to perform this operation. The **sem_post()** routine may be called asynchronously, i.e. from a signal handler.

To use this function, link in the realtime library by specifying **-lrt** on the compiler or linker command line.

EXAMPLES

The following call to **sem_post()** will post the semaphore *sem*.

```
sem_post(sem);
```

RETURN VALUE

A successful call to **sem_post()** will return 0 and the calling thread would have posted the semaphore. Otherwise, the call to **sem_post()** will return -1 with errno set to the appropriate value of the error condition.

ERRORS

sem_post() fails and does not perform the requested operation if any of the following conditions are encountered:

[EPERM]	The calling process does not have the privileges necessary to post the semaphore.
[EINVAL]	The argument *sem* does not refer to a valid semaphore.

SEE ALSO

<semaphore.h>.

STANDARDS CONFORMANCE

sem_post(): POSIX

S

NAME

sem_unlink - unlink a named POSIX semaphore

SYNOPSIS

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

int sem_unlink(const char *name);
```

DESCRIPTION

sem_unlink() is used to unlink named semaphores. A successful call to **sem_unlink()** marks the semaphore, specified by *name*, for removal. Calling **sem_unlink()** does not affect processes, including the calling process, which currently have a descriptor, obtained from a call to **sem_open()**. Named semaphores are uniquely identified by character strings. All character string names will be pre-processed to ensure variations of a pathname resolve to the same semaphore name. If the semaphore is successfully marked for removal by a call to **sem_unlink()**, the semaphore will be removed when all processes remove their descriptors to the specified semaphore by calling **sem_close()**. Subsequent calls to **sem_open()** using the string *name* will refer to a new semaphore.

To use this function, link in the realtime library by specifying **-lrt** on the compiler or linker command line.

EXAMPLES

The following call to **sem_unlink()** will remove the named semaphore named by the string *name*. If the semaphore is currently referenced by one or more processes, the semaphore will be marked for removal and removed when there are no more processes referencing it.

```
sem_unlink(name);
```

RETURN VALUE

If the semaphore was unlinked successfully, **sem_unlink()** returns 0.

If the semaphore could not be unlinked, the call returns -1 and sets **errno** to indicate the error.

ERRORS

sem_unlink() fails and does not perform the requested operation if any of the following conditions are encountered:

- [EACCES] The named semaphore exists and the process does not have the permissions to unlink the semaphore.
- [ENAMETOOLONG] The *name* string is longer than **{PATH_MAX}**.
- [ENOENT] The flag **O_CREAT** is not set in oflag (see *sem_open*(2)) and the named semaphore does not exist.

S

SEE ALSO

sem_close(2), sem_open(2), <semaphore.h>.

STANDARDS CONFORMANCE

sem_unlink(): POSIX

NAME

sem_wait, sem_trywait - lock a POSIX semaphore

SYNOPSIS

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

int sem_wait(sem_t *sem);

int sem_trywait(sem_t *sem);
```

DESCRIPTION

sem_wait() is used to lock a semaphore. The calling thread will not return from its call to **sem_wait()** until one of the following events occur: it successfully obtains a lock on the semaphore; it is interrupted by a signal or an error condition occurs.

sem_trywait() is used to lock a semaphore, if it is available. The value of the semaphore *sem* is checked at some unspecified time during the call. If the semaphore is available at the time its value is checked, the calling thread will atomically, with respect to the checking of the value, lock the semaphore. The thread will now own a lock on the semaphore; the call will return successfully. If the semaphore is unavailable at the time its value is checked, then the call returns -1 with errno set to EAGAIN.

If the specified semaphore referred to by *sem* is a named semaphore, then this semaphore must have been opened by the calling process with **sem_open()**. The calling process must have both read and write permissions on the semaphore to perform these operations. The semaphore will be locked upon successful return and will stay locked until it is explicitly released by a call to **sem_post()**.

To use this function, link in the realtime library by specifying **-lrt** on the compiler or linker command line.

EXAMPLES

The following call to **sem_wait()** will lock the semaphore *sem*.

```
sem_wait(sem);
```

The following call to **sem_trywait()** will lock the semaphore *sem*, if it is available.

```
sem_trywait(sem);
```

RETURN VALUE

A successful call to **sem_wait()** will return 0 and the calling thread will then own a lock on the semaphore. Otherwise, the call to **sem_wait()** will return -1 with errno set to the appropriate value of the error condition.

A successful call to **sem_trywait()** will return 0, if the semaphore was available and the calling thread was able to lock the semaphore. Otherwise, the call to **sem_trywait()** will return -1 with errno set to the appropriate value of the error condition.

ERRORS

sem_wait() and **sem_trywait()** fail and do not perform the requested operation if any of the following conditions are encountered:

[EPERM]	The calling process does not have the privileges necessary to lock the semaphore.
[EAGAIN]	The semaphore was not available and hence could not be locked by **sem_trywait()**. This error condition only occurs in **sem_trywait()**.
[EINVAL]	The argument *sem* does not refer to a valid semaphore.
[EINTR]	The function was interrupted by a signal

SEE ALSO

sem_post(2), <semaphore.h>.

STANDARDS CONFORMANCE

sem_wait(),**sem_trywait()**: POSIX

S

NAME

semctl - semaphore control operations

SYNOPSIS

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

int semctl(int semid,
    int semnum,
    int cmd,
    union arg
);

union semun {
     int val;
     struct semid_ds *buf;
     ushort *array;
} arg;
```

DESCRIPTION

The **semctl()** system call provides a variety of semaphore control operations as specified by *cmd*. For the meaning of unspecified variables, see *semaphore identifier* in *glossary*(9).

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

GETVAL Return the value of *semval*. Requires semaphore Read permission.

SETVAL Set the value of *semval* to *arg.val*, where *arg* is the fourth argument of **semctl()** taken as an **int**. When this *cmd* is successfully executed, the *semadj* value corresponding to the specified semaphore in all processes is cleared. Requires semaphore Alter permission.

GETPID Return the value of *sempid*. Requires semaphore Read permission.

GETNCNT Return the value of *semncnt*. Requires semaphore Read permission.

GETZCNT Return the value of *semzcnt*. Requires semaphore Read permission.

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

GETALL Place *semval*s into array pointed to by *arg.array*, where *arg* is the fourth argument of **semctl()** taken as a pointer to **unsigned short int**. Requires semaphore Read permission.

SETALL Set *semval*s according to the array pointed to by *arg.array*, where *arg* is the fourth argument of **semctl()** taken as a pointer to **unsigned short int**. When this *cmd* is successfully executed, the *semadj* values corresponding to each specified semaphore in all processes are cleared. Requires semaphore Alter permission.

S

The following values for *cmd* 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*, where *arg* is the fourth argument of **semctl()** taken as a pointer to **struct semid_ds**. The contents of this structure are defined in *glossary*(9). Requires semaphore Read permission.

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*, where *arg* is the fourth argument of **semctl()** taken as a pointer to **struct semid_ds**:

```
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 superuser 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 superuser or to the value of either **sem_perm.uid** or **sem_perm.cuid** in the data structure associated with *semid*.

RETURN VALUE

Upon successful completion, **semctl()** returns a value based 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 return **0**.

If it fails, it returns **-1** and sets **errno** to indicate the error.

ERRORS

If **semctl()** fails, it sets **errno** to one of the following values:

[EACCES] Operation permission is denied to the calling process (see *semaphore operation permissions* in *glossary*(9).

[EFAULT] *cmd* is **SETVAL**, **GETALL**, **SETALL**, **IPC_SET**, or **IPC_STAT**, and *arg* is an invalid pointer.

[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, or the command contains invalid parameters.

[EPERM] *cmd* is equal to **IPC_RMID** or **IPC_SET** and the process does not have an effective user ID equal to either that of superuser or to the value of either **sem_perm.uid** or **sem_perm.cuid** in the data structure associated with *semid*.

[ERANGE] *cmd* is **SETVAL** or **SETALL** and the value to which *semval* is to be set is greater than the system imposed maximum.

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 in the *semget*(2) manual entry. For an example of performing "P" and "V" operations on the semaphores below, refer to *semop*(2).

```
union semun {
     int val;
     struct semid_ds *buf;
     ushort *array;
} arg;

ushort semarray[4];

semarray[0] = 0;
semarray[1] = 1;
semarray[2] = 0;
semarray[3] = 1;

arg.array = &semarray[0];
semctl (mysemid, 0, SETALL, arg);
```

SEE ALSO

ipcrm(1), ipcs(1), semget(2), semop(2), stdipc(3C), glossary(9).

STANDARDS CONFORMANCE

semctl(): SVID2, SVID3, XPG2, XPG3, XPG4

S

NAME

semget - get set of semaphores

SYNOPSIS

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

int semget(key_t key, int nsems, int 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 is never 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 can be requested by ORing 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 *glossary*(9).

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);
```

S

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.

ERRORS

semget() fails if one or more of the following is 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.

[EEXIST] A semaphore identifier exists for *key* but **((***semflg* **& IPC_CREAT) && (***semflg* **& IPC_EXCL))** is "true".

SEE ALSO

ipcrm(1), ipcs(1), semctl(2), semop(2), stdipc(3C).

STANDARDS CONFORMANCE

semget(): SVID2, SVID3, XPG2, XPG3, XPG4

S

NAME

semop - semaphore operations

SYNOPSIS

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

int semop(
     int semid,
     struct sembuf *sops,
     size_t 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 are 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 occurs:

If *semval* (see *semaphore identifier* in *glossary*(9)) 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 *glossary*(9) 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()** returns immediately.

If *semval* is less than the absolute value of *sem_op* and (*sem_flg* & **IPC_NOWAIT**) is "false", **semop()** increments the semncnt associated with the specified semaphore and suspend execution of the calling process until one of the following conditions 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 occurs:

If *semval* is zero, **semop()** proceeds to the next semaphore operation specified by *sops*, or returns immediately if this is the last operation.

If *semval* is not equal to zero and (*sem_flg* & **IPC_NOWAIT**) is "true", **semop()** returns immediately.

If *semval* is not equal to zero and (*sem_flg* & **IPC_NOWAIT**) is "false", **semop()** increments the semzcnt associated with the specified semaphore and suspends execution of the

S

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 in the *semctl*(2) manual entry.

```
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);
```

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.

ERRORS

semop() fails if one or more of the following is 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 *glossary*(9)).

[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.

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, SVID3, XPG2, XPG3, XPG4

NAME

send(), sendmsg(), sendto() - send a message from a socket

SYNOPSIS

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

int send(int s, const void *msg, int len, int flags);

int sendto(
    int          s,
    const void *msg,
    int          len,
    int          flags,
    const void *to,
    int          tolen
);

int sendmsg(int s, const struct msghdr msg[], int flags);
```

_XOPEN_SOURCE_EXTENDED Only (UNIX 98)

```
ssize_t send(int s, const void *msg, size_t len, int flags);

ssize_t sendto(
        int                      s,
        const void              *msg,
        size_t                   len,
        int                      flags,
        const struct sockaddr   *to,
        socklen_t                tolen
);

ssize_t sendmsg(int s, const struct msghdr *msg, int flags);
```

Obsolescent _XOPEN_SOURCE_EXTENDED Only (UNIX 95)

```
ssize_t sendto(
        int                      s,
        const void              *msg,
        size_t                   len,
        int                      flags,
        const struct sockaddr   *to,
        size_t                   tolen
);
```

DESCRIPTION

The **send()**, **sendmsg()**, and **sendto()** system calls transmit a message to another socket. **send()** can be used only when the socket is in a connected state, whereas **sendmsg()** and **sendto()** can be used at any time. **sendmsg()** allows the send data to be gathered from several buffers specified in the **msghdr** structure.

S

s is a socket descriptor that specifies the socket on which the message will be sent.

msg points to the buffer containing the message.

If the socket uses connection-based communications, such as a SOCK_STREAM socket, these calls can only be used after the connection has been established (see *connect*(2)). In this case, any destination specified by *to* is ignored. For connectionless sockets, such as SOCK_DGRAM, **sendto()** must be used unless the destination address has already been specified by **connect()**. If the destination address has been specified and **sendto()** is used, an error results if any address is specified by *to*.

The address of the target socket is contained in a socket address structure pointed to by *to*, with *tolen* specifying the size of the structure.

If a **sendto()** is attempted on a SOCK_DGRAM socket before any local address has been bound to it, the system automatically selects a local address to be used for the message. In this case, there is no guarantee that the same local address will be used for successive **sendto()** requests on the same socket.

The length of the message is given by *len* in bytes. The length of data actually sent is returned. If the message is too long to pass atomically through the underlying protocol, the message is not transmitted, **−1** is returned, and **errno** is set to [EMSGSIZE]. For SOCK_DGRAM sockets, this size is fixed by the implementation (see the DEPENDENCIES section). Otherwise there is no size limit.

When **send()** or **sendto()** returns a positive value, it only indicates this number of bytes have been sent to the local transport provider. It does not mean this number of bytes have been delivered to the peer socket application. A SOCK_DGRAM socket does not guarantee end-to-end delivery. A SOCK_STREAM socket guarantees eventual end-to-end delivery, however its underlying transport provider may later detect an irrecoverable error and returns a value of **−1** at another socket function call.

When send() or **sendto()** returns a value of **−1** , it indicates a locally detected error. **errno** is set to indicate the error.

sendmsg() performs the same action as **send()**, but it gathers the output data from the buffers specified in the **msghdr** structure (see **_XOPEN_SOURCE_EXTENDED Only** below). This structure is defined in **<sys/socket.h>** and has the following form (**HP-UX BSD Sockets Only**):

```
struct msghdr {
    caddr_t   msg_name;              /* optional address */
    int       msg_namelen;           /* size of address  */
    struct    iovec *msg_iov;        /* scatter array for data */
    int       msg_iovlen;            /* # of elements in msg_iov */
    caddr_t   msg_accrights;         /* access rights */
    int       msg_accrightslen;      /* size of msg_accrights */
}
```

msg_name points to a **sockaddr** structure in which the address of the destination socket should be stored, if the socket is connectionless; *msg_name* may be a null pointer if no name is specified. *msg_iov* specifies the locations of the character arrays for storing the outbound data. *msg_accrights* specifies a buffer that contains any access rights to be sent along with the message. Access rights are limited to file descriptors of size *int*. If access rights are not being transferred, set the *msg_accrights* field to NULL. Access rights are supported only for AF_UNIX.

If no buffer space is available to hold the data to be transmitted, **send()** blocks unless nonblocking mode is enabled. The three ways to enable nonblocking mode are:

- with the **FIOSNBIO ioctl()** request,
- with the **O_NONBLOCK** flag, and
- with the **O_NDELAY fcntl()** flag.

If nonblocking I/O is enabled using **FIOSNBIO** or the equivalent **FIONBIO** request (defined in **<sys/ioctl.h>** and explained in *ioctl*(2), *ioctl*(5), and *socket*(7)), although the use of **FIONBIO** is not recommended, the **send()** request completes in one of three ways:

- If there is enough space available in the system to buffer all of the data, **send()** completes successfully, having written out all of the data, and returns the number of bytes written.
- If there is not enough space in the buffer to write out the entire request, **send()** completes successfully, having written as much data as possible, and returns the number of bytes it was able to write.
- If there is no space in the system to buffer any of the data, **send()** fails, having written no data, and **errno** is set to [EWOULDBLOCK].

If nonblocking I/O is disabled using **FIOSNBIO**, **send()** always executes completely (blocking as necessary) and returns the number of bytes written.

If the **O_NONBLOCK** flag is set using **fcntl()** (defined in **<sys/fcntl.h>** and explained in *fcntl*(2) and *fcntl*(5)), POSIX-style nonblocking I/O is enabled. In this case, the **send()** request completes in one of three ways:

- If there is enough space available in the system to buffer all of the data, **send()** completes successfully, having written out all of the data, and returns the number of bytes written.
- If there is not enough space in the buffer to write out the entire request, **send()** completes successfully, having written as much data as possible, and returns the number of bytes it was able to write.
- If there is no space in the system to buffer any of the data, **send()** completes, having written no data, and returns –1, with **errno** set to [EAGAIN].

S

If the **O_NDELAY** flag is set using **fcntl()** (defined in **<sys/fcntl.h>** and explained in *fcntl*(2) and *fcntl*(5)), nonblocking I/O is enabled. In this case, the **send()** request completes in one of three ways:

- If there is enough space available in the system to buffer all of the data, **send()** completes successfully, having written out all of the data, and returns the number of bytes written.
- If there is not enough space in the buffer to write out the entire request, **send()** completes successfully, having written as much data as possible, and returns the number of bytes it was able to write.
- If there is no space in the system to buffer any of the data, **send()** completes successfully, having written no data, and returns 0.

If the **O_NDELAY** flag is cleared using **fcntl()**, nonblocking I/O is disabled. In this case, the **send()** always executes completely (blocking as necessary) and returns the number of bytes written.

Since the **fcntl() O_NONBLOCK** and **O_NDELAY** flags and **ioctl() FIOSNBIO** requests are supported, the following clarifies on how these features interact. If the **O_NONBLOCK** or **O_NDELAY** flag has been set, **send()** requests behave accordingly, regardless of any **FIOSNBIO** requests. If neither the **O_NONBLOCK** flag nor the **O_NDELAY** flag has been set, **FIOSNBIO** requests control the behavior of **send()**.

By default nonblocking I/O is disabled.

The supported values for *flags* are zero or **MSG_OOB** (to send out-of-band data). A **write()** call made to a socket behaves in exactly the same way as **send()** with *flags* set to zero. **MSG_OOB** is not supported for AF_UNIX or AF_VME_LINK sockets.

select(2) can be used to determine when it is possible to send more data.

AF_CCITT Only

Sockets of the address family AF_CCITT operate in message mode. Although they are specified as connection-based (SOCK_STREAM) sockets, the X.25 subsystem communicates via messages. They require that a connection be established with the **connect()** or **accept()** calls.

The **O_NDELAY** flag is not supported. Use **FIOSNBIO** requests to control nonblocking I/O. If the available buffer space is not large enough for the entire message and the socket is in nonblocking mode, **errno** is set to [EWOULDBLOCK]. If the amount of data in the **send()** exceeds the maximum outbound message size, **errno** is set to [EMSGSIZE].

The **sendto()** call is not supported.

Each call sends either a complete or a partial X.25 message. This is controlled by the setting of the More-Data-To-Follow (MDTF) bit. If the user wants to send a partial message, MDTF should be set to 1 before the **send()** call. The MDTF bit should be cleared to 0 before sending the final message fragment.

Message fragment length may range from 0 bytes up to the size of the socket's send buffer (see *af_ccitt*(7F)). The MDTF bit and multiple **send()** calls can be combined to transmit complete X.25 packet sequences (i.e., zero or more DATA packets in which the More Data bit is set, followed by one DATA packet in which the More Data bit is clear) of arbitrary length. Note that a 0-byte message is not actually sent, but may be necessary to flush a complete X.25 message if the user is controlling the MDTF bit.

Sockets of the AF_CCITT address family can send 1 byte of out-of-band data (known as an INTERRUPT data packet in X.25 terminology), or up to 32 bytes if the X.25 interface is configured for 1984 CCITT X.25 recommendations. INTERRUPT data packets sent in blocking mode cause the process to block until confirmation is received. INTERRUPT data packets sent with the socket in nonblocking mode do not cause the process to block; instead, an out-of-band message is queued to the socket when the INTERRUPT confirmation packet is received (see *recv*(2)).

_XOPEN_SOURCE_EXTENDED Only

For **X/Open Sockets**, the **msghdr** structure has the following form:

```
(UNIX 98)

struct msghdr {
    void       *msg_name;          /* optional address */
    socklen_t   msg_namelen;       /* size of address  */
    struct      iovec *msg_iov;    /* scatter array for data */
    int         msg_iovlen;        /* # of elements in msg_iov */
```

S

```
        void        *msg_control;         /* ancillary data, see below */
        socklen_t    msg_controllen;      /* ancillary data buffer len */
        int          msg_flags;           /* flags on received message */
    }
```

Obsolescent (UNIX 95)

```
    struct msghdr {
        void        *msg_name;            /* optional address */
        size_t       msg_namelen;         /* size of address  */
        struct       iovec *msg_iov;      /* scatter array for data */
        int          msg_iovlen;          /* # of elements in msg_iov */
        void        *msg_control;         /* ancillary data, see below */
        size_t       msg_controllen;      /* ancillary data buffer len */
        int          msg_flags;           /* flags on received message */
    }
```

msg_control specifies a buffer of ancillary data to send along with the message. Ancillary data consists of a sequence of pairs, each consisting of a *cmsghdr* structure followed by a data array. The data array contains the ancillary data message, and the *cmsghdr* structure contains descriptive information that allows an application to correctly parse the data. *cmsghdr* has the following structure:

(UNIX 98)

```
    struct cmsghdr {
        socklen_t  cmsg_len;      /* data byte count, including hdr*/
        int        cmsg_level;    /* originating protocol */
        int        cmsg_type;     /* protocol-specific type */
    }
```

Obsolescent (UNIX 95)

```
    struct cmsghdr {
        size_t     cmsg_len;      /* data byte count, including hdr*/
        int        cmsg_level;    /* originating protocol */
        int        cmsg_type;     /* protocol-specific type */
    }
```

The supported value for cmsg_level is SOL_SOCKET, and the supported value for cmsg_type is SCM_RIGHTS. Together they indicate the data array contains the access rights to be sent. Access rights are supported only for AF_UNIX. Access rights are limited to file descriptors of size *int*. If ancillary data are not being transferred, set the *msg_control* field to NULL, and set the *msg_controllen* field to 0.

The *msg_flags* member is ignored.

S

RETURN VALUE

send(), **sendmsg()**, and **sendto()** return the following values:

- *n* Successful completion. *n* is the number of bytes sent.
- **−1** Failure. **errno** is set to indicate the error.

ERRORS

If **send()**, **sendmsg()**, or **sendto()** fails, **errno** is set to one of the following values.

[EACCES]	Process doing a **send()** of a broadcast packet does not have broadcast capability enabled for the socket. Use **setsockopt()** to enable broadcast capability.
[EAFNOSUPPORT]	The specified address is not a valid address for the address family of this socket.
[EAGAIN]	Nonblocking I/O is enabled using the **O_NONBLOCK** flag with **fcntl()**, and the requested operation would block, or the socket has an error that was set asynchronously. An asynchronous error can be caused by a gateway failing to forward a datagram from this socket because the datagram exceeds the MTU of the next-hop network and the "Don't Fragment" (DF) bit in the datagram is set. (See **SO_PMTU** in *getsockopt*(2)).

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

[ECONNRESET] A connection was forcibly closed by a peer.

[EDESTADDRREQ] The *to* parameter needs to specify a destination address for the message. This is also given if the specified address contains unspecified fields (see *inet*(7F)).

[EFAULT] An invalid pointer was specified in the *msg* or *to* parameter, or in the **msghdr** structure.

[EINTR] The operation was interrupted by a signal before any data was sent. (If some data was sent, **send()** returns the number of bytes sent before the signal, and [EINTR] is not set).

[EINVAL] The *len* or *tolen* parameter, or a length in the **msghdr** structure is invalid. A **sendto()** system call was issued on an X.25 socket, or the connection is in its reset sequence and cannot accept data.

[EIO] A timeout occurred.

[EISCONN] An address was specified by *to* for a SOCK_DGRAM socket which is already connected.

[EMSGSIZE] A length in the **msghdr** structure is invalid. The socket requires that messages be sent atomically, and the size of the message to be sent made this impossible.

SOCK_DGRAM/AF_INET or SOCK_STREAM/AF_CCITT: The message size exceeded the outbound buffer size.

[ENETDOWN] The interface used for the specified address is "down" (see *ifconfig*(1M)), no interface for the specified address can be found (SO_DONTROUTE socket option in use), or the X.25 Level 2 is down.

[EHOSTUNREACH]
The destination host is not reachable.

[ENETUNREACH] The destination network is not reachable. Some of the possible causes for this error are:

(LAN) All encapsulations (e.g., ether, ieee) have been turned off (see also *ifconfig*(1M)).

(X.25) The X.25 Level 2 is down. The X.25 link layer is not working (wires might be broken, connections are loose on the interface hoods at the modem, the modem failed, the packet switch at the remote end lost power or failed for some reason, or electrical noise interfered with the line for an extremely long period of time).

[ENOBUFS] No buffer space is available in the system to perform the operation.

[ENOMEM] No memory is available in the system to perform the operation.

[ENOTCONN] A **send()** on a socket that is not connected, or a **send()** on a socket that has not completed the connect sequence with its peer, or is no longer connected to its peer.

[ENOTSOCK] *s* is a valid file descriptor, but it is not a socket.

[EOPNOTSUPP] The **MSG_OOB** flag was specified; it is not supported for AF_UNIX or AF_VME_LINK sockets.

[EPIPE] and **SIGPIPE** signal
An attempt was made to send on a socket that was connected, but the connection has been shut down either by the remote peer or by this side of the connection. Note that the default action for **SIGPIPE**, unless the process has established a signal handler for this signal, is to terminate the process.

[EWOULDBLOCK] Nonblocking I/O is enabled using **ioctl() FIOSNBIO** request and the requested operation would block.

DEPENDENCIES

UDP messages are fragmented at the IP level into Maximum Transmission Unit (MTU) sized pieces; MTU varies for different link types. These pieces, called IP fragments, can be transmitted, but IP does not

S

guarantee delivery. Sending large messages may cause too many fragments and overrun a receiver's ability to receive them. If this happens the complete message cannot be reassembled. This affects the apparent reliability and throughput of the network as viewed by the end user.

The default and maximum buffer sizes are protocol-specific. Refer to the appropriate entries in Sections 7F and 7P for details. The buffer size can be set by calling **setsockopt()** with **SO_SNDBUF**.

AF_CCITT

If the receiving process is on a Series 700/800 HP-UX system, and the connection has been set up to use the D-bit, data sent with the D-bit set is acknowledged when the receiving process has read the data. Otherwise, the acknowledgement is sent when the firmware receives it.

OBSOLESCENCE

Currently, the **socklen_t** and **size_t** types are the same size. This is compatible with both the UNIX 95 and UNIX 98 profiles. However, in a future release, **socklen_t** might be a different size. In that case, the size of the **msghdr** and **cmsghdr** structures and the relative position of their members will be different, which might affect application behavior. Applications that use **socklen_t** now, where appropriate, will avoid such migration problems. On the other hand, applications that need to be portable to the UNIX 95 profile should follow the X/Open specification (see *xopen_networking*(7)).

FUTURE DIRECTION

Currently, the default behavior is the **HP-UX BSD Sockets**; however, it might be changed to **X/Open Sockets** in a future release. At that time, any **HP-UX BSD Sockets** behavior that is incompatible with **X/Open Sockets** might be obsoleted. Applications that conform to the X/Open specification now will avoid migration problems (see *xopen_networking*(7)).

MULTITHREAD USAGE

The **send()**, **sendmsg()**, and **sendto()** system calls are thread-safe. They each have a cancellation point; and they are async-cancel safe, async-signal safe, and fork-safe.

AUTHOR

send(), **sendmsg()**, and **sendto()** were developed by HP and the University of California, Berkeley.

SEE ALSO

ifconfig(1M), getsockopt(2), recv(2), select(2), setsockopt(2), socket(2), socket(7), socketx25(7), af_ccitt(7F), af_vme_link(7F), inet(7F), tcp(7P), udp(7P), unix(7P), xopen_networking(7).

STANDARDS CONFORMANCE

send(): XPG4

NAME

sendfile() - send the contents of a file through a socket

SYNOPSIS

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

ssize_t sendfile(int s, int fd, off_t offset, size_t nbytes,
        const struct iovec *hdtrl, int flags);
```

DESCRIPTION

The **sendfile()** system call transmits the contents of a file associated with the file descriptor *fd* , plus an optional header and trailer buffers across a socket connection specified by *s* . **sendfile()** can be used only when the socket is in a connected state.

offset specifies the offset within the file at which to start the file data transfer.

nbytes is the number of bytes to be sent from the file. If this parameter is set to zero, data from the *offset* to the end of the file will be sent.

hdtrl points to a two entry **iovec** structure. See *write*(2) for a description of the **iovec** structure. The first entry is for header information. If this pointer is non-NULL, the contents of the buffer are sent before sending any data from the file. The second entry is for trailer information. If this pointer is non-NULL, the contents of the buffer will be sent after the data from the file. If both pointers are NULL, or *hdtrl* is a NULL pointer, only the specified range of the file will be transferred.

At the end of the call, the socket connection will be left completely open for both reading and writing, unless the *flags* parameter is set to:

SF_DISCONNECT Disallow further sends and receives.

RETURN VALUE

Upon successful completion, **sendfile()** returns the number of bytes sent. This includes the header, trailer, and the file contents. Otherwise, **−1** is returned and **errno** is set to indicate the error.

If no buffer space is available to hold the data to be transmitted, **sendfile()** blocks unless nonblocking mode is enabled. See *send(2)* for a description of the nonblocking mode behavior.

ERRORS

If **sendfile()** fails, **errno** is set to one of the following values.

[EBADF] An invalid socket descriptor *s* , or file descriptor *fd* is specified.

[ENOTSOCK] *s* is a valid file descriptor, but it is not a socket.

[EFAULT] An invalid pointer was specified in the *hdtrl* parameter or the **iovec** structure.

[ENOBUFS] No buffer space is available in the system to perform the operation.

[EINTR] The operation was interrupted by a signal before any data was sent. (If some data was sent, **sendfile()** returns the number of bytes sent before the signal, and [EINTR] is not set).

[EINVAL] The *offset* or *flags* parameter is invalid.

The *hdtrl* parameter, or a length in the **iovec** structure is invalid.

[ENOTCONN] A **sendfile()** on a socket that is not connected, or a **sendfile()** on a socket that has not completed the connect sequence with its peer, or is no longer connected to its peer.

[EPIPE] With **SIGPIPE** signal. An attempt was made to send on a socket that was connected, but the connection has been shut down either by the remote peer or by this side of the connection. Note that the default action for **SIGPIPE**, unless the process has established a signal handler for this signal, is to terminate the process.

[EAGAIN] Nonblocking I/O is enabled using the **O_NONBLOCK** flag with **fcntl()**, and the requested operation would block.

[EWOULDBLOCK] The socket is marked non-blocking and the requested operation would block.

[ENOMEM] No memory is available in the system to perform the operation.

S

[EOPNOTSUPP] The socket is not a TCP socket.

SEE ALSO

send(2), read(2), write(2), socket(2), connect(2), shutdown(2), tcp(7P).

NAME

serialize() - force target process to run serially with other processes

SYNOPSIS

```
#include <unistd.h>

int serialize(int timeshare, pid_t pid);
```

DESCRIPTION

The **serialize()** system call is used to force the target process referenced by the *pid* value passed in to run serially with other processes also marked for serialization. If the value of *pid* is zero, then the currently running process is marked for serialization. Once a process has been marked by **serialize()**, the process stays marked until process completion, unless **serialize()** is reissued on the serialized process with *timeshare* set to 1. If *timeshare* is set to 1, the process specified in **pid** will be returned to normal timeshare scheduling algorithms.

This call is used to improve process throughput since process throughput usually increases for large processes when they are executed serially instead of allowing each program to run for only a short period of time. By running large processes one at a time, the system makes more efficient use of the CPU as well as system memory, since each process does not end up constantly faulting in its working set, to only have the pages stolen when another process starts running. As long as there is enough memory in the system, processes marked by **serialize()** behave no differently from other processes in the system. However, once memory becomes tight, processes marked by **serialize()** are run one at a time with the highest priority processes being run first. Each process runs for a finite interval of time before another serialized process is allowed to run.

RETURN VALUE

serialize() returns zero upon successful completion, or nonzero if the system call failed.

ERRORS

If **serialize()** fails, it sets **errno** (see *errno*(2)) to the following value:

[ESRCH] The *pid* passed in does not exist.

WARNINGS

The user has no way of forcing an execution order on serialized processes.

AUTHOR

serialize() was developed by HP.

SEE ALSO

serialize(1).

S

NAME

setacl, fsetacl - set access control list (ACL) information

SYNOPSIS

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

int setacl(
     const char *path,
     int nentries,
     const struct acl_entry *acl
);

int fsetacl(
     int fildes,
     int nentries,
     const struct acl_entry *acl
);
```

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 super-user 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. *nentries* indicates how many valid entries are defined in the *acl* parameter. If *nentries* 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.

S

Altering a base ACL entry's modes with **setacl()** changes the file's corresponding permission bits. The permission bits can be altered also by using **chmod()** (see *chmod*(2)) and read using **stat()** (see *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, they return −1, the file's ACL is not modified, and **errno** is set to indicate the error.

ERRORS

setacl() and **fsetacl()** fail if any of the following conditions are encountered:

[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 super-user.
[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]	ACLs are only supported on HFS file systems. Additionally, **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.
[EDQUOT]	User's disk quota block or inode limit has been reached for this file system.

EXAMPLES

The following code fragment defines and sets an ACL on file **../shared** which allows the file's owner to read, write, and execute or search the file, and allows 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

NFS

setacl() and **fsetacl()** are not supported on remote files.

HFS

ACLs are only supported on HFS file systems.

AUTHOR

setacl() and **fsetacl()** were developed by HP.

SEE ALSO

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

NAME

setaudid - set the audit ID (aid) for the current process

SYNOPSIS

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

int setaudid(aid_t audid);
```

DESCRIPTION

setaudid() sets the audit ID (*aid*) for the current process. This call is restricted to the super-user.

RETURN VALUE

Upon successful completion, **setaudid()** returns a value of 0; otherwise, it returns −1 and sets **errno** to indicate the error.

ERRORS

setaudid() fails if any of the following conditions are encountered:

[EPERM] The caller is not a superuser.

[EINVAL] The audit ID (audid) is invalid.

AUTHOR

setaudid() was developed by HP.

SEE ALSO

getaudid(2).

S

NAME

setaudproc - controls process level auditing for the current process and its decendents

SYNOPSIS

`#include <sys/audit.h>`

`int setaudproc(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 super-users.

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 propagates 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 descendents* are 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()`** should be used (see *audswitch*(2)) 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, **`setaudproc()`** returns 0; otherwise, it returns –1 and sets **`errno`** to indicate the error.

AUTHOR

`setaudproc()` was developed by HP.

SEE ALSO

getaudproc(2), audswitch(2), audusr(1M), audevent(1M), audit(5).

S

NAME

setevent - set current events and system calls which are to be audited

SYNOPSIS

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

int setevent(
     const struct aud_type a_syscall[],
     const 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 super-user.

RETURN VALUE

Upon successful completion, **setevent()** returns 0; otherwise, it returns −1 and sets **errno** to indicate the error.

ERRORS

setevent() fails if the following condition is encountered:

[EPERM] The caller is not super-user.

AUTHOR

setevent() was developed by HP.

SEE ALSO

getevent(2), audevent(1M).

S

NAME

setgroups - set group access list

SYNOPSIS

```
#include <unistd.h>

int setgroups(int ngroups, const 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_MAX**, as defined in **<limits.h>**.

Only super-user can set new groups by adding to the group access list of the current user process; any user can delete groups from it.

RETURN VALUE

Upon successful completion, **setgroups()** returns 0; otherwise it returns –1 and sets **errno** to indicate the error.

ERRORS

setgroups() fails if any of the following conditions are encountered:

[EPERM] The caller is not super-user 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 is implementation dependent.

[EINVAL] *ngroups* is greater than NGROUPS_MAX or not positive.

[EINVAL] An entry in *gidset* is not a valid group ID.

AUTHOR

setgroups() was developed by the University of California, Berkeley.

SEE ALSO

getgroups(2), initgroups(3C).

STANDARDS CONFORMANCE

setgroups() : AES, SVID3

NAME

sethostname - set name of host cpu

SYNOPSIS

```
#include <unistd.h>

int sethostname(const char *name, size_t namelen);
```

DESCRIPTION

The **sethostname()** system call sets the name of the host processor to *name*, which has a length of *namelen* characters. At system boot time **sethostname()** is normally executed by the **hostname** command (see *hostname*(1)) in the **/sbin/init.d/hostname** script. Host names are limited to **MAXHOSTNAMELEN** characters, as defined in **<sys/param.h>**.

RETURN VALUE

sethostname() returns the following values:

- **0** Successful completion.
- **-1** Failure. **errno** is set to indicate the error.

ERRORS

If **sethostname()** fails, **errno** is set to one of the following values.

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

[EPERM] The user does not have appropriate privileges.

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 <unistd.h>

int setpgid(pid_t pid, pid_t pgid);

int setpgrp2(pid_t pid, pid_t pgid);
```

DESCRIPTION

The **setpgid()** and **setpgrp2()** system calls cause 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.

RETURN VALUE

setpgid() and **setpgrp2()** return the following values:

- **0** Successful completion.
- **-1** Failure. **errno** is set to indicate the error.

ERRORS

If **setpgid()** or **setpgrp2()** fails, **errno** is set to one of the following values.

[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.

AUTHOR

setpgid() and **setpgrp2()** were developed by HP and the University of California, Berkeley.

SEE ALSO

bsdproc(3C), exec(2), exit(2), fork(2), getpid(2), kill(2), setsid(2), signal(2), termio(7).

STANDARDS CONFORMANCE

setpgid(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

setpgrp - set process group ID

SYNOPSIS

```
#include <unistd.h>

pid_t setpgrp(void);
```

DESCRIPTION

If the calling process is not already a session leader, **setpgrp()** sets the process group ID of the calling process to the process ID of the calling process. If **setpgrp()** creates a new session, then the new session has no controlling terminal.

The **setpgrp()** function has no effect when the calling process is a session leader.

RETURN VALUE

Upon successful completion, **setpgrp()** returns the new process group ID.

ERRORS

No errors are defined.

SEE ALSO

exec(2), fork(2), getpid(2), getsid(2), kill(2), setsid(2), <unistd.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

S

NAME

setresuid, setresgid - set real, effective, and saved user and group IDs

SYNOPSIS

```
#include <unistd.h>

int setresuid(uid_t ruid, uid_t euid, uid_t suid);

int setresgid(gid_t rgid, gid_t egid, gid_t 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 that of a user having appropriate privileges, **setresuid()** sets the real, effective and saved user IDs to *ruid*, *euid*, and *suid*, respectively. Otherwise, **setresuid()** only sets 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()** leaves 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 that of a user having appropriate privileges, **setresgid()** sets the real, effective, and saved group ID to *rgid*, *egid*, and *sgid*, respectively. Otherwise, **setresgid()** only sets the real, effective and saved group ID if *rgid*, *egid*, and *sgid* each match at least one of the current real, effective or saved group ID.

If *rgid*, *egid*, or *sgid* is **-1**, **setresgid()** leaves the current real, effective or saved group ID unchanged.

RETURN VALUE

Upon successful completion, **setresuid()** and **setresgid()** return 0; otherwise, they return −1 and set **errno** to indicate the error.

ERRORS

setresuid() and **setresgid()** fail if any of the following conditions are encountered:

[EINVAL] *ruid*, *euid*, or *suid* (*rgid*, *egid*, or *sgid*) is not a valid user (group) ID.

[EPERM] None of the conditions above are met.

AUTHOR

setresuid() and **setresgid()** were developed by HP.

SEE ALSO

exec(2), getuid(2), setuid(2).

S

NAME

setreuid - set real and effective user IDs

SYNOPSIS

```
#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);
```

DESCRIPTION

The **setreuid()** function sets the real and effective user IDs of the current process to the values specified by the *ruid* and *euid* arguments. If *ruid* or *euid* is –1, the corresponding effective or real user ID of the current process is left unchanged.

A process with appropriate privileges can set either ID to any value. An unprivileged process can only set the effective user ID if the *euid* argument is equal to either the real, effective, or saved user ID of the process.

It is unspecified whether a process without appropriate privileges is permitted to change the real user ID to match the current real, effective or saved user ID of the process.

RETURN VALUE

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

ERRORS

The **setreuid()** function will fail if:

[EINVAL] The value of the *ruid* or *euid* argument is invalid or out-of-range.

[EPERM] The current process does not have appropriate privileges, and either an attempt was made to change the effective user ID to a value other than the real user ID or the saved set-user-ID or an attempt was made to change the real user ID to a value not permitted by the implementation.

SEE ALSO

getuid(2), setuid(2), <unistd.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

NAME

setsid, setpgrp, setpgrp3 - create session and set process group ID

SYNOPSIS

```
#include <unistd.h>

pid_t setsid(void);

pid_t setpgrp(void);

pid_t setpgrp3(void);
```

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, it becomes the process group leader of a new process group, and it 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.

The **setpgrp()** function is provided for backward compatibility only.

setpgrp3() function is provided for HPUX compatibity in future releases. **setpgrp3()** is functionally equivalent to setpgrp().

RETURN VALUE

Upon successful completion, **setsid()** returns the value of the new process group ID of the calling process. Otherwise, it returns a value of −**1**, and sets **errno** to indicate the error.

The **setpgrp()** function returns the value of the process group ID of the calling process.

ERRORS

If **setsid()** fails, no changes occur, and **errno** (see *errno*(2)) is set to one of the following values:

[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.

WARNINGS

The semantics for **setpgrp()** may change in a future release (see setpgrp3()).

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).

S

STANDARDS CONFORMANCE

setsid(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

setpgrp(): SVID2, SVID3, XPG2

NAME

setuid, setgid - set user and group IDs

SYNOPSIS

#include <unistd.h>

int setuid(uid_t uid);

int setgid(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 *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 *euid* is zero, **setgid()** sets the *rgid* and *egid* to *gid*.
- If *euid* is not zero, but the argument *gid* is equal to the *rgid* or the *sgid*, **setgid()** sets the *egid* to *gid*; the *rgid* and *sgid* remain unchanged.
- If *euid* is not zero, but the argument *gid* is equal to the *egid*, and the calling process is a member of a group that has the **PRIV_SETRUGID** privilege (see *privgrp*(4)), **setgid()** sets the *rgid* to *gid*; the *egid* and *sgid* remain unchanged.

RETURN VALUE

Upon successful completion, **setuid()** and **setgid()** returned 0; otherwise, they return –1 and set **errno** to indicate the error.

ERRORS

setuid() and **setgid()** fail and return –1 if any of the following conditions are encountered:

[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()** and **setresgid()** (see *setresuid*(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(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

setgid(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

S

NAME

shm_open - create/open a shared memory object

SYNOPSIS

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

int shm_open(const char *name, int oflag, mode_t mode);
```

DESCRIPTION

The **shm_open()** system call establishes a connection between a shared memory object and a file descriptor. It creates an open file description that corresponds to the shared memory object and returns a file descriptor that refers to that open file description. This file descriptor (which is the lowest numbered file descriptor not currently open for that process) is used by other functions to refer to that shared memory object.

The *name* argument points to the shared memory object name, and must conform to the general construction rules for a pathname.

The *oflag* argument is the bitwise inclusive OR of the flags listed under *Read-Write Flags* and *General Flags* (these flags are defined in the header file RC < fcntl.h >).

The new file descriptor has the FD_CLOEXEC flag set, and consequently does not remain open across **exec*()** system calls.

Read-Write Flags

The value of *oflag* must be composed by taking the inclusive OR of exactly one of the following flags:

O_RDONLY	Open for read access only.
O_RDWR	Open for read and write access.

General Flags

Any combination of the following flags may also be used in setting the value of *oflag.*

O_CREAT	If the shared memory object exists, this flag will have no effect, except as noted under O_EXCL below. Otherwise the shared memory object is created; the shared memory object's user ID is set to the effective user ID of the process; the shared memory object's group ID is set to the effective group ID of the process. The shared memory object's permission bits is set to the value of the *mode* argument except those set in the file mode creation mask of the process. The new shared memory object will have a size of zero.
O_EXCL	If O_EXCL and O_CREAT are set in *oflag* and the named shared memory object exists, **shm_open()** will fail. The O_EXCL flag is ignored if O_CREAT is not set in *oflag.*
O_TRUNC	If the shared memory object exists, and it is successfully opened for reading and writing (O_RDWR set in oflag), the object will be truncated to zero length. The mode and owner shall remain unchanged by this function call.

RETURN VALUE

shm_open() returns the following values:

n	Successful completion. *n* is the lowest numbered unused file descriptor for the process.
-1	Failure. **errno** is set to indicate the error.

ERRORS

If **shm_open()** fails, **errno** is set to one of the following values:

[EACCES]	The shared memory object exists and the permissions specified by *oflag* are denied, or the shared memory object does not exist and permission to create the it is denied, or O_TRUNC is specified and write permission is denied.
[EEXIST]	The O_CREAT and O_EXCL are set in *oflag* and the named shared memory object already exists.
[EINTR]	The **shm_open()** operation was interrupted by a signal.

S

[EMFILE] Too many file descriptors are currently in use by this process.

[ENAMETOOLONG]
The length of the *name* string exceeds **PATH_MAX**, or the length of a (pathname) component of the *name* string exceeds **NAME_MAX** while **_POSIX_NO_TRUNC** is in effect.

[ENFILE] Too many shared memory objects are currently open in the system.

[ENOENT] The O_CREAT flag is not set in *oflag* and the named shared memory object does not exist.

[ENOSPC] There are insufficient resources for the creation of the new shared memory object.

[ENOSYS] **shm_open()** is not supported by the implementation.

SEE ALSO

shm_unlink(2), close(2), mmap(2), munmap(2).

STANDARDS CONFORMANCE

shm_open() in librt: POSIX 1003.1b

S

NAME

shm_unlink - unlink a shared memory object

SYNOPSIS

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

int shm_unlink(const char *name);
```

DESCRIPTION

The **shm_unlink()** system call removes the name of the shared memory object named by the string pointed to by *name*. If one or more references to the shared memory object exists when the object is unlinked, the name will be removed before **shm_unlink()** returns, but the removal of the memory object contents will be postponed until all open and map references to the shared memory object have been removed.

RETURN VALUE

shm_unlink() returns the following values:

0 Successful completion.

-1 Failure, **errno** is set to indicate the error.

ERRORS

If **shm_unlink()** fails, **errno** is set to one of the following values:

[EACCES] Permission to unlink the named shared memory object is denied.

[ENAMETOOLONG]
The length of the *name* string exceeds **PATH_MAX**, or the length of a (pathname) component of the *name* string exceeds **NAME_MAX** while **_POSIX_NO_TRUNC** is in effect.

[ENOENT] The named shared memory object does not exist.

[ENOSYS] **shm_unlink()** is not supported by the implementation.

SEE ALSO

shm_open(2), close(2), mmap(2), munmap(2).

STANDARDS CONFORMANCE

shm_unlink() in librt: POSIX 1003.1b

NAME

shmctl() - shared memory control operations

SYNOPSIS

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

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

DESCRIPTION

The **shmctl()** system call provides a variety of shared memory control operations as specified by the *cmd* argument. *cmd* can have the following values:

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 *glossary*(9).

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 a user having appropriate privileges 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 disappears 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 a user with appropriate privileges 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 that of a user having appropriate privileges 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 the **PRIV_MLOCK** privilege (see *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 a user having appropriate privileges 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 the **PRIV_MLOCK** privilege (see *getprivgrp*(2)).

RETURN VALUE

shmctl() returns the following values:

0 Successful completion.
-1 Failure. **errno** is set to indicate the error.

ERRORS

If **shmctl()** fails, **errno** is set to one of the following values.

[EACCES] *cmd* is equal to **IPC_STAT** and Read operation permission is denied to the calling process (see **shared memory operation permissions** in *glossary*(9)).

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

[EINVAL] *cmd* is equal to **SHM_UNLOCK** and the shared-memory segment specified by *shmid* is not locked in memory.

[EINVAL] *shmid* is not a valid shared memory identifier.

[EINVAL] *cmd* is not a valid command, or the command contains invalid parameters.

[ENOMEM] *cmd* is equal to **SHM_LOCK** and there is not sufficient lockable memory to fill the request.

[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 a user having appropriate privileges 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 a user having appropriate privileges and the calling process does not have the **PRIV_MLOCK** privilege (see *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()** (see *shmget*(2).

```
shmctl (myshmid, IPC_RMID, 0);
```

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, SVID3, XPG2, XPG3, XPG4

S

NAME

shmget - get shared memory segment

SYNOPSIS

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

int shmget(key_t key, size_t size, int 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*(9)) 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". If **`IPC_CREAT`** is set in *shmflg* the shared memory segment created can only be shared by processes of the same executable type. That is, an application compiled as a 32-bit process will be able to share the same memory segment with other 32-bit processes, and an application compiled as a 64-bit process will be able to share the same memory segment with other 64-bit processes. If a 64-bit bit process wants to create a shared memory segment which can also be shared with 32-bit processes, the 64-bit process must specify **`IPC_SHARE32`** in addition to **`IPC_CREAT`** in *shmflg*. The 32-bit process does not need to specify **`IPC_SHARE32`**.

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.
- **`shm_perm.cuid`**, 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);
```


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.

ERRORS

`shmget()` fails if any of the following conditions are encountered:

[EINVAL] *size* is less than the system-imposed minimum or greater than the system-imposed maximum.

[EINVAL] A shared memory identifier exists for *key* but is in 64-bit address space and the process performing the request has been compiled as a 32-bit executable. In order to avoid receiving this error, both **`IPC_SHARE32`** and **`IPC_CREAT`** must be set in *shmflg* by the 64-bit procses upon segment creation.

[EACCES] A shared memory identifier exists for *key* but operation permission (see *glossary*(9)) 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".

SEE ALSO

ipcrm(1), ipcs(1), shmctl(2), shmop(2), stdipc(3C).

STANDARDS CONFORMANCE

shmget(): SVID2, SVID3, XPG2, XPG3, XPG4

S

NAME

shmat(), shmdt() - shared memory operations

SYNOPSIS

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

void *shmat(int shmid, void *shmaddr, int shmflg);

int shmdt(void *shmaddr);
```

DESCRIPTION

shmat() attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the data segment of the calling process.

The segment is attached 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.

If the shared memory segment has never been attached to by any process prior to the current **shmat()** call, *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. Once the operating system selects a location for a shared memory segment, the same location will be used across any subsequent **shmat()** and **shmdt()** calls on the segment until it is removed by the **IPC_RMID** operation of **shmctl()**.

If this is not the first **shmat()** call on the shared memory segment throughout the system, *shmaddr* must either be zero or contain a nonzero address that is identical to the one returned from previous **shmat()** calls for that segment. Even if no processes are currently attached to the segment, as long as the segment has been attached before, the same rule applies.

If the calling process is already attached to the shared memory segment, **shmat()** fails and returns **SHM_FAILED** regardless of what value is passed in *shmaddr*.

shmdt() detaches from the calling process's data segment the shared memory segment located at the address specified by *shmaddr*.

RETURN VALUE

shmat() returns the following values:

n Successful completion. *n* is the data segment start address of the attached shared memory segment.

SHM_FAILED
Failure. The shared memory segment is not attached. **errno** is set to indicate the error. The symbol **SHM_FAILED** is defined in the header <sys/shm.h>. No successful return from **shmat()** will return the value **SHM_FAILED**.

shmdt() returns the following values:

0 Successful completion.
-1 Failure. **errno** is set to indicate the error.

ERRORS

If **shmat()** fails, **errno** is set to one of the following values.

[EACCES] Operation permission is denied to the calling process.

[EINVAL] *shmid* is not a valid shared memory identifier, (possibly because the shared memory segment was already removed using *shmctl*(2) with **IPC_RMID**), or the calling process is already attached to *shmid*.

[EINVAL] *shmaddr* is not zero and the machine does not permit nonzero values, or *shmaddr* is not equal to the current attach location for the shared memory segment.

[ENOMEM] The available data space is not large enough to accommodate the shared memory segment.

[EMFILE] The number of shared memory segments attached to the calling process exceed the system-imposed limit.

If **shmdt()** fails, **errno** is set to one of the following values.

S

[EINVAL] *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;
shmptr = (char *) shmat(myshmid, 0, 0);
```

The following call to **shmdt()** then detaches the shared memory segment.

```
shmdt (shmptr);
```

SEE ALSO

ipcs(1), exec(2), exit(2), fork(2), ftok(3C), shmctl(2), shmget(2).

STANDARDS CONFORMANCE

shmat(): SVID2, SVID3, XPG2, XPG3, XPG4

shmdt(): SVID2, SVID3, XPG2, XPG3, XPG4

NAME

shutdown - shut down a socket

SYNOPSIS

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

int shutdown(int s, int how);
```

DESCRIPTION

The **shutdown()** system call is used to shut down a socket. In the case of a full-duplex connection, **shutdown()** can be used to either partially or fully shut down the socket, depending upon the value of *how*.

how	**Interpretation**
SHUT_RD or 0	Further receives are disallowed
SHUT_WR or 1	Further sends are disallowed
SHUT_RDWR or 2	Further sends and receives are disallowed

The *s* parameter is a socket descriptor for the socket to be shut down.

Once the socket has been shut down for receives, all further **recv()** calls return an end-of-file condition. A socket that has been shut down for sending causes further **send()** calls to return an EPIPE error and send the **SIGPIPE** signal. After a socket has been fully shut down, operations other than **recv()** and **send()** return appropriate errors, and the only other thing that can be done to the socket is a **close()**.

Multiple shutdowns on a connected socket and shutdowns on a socket that is not connected may not return errors.

A **shutdown()** on a connectionless socket, such as **SOCK_DGRAM ,** only marks the socket as unable to do further **send()** or **recv()** calls, depending upon the value of *how*. Once this type of socket has been disabled for both sending and receiving data, it becomes fully shut down. For **SOCK_STREAM** sockets, if *how* is **1 or 2**, the connection begins to be closed gracefully in addition to the normal actions. However, the **shutdown()** call does not wait for the completion of the graceful disconnection. The disconnection is complete when both sides of the connection have done a **shutdown()** with *how* equal to **1 or 2**. Once the connection has been completely terminated, the socket becomes fully shut down. The **SO_LINGER** option (see *socket*(2)) does not have any meaning for the **shutdown()** call, but does for the **close()** call. For more information on how the **close()** call interacts with sockets, see *socket*(2).

If a **shutdown()** is performed on a **SOCK_STREAM** socket that has a **listen()** pending on it, that socket becomes fully shut down when *how* = 1.

AF_CCITT only:

The *how* parameter behaves differently if the socket is of the the **AF_CCITT** address family. If *how* is set to **0** the specified socket can no longer receive data. The SVC is not cleared and remains intact. However, if data is subsequently received on the SVC, it is cleared. The connection is not completely down until either side executes a **close()** or **shutdown()** with *how* set to **1 or 2**.

If *how* is set to **1** or **2**, the SVC can no longer send or receive data and the SVC is cleared. The socket's resources are maintained so that data arriving prior to the **shutdown()** call can still be read.

S

RETURN VALUE

Upon successful completion, **shutdown()** returns 0; otherwise it returns −1 and **errno** is set to indicate the error.

ERRORS

shutdown() fails if any of the following conditions are encountered:

[EBADF]	*s* is not a valid file descriptor.
[ENOTSOCK]	*s* is a valid file descriptor, but it is not a socket.
[EINVAL]	**HP-UX BSD Sockets only.** The specified socket is not connected.
[ENOTCONN]	**_XOPEN_SOURCE_EXTENDED only.** The specified socket is not connected.
[EINVAL]	**_XOPEN_SOURCE_EXTENDED only.** The *how* argument is invalid.

FUTURE DIRECTION

Currently, the default behavior is the **HP-UX BSD Sockets**; however, it might be changed to **X/Open Sockets** in a future release. At that time, any **HP-UX BSD Sockets** behavior that is incompatible with **X/Open Sockets** might be obsoleted. Applications that conform to the X/Open specification now will avoid migration problems (see *xopen_networking*(7)).

AUTHOR

shutdown() was developed by HP and the University of California, Berkeley.

SEE ALSO

close(2), connect(2), socket(2), xopen_networking(7).

STANDARDS CONFORMANCE

shutdown(): XPG4

NAME

sigaction - examine and change signal action

SYNOPSIS

```
#include <signal.h>

int sigaction (
     int sig,
     const struct sigaction *act,
     struct sigaction *oact
);
```

DESCRIPTION

The **sigaction()** function allows the calling process to examine and/or specify the action to be associated with a specific signal. The argument *sig* specifies the signal; acceptable values are defined in **<signal.h>**.

The structure sigaction, used to describe an action to be taken, is defined in the header **<signal.h>** to include at least the following members:

Member Type	Member Name	Description
void(*)(int)	**sa_handler**	SIG_DFL, SIG_IGN or pointer to a function.
sigset_t	**sa_mask**	Additional set of signals to be blocked during execution of signal-catching function.
int	**sa_flags**	Special flags to affect behavior of signal.
void(*)(int, siginfo_t*,void *)	**sa_sigaction**	signal-catching function.

If the argument *act* is not a null pointer, it points to a structure specifying the action to be associated with 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 the argument *oact*. If the argument act is a null pointer, signal handling is unchanged; thus, the call can be used to enquire about the current handling of a given signal. The *sa_handler* field of the **sigaction** structure identifies the action to be associated with the specified signal. If the *sa_handler* field specifies a signal-catching function, the *sa_mask* field identifies a set of signals that will be added to the process' signal mask before the signal-catching function is invoked. The **SIGKILL** and **SIGSTOP** signals will not be added to the signal mask using this mechanism; this restriction will be enforced by the system without causing an error to be indicated.

The *sa_flags* field can be used to modify the behavior of the specified signal. The following flags, defined in the header **<signal.h>**, can be set in *sa_flags*:

SA_NOCLDSTOP Do not generate **SIGCHLD** when children stop.

SA_ONSTACK If set and an alternate signal stack has been declared with **sigaltstack()** or **sigstack()**, the signal will be delivered to the calling process on that stack. Otherwise, the signal will be delivered on the current stack.

SA_RESETHAND If set, the disposition of the signal will be reset to **SIG_DFL** and the **SA_SIGINFO** flag will be cleared on entry to the signal handler (Note: **SIGILL**, **SIGTRAP**, and **SIGPWR** cannot be automatically reset when delivered; the system silently enforces this restriction). Otherwise, the disposition of the signal will not be modified on entry to the signal handler. In addition, if this flag is set, **sigaction()** behaves as if the **SA_NODEFER** flag were also set.

SA_RESTART This flag affects the behaviour of interruptible functions; that is, those specified to fail with **errno** set to **EINTR**. If set, and a function specified as interruptible is interrupted by this signal, the function will restart and will not fail with **EINTR** unless otherwise specified. If the flag is not set, interruptible functions interrupted by this signal will fail with **errno** set to **EINTR**.

SA_SIGINFO

If cleared and the signal is caught, the signal-catching function will be entered as:

void func(int signo); where **signo** is the only argument to the signal catching function. In this case the *sa_handler* member must be used to describe the signal catching function and the application must not modify the *sa_sigaction* member.

If **SA_SIGINFO** is set and the signal is caught, the signal-catching function will be entered as:

void func(int signo, siginfo_t *info, void *context); where two additional arguments are passed to the signal catching function. If the second argument is not a null pointer, it will point to an object of type *siginfo_t* explaining the reason why the signal was generated; the third argument can be cast to a pointer to an object of type *ucontext_t* to refer to the context of the receiving process or thread that was interrupted when the signal was delivered. In this case the *sa_sigaction* member must be used to describe the signal catching function and the application must not modify the *sa_handler* member.

The *si_signo* member of *info* contains the system-generated signal number.

The *si_errno* member may contain implementation-dependent additional error information; if non-zero, it contains an error number identifying the condition that caused the signal to be generated.

The *si_code* member contains a code identifying the cause of the signal. If the value of *si_code* is less than or equal to 0, then the signal was generated by a process and *si_pid* and *si_uid* respectively indicate the process ID and the real user ID of the sender. The values of *si_pid* and *si_uid* are otherwise meaningless.

If **SA_SIGINFO** is set in *sa_flags*, subsequent occurrences of *sig* generated by **sigqueue()** or as a result of any signal-generating function that supports the specification of an application-defined value - when *sig* is already pending - will be queued in FIFO order until delivered, and the application specified value will be passed to the signal-catching function as the *si_value* member of *info* (See Realtime Signals Extension). If **SA_SIGINFO** is not set in *sa_flags*, then the disposition of subsequent occurrences of *sig* when it is already pending is implementation-defined.

SA_NOCLDWAIT

If set, and *sig* equals **SIGCHLD**, child processes of the calling process will not be transformed into zombie processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited for children that were transformed into zombie processes, it will block until all of its children terminate, and **wait()**, **wait3()**, **waitid()**, and **waitpid()** will fail and set errno to **ECHILD**. Otherwise, terminating child processes will be transformed into zombie processes, unless **SIGCHLD** is set to **SIG_IGN**.

SA_NODEFER

If set and *sig* is caught, *sig* will not be added to the process' signal mask on entry to the signal handler unless it is included in *sa_mask*. Otherwise, *sig* will always be added to the process' signal mask on entry to the signal handler.

If *sig* is **SIGCHLD** and the **SA_NOCLDSTOP** flag is not set in *sa_flags*, and the implementation supports the **SIGCHLD** signal, then a **SIGCHLD** signal will be generated for the calling process whenever any of its child processes stop. If *sig* is **SIGCHLD** and the **SA_NOCLDSTOP** flag is set in *sa_flags*, then the implementation will not generate a **SIGCHLD** signal in this way.

When a signal is caught by a signal-catching function installed by **sigaction()**, a new signal mask is calculated and installed for the duration of the signal-catching function (or until a call to either **sigprocmask()** or **sigsuspend()** is made). This mask is formed by taking the union of the current signal mask and the value of the *sa_mask* for the signal being delivered unless **SA_NODEFER** or **SA_RESETHAND** is set, and then including the signal being delivered. If and when the user's signal handler returns normally, the original signal mask is restored.

Once an action is installed for a specific signal, it remains installed until another action is explicitly requested (by another call to **sigaction()**), until the **SA_RESETHAND** flag causes resetting of the handler, or until one of the exec functions is called.

If the previous action for *sig* had been established by **signal()**, the values of the fields returned in the structure pointed to by *oact* are unspecified, and in particular *oact->sa_handler* is not necessarily the same value passed to **signal()**. However, if a pointer to the same structure or a copy thereof is passed to a subsequent call to **sigaction()** via the *act* argument, handling of the signal will be as if the original call to **signal()** were repeated.

If **sigaction()** fails, no new signal handler is installed.

It is unspecified whether an attempt to set the action for a signal that cannot be caught or ignored to **SIG_DFL** is ignored or causes an error to be returned with **errno** set to **EINVAL**.

A signal is said to be **generated** for (or sent to) a process when the event that causes the signal first occurs. Examples of such events include detection of hardware faults, timer expiration and terminal activity, as well as the invocation of **kill()** and **sigqueue()**. In some circumstances, the same event generates signals for multiple processes.

Each process has an action to be taken in response to each signal defined by the system (see Signal Actions). A signal is said to be delivered to a process when the appropriate action for the process and signal is taken.

During the time between the generation of a signal and its delivery, the signal is said to be **pending**. Ordinarily, this interval cannot be detected by an application. However, a signal can be **blocked** from delivery to a process. If the action associated with a blocked signal is anything other than to ignore the signal, and if that signal is generated for the process, the signal will remain pending until either it is unblocked or the action associated with it is set to ignore the signal. If the action associated with a blocked signal is to ignore the signal and if that signal is generated for the process, it is unspecified whether the signal is discarded immediately upon generation or remains pending.

Each process has a signal mask that defines the set of signals currently blocked from delivery to it. The signal mask for a process is initialized from that of its parent. The **sigaction()**, **sigprocmask()**, and **sigsuspend()** functions control the manipulation of the signal mask.

The determination of which action is taken in response to a signal is made at the time the signal is delivered, allowing for any changes since the time of generation. This determination is independent of the means by which the signal was originally generated. If a subsequent occurrence of a pending signal is generated, it is implementation-dependent as to whether the signal is delivered more than once. The order in which multiple, simultaneously pending signals are delivered to a process is unspecified.

When any stop signal (**SIGSTOP**, **SIGTSTP**, **SIGTTIN**, **SIGTTOU**) is generated for a process, any pending **SIGCONT** signals for that process will be discarded. Conversely, when **SIGCONT** is generated for a process, all pending stop signals for that process will be discarded. When **SIGCONT** is generated for a process that is stopped, the process will be continued, even if the **SIGCONT** signal is blocked or ignored. If **SIGCONT** is blocked and not ignored, it will remain pending until it is either unblocked or a stop signal is generated for the process.

Some signal-generating functions, such as high-resolution timer expiration, asynchronous I/O completion, interprocess message arrival, and the **sigqueue()** function, support the specification of an application-defined value, either explicitly as a parameter to the function or in a **sigevent** structure parameter (see **signal(5)**).

Realtime Signals Extension

When a signal is generated by **sigqueue()** or any signal-generating function that supports the specification of an application-defined value, and if the **SA_SIGINFO** flag is set for that signal, the signal will be queued to the process along with the application-specified signal value. Multiple occurrences of signals so generated are queued in FIFO order. When multiple unblocked signals, all in the range **SIGRTMIN** to **SIGRTMAX**, are pending, the implementation delivers the pending unblocked signal with the lowest signal number within that range. The selection order between realtime and nonrealtime signals, or between multiple pending nonrealtime signals, is unspecified. Signals generated by **kill()** or other events that cause signals to occur, such as detection of hardware faults, **alarm()** timer expiration, or terminal activity, and for which the implementation does not support queueing, will have no effect on signals already queued for the same signal number.

If, when a pending signal is delivered, there are additional signals to be queued to that signal number, the signal will remain pending. Otherwise, the pending indication will be reset.

An implementation will document any condition not specified by this document under which the implementation generates signals.

Signal Actions

There are three types of action that can be associated with a signal: **SIG_DFL**, **SIG_IGN** or a pointer to a function. Initially, all signals will be set to **SIG_DFL** or **SIG_IGN** prior to entry of the **main()** routine (see the **exec** functions). The actions prescribed by these values are as follows:

SIG_DFL - signal-specific default action

- The default actions for the signals defined in this document are specified under **<signal.h>**.
- If the default action is to stop the process, the execution of that process is temporarily suspended. When a process stops, a **SIGCHLD** signal will be generated for its parent process, unless the parent process has set the **SA_NOCLDSTOP** flag. While a process is stopped, any additional signals that are sent to the process will not be delivered until the process is continued, except **SIGKILL** which always terminates the receiving process. A process that is a member of an orphaned process group will not be allowed to stop in response to the **SIGTSTP**, **SIGTTIN**, or **SIGTTOU** signals. In cases where delivery of one of these signals would stop such a process, the signal will be discarded.
- Setting a signal action to **SIG_DFL** for a signal that is pending, and whose default action is to ignore the signal (for example, **SIGCHLD**), will cause the pending signal to be discarded, whether or not it is blocked. Any queued values pending will be discarded, and the system resources used to queue them will be released and made available to queue other signals.

SIG_IGN - ignore signal

- Delivery of the signal will have no effect on the process. The behaviour of a process is undefined after it ignores a **SIGFPE**, **SIGILL**, or **SIGSEGV** signal that was not generated by **kill()**, **sigqueue()** or **raise()**.
- The system will not allow the action for the signals **SIGKILL** or **SIGSTOP** to be set to **SIG_IGN**.
- Setting a signal action to **SIG_IGN** for a signal that is pending will cause the pending signal to be discarded, whether or not it is blocked. Any queued values pending will be discarded, and the system resources used to queue them will be released and made available to queue other signals.
- If a process sets the action for the **SIGCHLD** signal to **SIG_IGN**, the behaviour is unspecified, except as specified below.

  If the action for the **SIGCHLD** signal is set to **SIG_IGN**, child processes of the calling processes will not be transformed into zombie processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited for children that were transformed into zombie processes, it will block until all of its children terminate, and **wait()**, **wait3()**, **waitid()**, and **waitpid()** will fail and set **errno** to **ECHILD**.

S

Pointer to a function - catch signal

- On delivery of the signal, the receiving process is to execute the signal-catching function at the specified address. After returning from the signal-catching function, the receiving process will resume execution at the point at which it was interrupted.
- If **SA_SIGINFO** is cleared, the signal-catching function will be entered as:

  ```
  void func(int signo);
  ```

  where **func** is the specified signal-catching function and *signo* is the signal number of the signal being delivered.
- If **SA_SIGINFO** is set, the signal-catching function will be entered as:

  ```
  void func(int signo, siginfo_t *siginfo, void *ucontextptr);
  ```

  where **func** is the specified signal-catching function, **signo** is the signal number of the signal being delivered, *siginfo* points to an object of type *siginfo_t* associated with the signal being delivered, and **ucontextptr** points to a *ucontext_t*.
- The behaviour of a process is undefined after it returns normally from a signal- catching function for a **SIGBUS**, **SIGFPE**, **SIGILL**, or **SIGSEGV** signal that was not generated by **kill()** or **raise()**.

- The system will not allow a process to catch the signals **SIGKILL** and **SIGSTOP**.
- If a process establishes a signal-catching function for the **SIGCHLD** signal while it has a terminated child process for which it has not waited, it is unspecified whether a **SIGCHILD** signal is generated to indicate that child process.
- When signal-catching functions are invoked asynchronously with process execution, the behaviour of some of the functions defined by this document is unspecified if they are called from a signal-catching function.

  The following table defines a set of functions that are either reentrant or not interruptible by signals. Therefore applications may invoke them, without restriction, from signal-catching functions:

access()	**fstat()**	**read()**	**sysconf()**
alarm()	**getegid()**	**rename()**	**tcdrain()**
cfgetispeed()	**geteuid()**	**rmdir()**	**tcflow()**
cfgetospeed()	**getgid()**	**setgid()**	**tcflush()**
cfsetispeed()	**getgroups()**	**setpgid()**	**tcgetattr()**
cfsetospeed()	**getpgrp()**	**setsid()**	**tcgetpgrp()**
chdir()	**getpid()**	**setuid()**	**tcsendbreak()**
chmod()	**getppid()**	**sigaction()**	**tcsetattr()**
chown()	**getuid()**	**sigaddset()**	**tcsetpgrp()**
close()	**kill()**	**sigdelset()**	**time()**
creat()	**link()**	**sigemptyset()**	**times()**
dup2()	**lseek()**	**sigfillset()**	**umask()**
dup()	**mkdir()**	**sigismember()**	**uname()**
execle()	**mkfifo()**	**signal()**	**unlink()**
execve()	**open()**	**sigpending()**	**utime()**
_exit()	**pathconf()**	**sigprocmask()**	**wait()**
		sigqueue()	
fcntl()	**pause()**	**sigsuspend()**	**waitpid()**
fork()	**pipe()**	**sleep()**	**write()**
fpathconf()	**raise()**	**stat()**	

  All functions not in the above table are considered to be unsafe with respect to signals. In the presence of signals, all functions defined by this document will behave as defined when called from or interrupted by a signal-catching function, with a single exception: when a signal interrupts an unsafe function and the signal-catching function calls an unsafe function, the behaviour is undefined.

Signal Effects on Other Functions

Signals affect the behaviour of certain functions defined by this document if delivered to a process while it is executing such a function. If the action of the signal is to terminate the process, the process will be terminated and the function will not return. If the action of the signal is to stop the process, the process will stop until continued or terminated. Generation of a **SIGCONT** signal for the process causes the process to be continued, and the original function will continue at the point the process was stopped. If the action of the signal is to invoke a signal-catching function, the signal-catching function will be invoked; in this case the original function is said to be **interrupted** by the signal. If the signal-catching function executes a return statement, the behaviour of the interrupted function will be as described individually for that function. Signals that are ignored will not affect the behaviour of any function; signals that are blocked will not affect the behaviour of any function until they are unblocked and then delivered.

S

RETURN VALUE

Upon successful completion, **sigaction()** returns 0. Otherwise −1 is returned, **errno** is set to indicate the error and no new signal-catching function will be installed.

ERRORS

The **sigaction()** function will fail if:

[EINVAL] The *sig* argument is not a valid signal number or an attempt is made to catch a signal that cannot be caught or ignore a signal that cannot be ignored.

The **sigaction()** function may fail if:

[EINVAL] An attempt was made to set the action to **SIG_DFL** for a signal that cannot be caught or ignored (or both).

APPLICATION USAGE

The **sigaction()** function supersedes the **signal()** interface, and should be used in preference. In particular, **sigaction()** and **signal()** should not be used in the same process to control the same signal. The behaviour of reentrant functions, as defined in the description, is as specified by this document, regardless of invocation from a signal-catching function. This is the only intended meaning of the statement that reentrant functions may be used in signal-catching functions without restrictions. Applications must still consider all effects of such functions on such things as data structures, files and process state. In particular, application writers need to consider the restrictions on interactions when interrupting **sleep()** and interactions among multiple handles for a file descriptor. The fact that any specific function is listed as reentrant does not necessarily mean that invocation of that function from a signal-catching function is recommended.

In order to prevent errors arising from interrupting non-reentrant function calls, applications should protect calls to these functions either by blocking the appropriate signals or through the use of some programmatic semaphore. This document does not address the more general problem of synchronizing access to shared data structures. Note in particular that even the "safe" functions may modify the global variable **errno**; the signal-catching function may want to save and restore its value. Naturally, the same principles apply to the reentrancy of application routines and asynchronous data access. Note that **longjmp()** and **siglongjmp()** are not in the list of reentrant functions. This is because the code executing after **longjmp()** and **siglongjmp()** can call any unsafe functions with the same danger as calling those unsafe functions directly from the signal handler. Applications that use **longjmp()** and **siglongjmp()** from within signal handlers require rigorous protection in order to be portable. Many of the other functions that are excluded from the list are traditionally implemented using either **malloc()** or **free()** functions or the standard I/O library, both of which traditionally use data structures in a non-reentrant manner. Because any combination of different functions using a common data structure can cause reentrancy problems, this document does not define the behaviour when any unsafe function is called in a signal handler that interrupts an unsafe function.

If the signal occurs other than as the result of calling **abort()**, **kill()**, **sigqueue()**, or **raise()**, the behaviour is undefined if the signal handler calls any function in the standard library other than one of the functions listed in the table above or refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type volatile *sig_atomic_t*. Furthermore, if such a call fails, the value of **errno** is indeterminate.

Usually, the signal is executed on the stack that was in effect before the signal was delivered. An alternate stack may be specified to receive a subset of the signals being caught.

When the signal handler returns, the receiving process will resume execution at the point it was interrupted unless the signal handler makes other arrangements. If **longjmp()** or **_longjmp()** is used to leave the signal handler, then the signal mask must be explicitly restored by the process.

S

POSIX.4-1993 defines the third argument of a signal handling function when **SA_SIGINFO** is set as a *void* * instead of a *ucontext_t* *, but without requiring type checking. New applications should explicitly cast the third argument of the signal handling function to *uncontext_*t *.

The BSD optional four argument signal handling function is not supported by this specification. The BSD declaration would be

```
void handler(int sig, int code, struct sigcontext *scp, char *addr);
```

where *sig* is the signal number, code is additional information on certain signals, *scp* is a pointer to the **sigcontext** structure, and *addr* is additional address information. Much the same information is available in the objects pointed to by the second argument of the signal handler specified when **SA_SIGINFO** is set.

Threads Considerations

The signal disposition, catch/ignore/default, established by **sigaction()** is shared by all threads in the process.

If the signal disposition for *sig* is set to **SIG_IGN** or is set to **SIG_DFL** and the default action for *sig* is to ignore the signal, any instances of *sig* pending on the process or any of the threads will be discarded. The signals are discarded regardless of whether the signal is blocked by any of the threads.

For more information regarding signals and threads, see *signal*(5).

FUTURE DIRECTIONS

The **fpathconf()** function is marked as an extension in the list of safe functions because it is not included in the corresponding list in the ISO POSIX-1 standard, but it is expected to be added in a future revision of that standard.

SEE ALSO

kill(2), setjmp(3C), sigaltstack(2), signal(2), sigprocmask(2), sigqueue(2), sigsetops(3C), sigsuspend(2), wait(2), waitid(2), <signal.h>, <ucontext.h>.

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 4

The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- The type of argument act is changed from **struct sigaction *** to **const struct sigaction ***.
- A statement is added to the DESCRIPTION section indicating that the consequence of attempting to set **SIG_DFL** for a signal that cannot be caught or ignored is unspecified. The **EINVAL** error, describing one possible reaction to this condition, is added to the ERRORS section.

Other changes are incorporated as follows:

- The **raise()** and **signal()** functions are added to the list of functions that are either reentrant or not interruptible by signals; **fpathconf()** is also added to this list and marked as an extension; **ustat()** is removed from the list, as this function is withdrawn from the interface definition. It is no longer specified whether **abort()**, **chroot()**, **exit()**, and **longjmp()** also fall into this category of functions.
- The APPLICATION USAGE section is added. Most of this text is moved from the DESCRIPTION SECTION in Issue 3.
- The FUTURE DIRECTIONS section is added.

Issue 4, Version 2

The following changes are incorporated for X/OPEN UNIX conformance:

- The DESCRIPTION describes **sa_sigaction**, the member of the sigaction structure that is the signal-catching function.
- The DESCRIPTION describes the **SA_ONSTACK**, **SA_RESETHAND**, **SA_RESTART**, **SA_SIGINFO**, **SA_NOCLDWAIT**, and **SA_NODEFER** settings of *sa_flags*. The text describes the implications of the use of **SA_SIGINFO** for the number of arguments passed to the signal-catching function. The text also describes the effects of the **SA_NODEFER** and **SA_RESETHAND** flags on the delivery of a signal and on the permanence of an installed action.
- The DESCRIPTION specifies the effect if the action for the **SIGCHLD** signal is set to **SIG_IGN**.
- In the DESCRIPTION, additional text describes the effect if the action is a pointer to a function. A new bullet covers the case where **SA_SIGINFO** is set. **SIGBUS** is given as an additional signal for which the behaviour of a process is undefined following a normal return from the signal-catching function.
- The APPLICATION USAGE section is updated to describe use of an alternate signal stack; resumption of the process receiving the signal; coding for compatibility with POSIX.4-1993; and implementation of signal-handling functions in BSD.

HP-UX EXTENSIONS

DESCRIPTION

More details on the semantics of specific signals can be found in the *signal*(5) manual entry.

SIG_DFL Upon receipt of the signal *sig*, the default action (specified on *signal*(5)) is performed.

ERRORS

[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 POSIX 1003.1-1988 Standard.

SEE ALSO

ptrace(2), sigpending(2), sigspace(2), sigsetops(3C).

STANDARDS CONFORMANCE

sigaction() : AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

sigaltstack - set and/or get signal alternate stack context.

SYNOPSIS

```
#include <signal.h>

int sigaltstack(const stack_t *ss, stack_t *oss);
```

DESCRIPTION

The **sigaltstack()** function allows a process to define and examine the state of an alternate stack for signal handlers. Signals that have been explicitly declared to execute on the alternate stack will be delivered on the alternate stack.

If *ss* is not a null pointer, it points to a **stack_t** structure that specifies the alternate signal stack that will take effect upon return from **sigaltstack()**. The *ss_flags* member specifies the new stack state. If it is set to **SS_DISABLE**, the stack is disabled and *ss_sp* and *ss_size* are ignored. Otherwise the stack will be enabled, and the *ss_sp* and *ss_size* members specify the new address and size of the stack.

The range of addresses starting at *ss_sp*, up to but not including *ss_sp*+*ss_size*, is available to the implementation for use as the stack. This interface makes no assumptions regarding which end is the stack base and in which direction the stack grows as items are pushed.

If *oss* is not a null pointer, on successful completion it will point to a **stack_t** structure that specifies the alternate signal stack that was in effect prior to the call to **sigaltstack()**. The *ss_sp* and *ss_size* members specify the address and size of that stack. The *ss_flags* member specifies the stack's state, and may contain one of the following values:

SS_ONSTACK	The process is currently executing on the alternate signal stack. Attempts to modify the alternate signal stack while the process is executing on it fails. This flag must not be modified by processes.
SS_DISABLE	The alternate signal stack is currently disabled.

The value **SIGSTKSZ** is a system default specifying the number of bytes that would be used to cover the usual case when manually allocating an alternate stack area. The value **MINSIGSTKSZ** is defined to be the minimum stack size for a signal handler. In computing an alternate stack size, a program should add that amount to its stack requirements to allow for the system implementation overhead. The constants **SS_ONSTACK**,**SS_DISABLE**,**SIGSTKSZ**, and **MINSIGSTKSZ** are defined in **<signal.h>**.

After a successful call to one of the **exec** functions, there are no alternate signal stacks in the new process image.

RETURN VALUE

Upon successful completion, **sigaltstack()** returns 0. Otherwise, it returns −1 and sets **errno** to indicate the error.

S

ERRORS

The **sigaltstack()** function will fail if:

[EINVAL]	The *ss* argument is not a null pointer, and the *ss_flags* member pointed to by *ss* contains flags other than **SS_DISABLE**.
[ENOMEM]	The size of the alternate stack area is less than **MINSIGSTKSZ**.
[EPERM]	An attempt was made to modify an active stack.

APPLICATION USAGE

The following code fragment illustrates a method for allocating memory for an alternate stack:

```
if ((sigstk.ss_sp = malloc(SIGSTKSZ)) == NULL)
     /* error return */
sigstk.ss_size = SIGSTKSZ;
sigstk.ss_flags = 0;
if (sigaltstack(&sigstk,(stack_t *)0) < 0)
     perror("sigaltstack");
```

In some implementations, a signal (whether or not indicated to execute on the alternate stack) will always execute on the alternate stack if it is delivered while another signal is being caught using the alternate stack.

On some implementations, stack space is automatically extended as needed. On those implementations, automatic extension is typically not available for an alternate stack. If the stack overflows, the behaviour is undefined.

Threads Considerations

Each thread may define an alternate signal handling stack.

LWP (Light Weight Processes) Considerations

Each LWP may define an alternate signal handling stack.

SEE ALSO

sigaction(2), sigsetjmp(2), <signal.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

NAME

sigblock - block signals

SYNOPSIS

```
#include <signal.h>

long sigblock(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.

Use **sigsetmask()** to set the mask absolutely (see *sigsetmask*(2)).

RETURN VALUE

sigblock() returns the previous set of masked signals.

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

Do not use **sigblock()** in conjunction with the facilities described under *sigset*(3C).

APPLICATION USAGE

Threads Considerations

Since each thread maintains its own blocked signal mask, **sigblock()** modifies only the calling thread's blocked signal mask.

For more information regarding signals and threads, refer to *signal*(5).

LP64 Programs

sigblock() accepts and returns long (64 bit) values. However, as for ILP32 programs, **sigblock()** supports signals numbered 1 through 32. The upper 32 bits of the mask argument are ignored. Also, the upper 32 bits of the returned mask have no meaning.

AUTHOR

sigblock() was developed by the University of California, Berkeley.

SEE ALSO

kill(2), sigprocmask(2), sigsetmask(2), sigvector(2), signal(5).

NAME

sigset, sighold, sigrelse, sigignore, sigpause - signal management

SYNOPSIS

```
#include <signal.h>

void (*sigset(int sig, void (*func)(int)))(int);

int sighold(int sig);

int sigrelse(int sig);

int sigignore(int sig);

int sigpause(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*(3C) 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 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() holds the signal *sig*. **sigrelse()** 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.

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**>.

S

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

sigset() fails and the system signal action for *sig* is not changed if any of the following occur:

[EFAULT] The *func* argument points to memory that is not a valid part of the process address space. Reliable detection of this error is implementation dependent.

sigset(), **sighold()**, **sigrelse()**, **sigignore()**, and **sigpause()** fail and the system signal action for *sig* is not changed if any of the following occur:

[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).

sigpause returns when the following occurs:

[EINTR] A signal was caught.

WARNINGS

These signal facilities should not be used in conjunction with *bsdproc*(3C), *signal*(2), *sigvector*(2), *sigblock*(2), *sigsetmask*(2), *sigpause*(3C) and *sigspace*(2).

SEE ALSO

kill(1), kill(2), signal(2), pause(2), wait(2), abort(3C), setjmp(3C), signal(5).

STANDARDS CONFORMANCE

sigset(): SVID2, SVID3

sighold(): SVID2, SVID3

sigignore(): SVID2, SVID3

sigpause(): SVID2, SVID3

sigrelse(): SVID2, SVID3

S

NAME

siginterrupt - allow signals to interrupt functions

SYNOPSIS

```
#include <signal.h>

int siginterrupt(int sig, int flag);
```

DESCRIPTION

The **siginterrupt()** function is used to change the restart behaviour when a function is interrupted by the specified signal. The function **siginterrupt** (*sig*, *flag*) has an effect as if implemented as:

```
siginterrupt(int sig, int flag) {
     int ret;
     struct sigaction act;

     (void) sigaction(sig, NULL, &act);
     if (flag)
          act.sa_flags &= ~SA_RESTART;
     else
          act.sa_flags |= SA_RESTART;
     ret = sigaction(sig, &act, NULL);
     return ret;
}
```

RETURN VALUE

Upon successful completion, **siginterrupt()** returns 0. Otherwise −1 is returned and **errno** is set to indicate the error.

ERRORS

The **siginterrupt()** function will fail if:

[EINVAL] The *sig* argument is not a valid signal number.

APPLICATION USAGE

The **siginterrupt()** function supports programs written to historical system interfaces. A portable application, when being written or rewritten, should use **sigaction()** with the **SA_RESTART** flag instead of **siginterrupt()**.

Threads Considerations

System call restart is a process attribute. Therefore, changing the restart behavior affects all threads in the process.

SEE ALSO

sigaction(2), <signal.h>.

S

CHANGE HISTORY

First released in Issue 4, Version 2.

NAME

signal, sigset, sighold, sigrelse, sigignore, sigpause - signal management

SYNOPSIS

```
#include <signal.h>

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

int sighold(int sig);

int sigignore(int sig);

int sigpause(int sig);

int sigrelse(int sig);

void (*sigset(int sig, void (*disp)(int)))(int);
```

DESCRIPTION

The **signal()** function chooses one of three ways in which receipt of the signal number *sig* is to be subsequently handled. If the value of func is **SIG_DFL**, default handling for that signal will occur. If the value of *func* is **SIG_IGN**, the signal will be ignored. Otherwise, *func* must point to a function to be called when that signal occurs. Such a function is called a signal handler.

When a signal occurs, if *func* points to a function, first the equivalent of a:

```
signal(sig, SIG_DFL);
```

is executed or an implementation-dependent blocking of the signal is performed. (If the value of *sig* is **SIGILL**, whether the reset to **SIG_DFL** occurs is implementation-dependent.) Next the equivalent of:

```
(*func)(sig);
```

is executed. The *func* function may terminate by executing a return statement or by calling **abort()**, **exit()**, or **longjmp()**. If **func()** executes a return statement and the value of *sig* was **SIGFPE** or any other implementation-dependent value corresponding to a computational exception, the behaviour is undefined. Otherwise, the program will resume execution at the point it was interrupted.

If the signal occurs other than as the result of calling **abort()**, **kill()** or **raise()**, the behaviour is undefined if the signal handler calls any function in the standard library other than one of the functions listed on the *sigaction*(2) page or refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type volatile *sig_atomic_t*. Furthermore, if such a call fails, the value of **errno** is indeterminate.

At program startup, the equivalent of:

```
signal(sig, SIG_IGN);
```

is executed for some signals, and the equivalent of:

```
signal(sig, SIG_DFL);
```

is executed for all other signals (see **exec**).

The **sigset()**, **sighold()**, **sigignore()**, **sigpause()** and **segrelse()** functions provide simplified signal management.

The **sigset()** function is used to modify signal dispositions. The *sig* argument specifies the signal, which may be any signal except **SIGKILL** and **SIGSTOP**. The *disp* argument specifies the signal's disposition, which may be **SIG_DFL**, **SIG_IGN** or the address of a signal handler. If **sigset()** is used, and *disp* is the address of a signal handler, the system will add *sig* to the calling process' signal mask before executing the signal handler; when the signal handler returns, the system will restore the calling process' signal mask to its state prior the delivery of the signal. In addition, if **sigset()** is used, and *disp* is equal to **SIG_HOLD**, *sig* will be added to the calling process' signal mask and *sig*'s disposition will remain unchanged. If **sigset()** is used, and *disp* is not equal to **SIG_HOLD**, *sig* will be removed from the calling process' signal mask.

The **sighold()** function adds *sig* to the calling process' signal mask.

The **sigrelse()** function removes *sig* from the calling process' signal mask.

The **sigignore()** function sets the disposition of *sig* to **SIG_IGN**.

The sigpause() function removes *sig* from the calling process' signal mask and suspends the calling process until a signal is received.

If the action for the **SIGCHLD** signal is set to **SIG_IGN**, child processes of the calling processes will not be transformed into zombie processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited for children that were transformed into zombie processes, it will block until all of its children terminate, and **wait()**, **wait3()**, **waitid()** and **waitpid()** will fail and set **errno** to **ECHILD**.

RETURN VALUE

If the request can be honoured, **signal()** returns the value of **func()** for the most recent call to **signal()** for the specified signal *sig*. Otherwise, **SIG_ERR** is returned and a positive value is stored in **errno**.

Upon successful completion, **sigset()** returns **SIG_HOLD** if the signal had been blocked and the signal's previous disposition if it had not been blocked. Otherwise, **SIG_ERR** is returned and **errno** is set to indicate the error.

For all other functions, upon successful completion, 0 is returned. Otherwise, −1 is returned and **errno** is set to indicate the error.

ERRORS

The **signal()** function will fail if:

[EINVAL] The *sig* argument is not a valid signal number or an attempt is made to catch a signal that cannot be caught or ignore a signal that cannot be ignored.

The **signal()** function may fail if:

[EINVAL] An attempt was made to set the action to **SIG_DFL** for a signal that cannot be caught or ignored (or both).

The **sigset()**, **sighold()**, **sigrelse()**, **sigignore()**, and **sigpause()** functions will fail if:

[EINVAL] The *sig* argument is an illegal signal number.

The **sigset()**, and **sigignore()** functions will fail if:

[EINVAL] An attempt is made to catch a signal that cannot be caught, or to ignore a signal that cannot be ignored.

APPLICATION USAGE

The **sigaction()** function provides a more comprehensive and reliable mechanism for controlling signals; new applications should use **sigaction()** rather than **signal()**.

The **sighold()** function, in conjunction with **sigrelse()** or **sigpause()**, may be used to establish critical regions of code that require the delivery of a signal to be temporarily deferred.

The **sigsuspend()** function should be used in preference to **sigpause()** for broader portability.

Threads Considerations

The signal disposition (such as catch/ignore/default) established by **signal()** is shared by all threads in the process. Blocked signal masks are maintained by each thread.

If **signal()** is used to set the signal disposition for *sig* to **SIG_IGN** or to **SIG_DFL** for a signal whose default action is to ignore the signal, any instances of *sig* pending on the process or any of the threads will be discarded. The signals are discarded regardless of whether the signal is blocked by any of the threads.

For more information regarding signals and threads, refer to *signal*(5).

SEE ALSO

exec(2), pause(2), sigaction(2), waitid(2), <signal.h>.

CHANGE HISTORY

First released in Issue 1.

Derived from Issue 1 of the SVID.

Issue 4

The following changes are incorporated for alignment with the ISO C standard:

- The function is no longer marked as an extension.
- The argument int is added to the definition of **func()** in the SYNOPSIS section.
- In Issue 3, this interface cross-referred to **sigaction()**. This issue provides a complete description of the function as defined in ISO C standard.

Another change is incorporated as follows:

- The APPLICATION USAGE section is added.

Issue 4, Version 2

The following changes are incorporated for X/OPEN UNIX conformance:

- The **sighold()**, **sigignore()**, **sigpause()**, **sigrelse()**, and **sigset()** functions are added to the SYNOPSIS.
- The DESCRIPTION is updated to describe semantics of the above interfaces.
- Additional text is added to the RETURN VALUE section to describe possible returns from the **sigset()** function specifically, and all of the above functions in general.
- The ERRORS section is restructured to describe possible error returns from each of the above functions individually.
- The APPLICATION USAGE section is updated to describe certain programming considerations associated with the X/OPEN UNIX functions.

HP-UX EXTENSIONS

SYNOPSIS

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

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

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*(3C), and *sigspace*(2).

Acceptable values for *sig* are defined in **<signal.h>**.

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.

SIGKILL and **SIGSTOP** signals cannot be ignored.

address Catch the signal.

Upon receipt of 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 where 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**>.

The pointer *scp* is valid only during the context of the signal-catching function. 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.

As defined by the IEEE POSIX Standard, HP-UX does not raise an exception on floating-point divide by zero. The result of floating-point divide by zero is infinity which can be checked by *isinf*(3M).

The signals **SIGKILL** and **SIGSTOP** cannot be caught.

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.

S

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 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() holds the signal *sig*. **sigrelse()** 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 the **ld** command (see *ld*(1)).

ERRORS

sigset() fails and the system signal action for *sig* is not changed if any of the following occur:

[EFAULT] The *func* argument points to memory that is not a valid part of the process address space. Reliable detection of this error is implementation-dependent.

sigset(), **sighold()**, **sigrelse()**, **sigignore()**, and **sigpause()** fail and the system signal action for *sig* is not changed if any of the following occur:

[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).

sigpause returns when the following occurs:

[EINTR] A signal was caught.

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*(3C), *sigaction*(2), *sigset*(3C), or *sigvector*(2).

S

signal() does not detect an invalid value for *action*, 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.

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(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, ANSI C

NAME

sigpending - examine pending signals

SYNOPSIS

```
#include <signal.h>

int sigpending(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 0. Otherwise −1 is returned and **errno** is set to indicate the error.

ERRORS

No errors are defined.

APPLICATION USAGE

Threads Considerations

The set of signals returned by **sigpending()** is the union of the signals pending on the process and calling thread. A signal may be pending on the process if all threads block the signal.

The set of signals returned by **sigpending()** is only advisory. Since other threads may be executing at the time of the call, a signal pending on the process may be delivered to a thread after this system call returns.

For more information regarding signals and threads, refer to *signal*(5).

LWP (Lightweight Processes) Considerations

The set of signals returned by **sigpending()** is the union of the signals pending on the process and calling LWP.

SEE ALSO

sigprocmask(2), sigsetops(3C), <signal.h>.

CHANGE HISTORY

First release in Issue 3.

S

HP-UX EXTENSIONS

ERRORS

sigpending() fails if the following condition is encountered:

[EFAULT] *set* points to an invalid address. The reliable detection of this error is implementation-dependent.

AUTHOR

sigpending() was derived from the IEEE POSIX 1003.1-1988 Standard.

SEE ALSO

sigaction(2), sigsuspend(2), sigprocmask(2), sigsetops(3C), signal(5).

STANDARDS CONFORMANCE

sigpending() : AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

sigprocmask - examine and change blocked signals

SYNOPSIS

```
#include <signal.h>

int sigprocmask(
     int how,
     const sigset_t *set,
     sigset_t *oset
);
```

DESCRIPTION

The **sigprocmask()** function allows the calling process to examine and/or change its signal mask.

If the argument set is not a null pointer, it points to a set of signals to be used to change the currently blocked set.

The argument *how* indicates the way in which the set is changed and consists of one of the following values:

SIG_BLOCK The resulting set will be the union of the current set and the signal set pointed to by set.

SIG_SETMASK The resulting set will be the signal set pointed to by set.

SIG_UNBLOCK The resulting set will be the intersection of the current set andthe complement of the signal set pointed to by set.

If the argument *oset* is not a null pointer, the previous mask is stored in the location pointed to by *oset*. If *set* is a null pointer, the value of the argument *how* is not significant and the process' signal mask is unchanged; thus the call can be used to inquire about currently blocked signals.

If there are any pending unblocked signals after the call to **sigprocmask()**, at least one of those signals will be delivered before the call to **sigprocmask()** returns.

It is not possible to block those signals which cannot be ignored. This is enforced by the system without causing an error to be indicated.

If any of the **SIGFPE**, **SIGILL**, or **SIGSEGV** signals are generated while they are blocked, the result is undefined, unless the signal was generated by a call to **kill()** or **raise()**.

If **sigprocmask()** fails, the process' signal mask is not changed.

RETURN VALUE

Upon successful completion, **sigprocmask()** returns 0. Otherwise –1 is returned, **errno** is set to indicate the error, and the process' signal mask will be unchanged.

ERRORS

The **sigprocmask()** function will fail if:

[EINVAL] The value of the how argument is not equal to one of the defined values.

APPLICATION USAGE

Threads Considerations

Since each thread maintains its own blocked signal mask, **sigprocmask()** modifies only the calling thread's blocked signal mask.

For more information regarding signals and threads, refer to *signal*(5).

LWP (Lightweight Processes) Considerations

sigprocmask() modifies only the calling LWP's blocked signal mask.

SEE ALSO

sigaction(2), sigpending(2), sigsetops(3C), sigsuspend(2), <signal.h>.

CHANGE HISTORY

First released in Issue 3.

Entry included foralignment with the POSIX.1-1988 standard.

Issue 4

The following change is incorporated for alignment with the ISO POSIX-1 standard:

- The type of the arguments *set* and *oset* are changed from *sigset_t** to *const sigset_t**.

Another change is incorporated as follows:

- The DESCRIPTION section is changed to indicate that signals can also be generated by **raise()**.

S

HP-UX EXTENSIONS

ERRORS

`sigprocmask()` fails if any of the following conditions are encountered:

[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 POSIX 1003.1-1988 Standard.

SEE ALSO

sigaction(2), sigsuspend(2), sigpending(2), sigsetops(3C), signal(5).

STANDARDS CONFORMANCE

`sigprocmask()`: AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

S

NAME

sigqueue() - queue a signal to a process

SYNOPSIS

```
#include <signal.h>

int sigqueue(pid_t pid, int signo, const union sigval value);
```

DESCRIPTION

The **sigqueue()** system call causes the signal specified by *signo* to be sent with the value specified by *value* to the process specified by *pid*. If *signo* is zero (the null signal), error checking is performed but no signal is actually sent. The null signal can be used to check the validity of *pid*.

The conditions required for a process to have permission to queue a signal to another process are the same as for the **kill()** system call.

The **sigqueue()** system call returns immediately. If **SA_SIGINFO** is set for *signo* at the receiving process (see *sigqueue*(2)) and if resources are available to queue the signal, the signal will be queued and sent to the receiving process. When the signal is delivered or accepted, the field *si_value* of the *siginfo* parameter (see **signal(5)**) will be set to *value.* If **SA_SIGINFO** is not set for *signo*, then *signo*, but not necessarily *value*, will be sent at least once to the receiving process.

If the value of *pid* causes *signo* to be generated for the sending process, and if *signo* is not blocked, either *signo* or at least one pending unblocked signal will be delivered to the sending process before the **sigqueue()** system call returns. Should any of multiple pending signals in the range **SIGRTMIN** to **SIGRTMAX** be selected for delivery or acceptance, it will be the lowest numbered one. The selection order between realtime and non-realtime signals, or between multiple pending non-realtime signals, is unspecified.

Application Usage

Threads Considerations

sigqueue() can be used to post signals to another process but can not be used to post signals to a specific thread in another process.

If the value of *pid* causes *signo* to be generated for the sending process, and if *signo* is not blocked for the calling thread and if no other thread has *signo* unblocked or is waiting in a **sigwait()** function for *signo*, either *signo* or at least one pending unblocked signal will be delivered to the calling thread before the **sigqueue()** function returns.

LWP Considerations

Signals can not be posted to specific Lightweight Processes (LWPs) in another process.

RETURN VALUE

Upon successful completion, the specified signal will be queued, and the **sigqueue()** function returns a value of **0** (zero). Otherwise, a value of **-1** is returned, and **errno** is set to indicate the error.

S

ERRORS

sigqueue() fails and no signal is sent if any of the following conditions occur:

[EAGAIN]	No resources are available to queue the signal. The process has already queued **{SIGQUEUE_MAX}** signals that are still pending at the receiver(s), or a systemwide resource limit has been exceeded.
[EINVAL]	The value of the *signo* argument is an invalid or unsupported signal number.
[EPERM]	The process does not have the appropriate privilege to send the signal to the receiving process.
[ESRCH]	The process *pid* does not exist.

SEE ALSO

kill(2), sysconf(2), signal(5).

NAME

sigsend(), sigsendset() - send a signal to a process or a group of processes

SYNOPSIS

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

#include <sys/procset.h>

int sigsend (idtype_t idtype, id_t id, int sig);

int sigsendset (const procset_t *psp, int sig);
```

DESCRIPTION

The **sigsend()** system call 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 *id* and *idtype*. The signal to be sent is specified by *sig* and is either one from the list given in **signal()** (see *signal*(2)) or 0.

If *sig* is equal to zero (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *id* and *idtype*.

The real or effective user ID of the sending process must match the real or effective user ID of the receiving process, unless the process has appropriate privileges, or *sig* is **SIGCONT** and the sending process has the same session ID as the receiving process.

idtype and *id* work together as follows:

- If *idtype* is **P_PID**, *sig* will be sent to the process with a process ID equal to **(pid_t)id**.
- If *idtype* is **P_PGID**, *sig* will be sent to any process with a process group ID equal to **(pid_t)id**.
- If *idtype* is **P_SID**, *sig* will be sent to any process with a session ID equal to **(pid_t)id**.
- If *idtype* is **P_UID**, *sig* will be sent to any process with an effective user ID equal to **(uid_t)id**.
- If *idtype* is **P_GID**, *sig* will be sent to any process with an effective group ID equal to **(gid_t)id**.
- If *idtype* is **P_ALL**, *sig* will be sent to all processes and *id* will be ignored.
- If *id* is **P_MYID**, the value of *id* is taken from the calling process.

The process with a process ID of **0** is always excluded. The process with a process ID of **1** is included only if *idtype* is equal to **P_PID**.

sigsendset() provides an alternate interface for sending signals to a set of processes.

psp is a pointer to a structure that includes the following members:

```
idop        p_op;
idtype_t    p_lidtype;
id_t        p_lid;
idtype_t    p_ridtype;
id_t        p_rid;
```

The structure defines a set of processes as the result of a set operation (difference, union, intersection, or exclusion) on two operands (*idtype*/*id* pairs). The left (right) operand is specified by **p_lid (p_rid)** and **p_lidtype (p_ridtype)**. **p_lid (p_rid)** takes the values specified by *id* and **p_lidtype (p_ridtype)** takes the values specified by *idtype* in the **sigsend()** system call defined above. *p_op* specifies the operand, and takes one of the following values:

POP_DIFF Set difference. The resultant set consists of the processes that are in the left operand and not in the right operand.

POP_AND Set intersection. The resultant set consists of the processes that are in both the left and right operands.

POP_OR Set union. The resultant set consists of the processes that are in either the left or right operand or both.

POP_XOR Set exclusive **OR**. The resultant set consists of the processes that are in either the left or right operand but not in both.

S

RETURN VALUE

Upon successful completion, **sigsend()** returns a value of **0**. Otherwise, it returns a value of –1 and sets **errno** to indicate the error.

ERRORS

If **sigsend()** fails, it sets errno (see *errno*(2)) to one of the following values:

[EINVAL] *sig* is neither a valid signal number nor zero.

[EINVAL] *idtype* is not a valid value.

[EINVAL] *sig* is **SIGKILL**, *idtype* is **P_PID**, and *id* is **1**.

[ESRCH] No process can be found corresponding to that specified by *id* and *idtype*.

[EPERM] The user ID of the sending process is not 0, and its real or effective user ID does not match the real or effective user ID of the receiving process, and the calling process is not sending **SIGCONT** to a process that shares the same session ID.

SEE ALSO

kill(2), signal(2).

NAME

sigsetmask - set current signal mask

SYNOPSIS

```
#include <signal.h>

long sigsetmask(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() 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

Do not use **sigsetmask()** in conjunction with the facilities described under *sigset*(3C).

APPLICATION USAGE

Threads Considerations

Since each thread maintains its own blocked signal mask, **sigsetmask()** modifies only the calling thread's blocked signal mask.

For more information regarding signals and threads, refer to *signal*(5).

LWP (Lightweight Processes) Considerations

sigsetmask() modifies only the calling LWP's blocked signal mask.

LP64 Programs

sigsetmask() accepts and returns long (64 bit) values. However, as for ILP32 programs, **sigsetmask()** supports signals numbered 1 through 32. The upper 32 bits of the mask argument are ignored. Also, the upper 32 bits of the returned mask have no meaning.

AUTHOR

sigsetmask() was developed by the University of California, Berkeley.

SEE ALSO

kill(2), sigblock(2), sigpause(3C), sigprocmask(2), sigvector(2).

NAME

sigspace - assure sufficient signal stack space

SYNOPSIS

```
#include <signal.h>

size_t sigspace(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()**, **sigvector()**, or **sigvec()** call (see *bsdproc*(3C)), 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. 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()** system call, but the guarantee of space is removed after any **exec()** system call (see *fork*(2) and *exec*(2)).

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()** (see *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, **sigspace()** returns the size of the former guaranteed space. Otherwise, it returns −1 and sets **errno** 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.

Do not use **sigspace()** in conjunction with the facilities described under *sigset*(3C).

Do not use **sigspace()** in conjunction with *sigstack*(2).

APPLICATION USAGE

Threads Considerations

Each thread may define an alternate signal handling stack.

LWP (Lightweight Processes) Considerations

Each LWP may define an alternate signal handling stack.

AUTHOR

sigspace() was developed by HP.

SEE ALSO

sigaction(2), sigstack(2), sigvector(2), malloc(3C), setjmp(3C).

S

NAME

sigstack - set and/or get signal stack context

SYNOPSIS

```
#include <signal.h>

int sigstack(
     struct sigstack *ss,
     struct sigstack *oss
);
```

DESCRIPTION

The **sigstack()** function 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.

If the *ss* argument is not a null pointer, it must point to a sigstack structure. The length of the application-supplied stack must be at least **SIGSTKSZ** bytes. If the alternate signal stack overflows, the resulting behaviour is undefined. (See APPLICATION USAGE below.)

- The value of the *ss_onstack* member indicates whether the process wants the system to use an alternate signal stack when delivering signals.
- The value of the *ss_sp* member indicates the desired location of the alternate signal stack area in the process' address space.
- If the *ss* argument is a null pointer, the current alternate signal stack context is not changed.

If the *oss* argument is not a null pointer, it points to a sigstack structure in in which the current alternate signal stack context is placed. The value stored in the *ss_onstack* member of *oss* will be non- zero if the process is currently executing on the alternate signal stack. If the *oss* argument is a null pointer, the current alternate signal stack context is not returned.

When a signal's action indicates its handler should execute on the alternate signal stack (specified by calling **sigaction()**), the implementation checks to see if the process is currently executing on that stack. If the process is not currently executing on the alternate signal stack, the system arranges a switch to the alternate signal stack for the duration of the signal handler's execution.

After a successful call to one of the **exec** functions, there are no alternate signal stacks in the new process image.

RETURN VALUE

Upon successful completion, **sigstack()** returns 0. Otherwise, it returns –1 and sets **errno** to indicate the error.

ERRORS

The **sigstack()** function will fail if:

[EPERM] An attempt was made to modify an active stack.

APPLICATION USAGE

A portable application, when being written or rewritten, should use **sigaltstack()** instead of **sigstack()**.

On some implementations, stack space is automatically extended as needed. On those implementations, automatic extension is typically not available for an alternate stack. If a signal stack overflows, the resulting behaviour of the process is undefined.

The direction of stack growth is not indicated in the historical definition of struct *sigstack*. The only way to portably establish a stack pointer is for the application to determine stack growth direction, or to allocate a block of storage and set the stack pointer to the middle. The implementation may assume that the size of the signal stack is **SIGSTKSZ** as found in **<signal.h>**. An implementation that would like to specify a signal stack size other than **SIGSTKSZ** should use **sigaltstack()**.

Programs should not use **longjmp()** to leave a signal handler that is running on a stack established with **sigstack()**. Doing so may disable future use of the signal stack. For abnormal exit from a signal handler, **siglongjmp()**, **setcontext()**, or **swapcontext()** may be used. These functions fully support switching from one stack to another.

The **`sigstack()`** function requires the application to have knowledge of the underlying system's stack architecture. For this reason, **`sigaltstack()`** is recommended over this function.

Threads Considerations

Each thread may define an alternate signal handling stack.

LWP (Lightweight Processes) Considerations

Each LWP may define an alternate signal handling stack.

SEE ALSO

exec(2), fork(2), setjmp(3C), sigaltstack(2), <signal.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

S

HP-UX EXTENSIONS

SYNOPSIS

```
int sigstack(
     const struct sigstack *ss,
     struct sigstack *oss
);
```

DESCRIPTION

The correct use of **sigstack()** is hardware dependent, and therefore is not portable between different HP-UX implementations. **sigspace()** is portable between different HP-UX implementations and should be used when the application does not need to know where the signal stack is located (see *sigspace*(2)). **sigstack()** is provided for compatibility 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.

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.

ERRORS

[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

Do not use *sigstack*(2) in conjunction with *sigspace*(2).

Methods for calculating the required stack size are not well developed.

Leaving the context of a service routine abnormally, such as by **longjmp()** (see *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.

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(const sigset_t *sigmask);
```

DESCRIPTION

The **sigsuspend()** function replaces the process' current signal mask with the set of signals pointed to by **sigmask** and then suspends the process until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process.

If the action is to terminate the process then **sigsuspend()** will never return. If the action is to execute a signal-catching function, then **sigsuspend()** will return after the signal-catching function returns, with the signal mask restored to the set that existed prior to the **sigsuspend()** call.

It is not possible to block signals that cannot be ignored. This is enforced by the system without causing an error to be indicated.

RETURN VALUE

Since **sigsuspend()** suspends process execution indefinitely, there is no successful completion return value. If a return occurs, −1 is returned and **errno** is set to indicate the error.

ERRORS

The **sigsuspend()** function will fail if:

[EINTR] A signal is caught by the calling process and control is returned from the signal-catching function.

APPLICATION USAGE

Threads Considerations

Since blocked signal masks are maintained at the thread level, **sigsuspend()** modifies only the calling thread's blocked signal mask. **sigsuspend()** suspends only the calling thread until it receives a signal.

If other threads in the process do not block the signal, the signal may be delivered to another thread in the process and the thread in **sigsuspend()** may continue waiting. For this reason, the use of *sigwait*(2) is recommended instead of **sigsuspend()** for multi-threaded applications.

For more information regarding signals and threads, refer to *signal*(5).

LWP (Lightweight Processes) Considerations

sigsuspend() modifies only the calling LWP's signal mask and suspends only the calling LWP until receipt of a signal.

S

SEE ALSO

pause(2), sigaction(2), sigsetops(3C), sigwait(2), <signal.h>.

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 4

The following change is incorporated for alignment with the ISO POSIX-1 standard:

- The type of the argument *sigmask* is changed from *sigset_t** to type *const sigset_t**.

Another change is incorporated as follows:

- The term "signal handler" is changed to "signal-catching function."

HP-UX EXTENSIONS

ERRORS

[EFAULT] *sigmask* points to an invalid address. The reliable detection of this error is implementation-dependent.

AUTHOR

sigsuspend() was derived from the IEEE POSIX 1003.1-1988 Standard.

SEE ALSO

sigaction(2), sigpending(2), sigprocmask(2), sigsetops(3C), signal(5).

STANDARDS CONFORMANCE

sigsuspend() : AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

sigvector - software signal facilities

SYNOPSIS

```
#include <signal.h>

int sigvector(
     int sig,
     const struct sigvec *vec,
     struct sigvec *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 entry, along with those for *sigblock*(2), *sigsetmask*(2), *sigpause*(3C), and *sigspace*(2), defines an alternate mechanism for handling these signals that ensures the delivery of signals and the 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 **sigspace()** (see *sigspace*(2)).

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()**, **sigsetmask()**, or **sigpause()** 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()** or **sigsetmask()** call is made). This mask is formed by taking the current signal mask, computing the bit-wise 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.

S

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()** 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** (see *csh*(1)) when maintaining process groups under Berkeley job control.

If **SV_RESETHAND** is set, the signal handler is installed with the same semantics as a handler installed with *signal*(2). This affects 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()** system call is performed (see *exec*(2)). 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()** or **write()** system call (see *read*(2) or *write*(2)), 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.

S

Exiting the handler abnormally (such as with **longjmp()** — see *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()**, **write()**, or **ioctl()** on fast devices (such as disks) cannot be interrupted, but I/O to a slow device (such as a printer) 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**
close (slow requests)	**SYS_CLOSE**
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 the *X/Open Portability Guide, Issue 2* specifies a different meaning for the symbol **SYS_OPEN** (see *limits*(5)).

After a **fork()** or **vfork()** 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()** or **wait3()**. Otherwise the presence of the child that caused the original signal always causes another signal to be delivered.

RETURN VALUE

Upon successful completion, **sigvector()** returns 0; otherwise, it returns −1 and sets **errno** to indicate the reason.

ERRORS

sigvector() fails and no new signal handler is installed if any of the following conditions are encountered:

[EFAULT] Either *vec* or *ovec* points to memory that is not a valid part of the process address space. Reliable detection of this error is implementation dependent.

[EINVAL] *sig* is not a valid signal number.

[EINVAL] An attempt was 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*(3C).

APPLICATION USAGE

Threads Considerations

The signal disposition (such as **catch/ignore/default**) established by **`sigvector()`** is shared by all threads in the process. Each thread maintains its own blocked signal mask. For more information regarding signals and threads, refer to *signal*(5).

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(3C), sigsetmask(2), sigspace(2), setjmp(3C), signal(5), termio(7).

NAME

sigwait(), sigwaitinfo(), sigtimedwait() - synchronously accept a signal

SYNOPSIS

```
#include <signal.h>

int sigwait(const sigset_t *set, int *sig);

int sigwaitinfo(const sigset_t *set, siginfo_t *info);

int sigtimedwait(const sigset_t *set, siginfo_t *info,
   const struct timespec *timeout);
```

DESCRIPTION

The **sigwait()** function atomically selects and clears a pending signal from *set* and returns the signal number in the location pointed to by *sig*. If none of the signals in *set* is pending at the time of the call, the calling thread will be suspended until one or more signals become pending or the thread is interrupted by an unblocked, caught signal. The signals in *set* should be blocked at the time of the call to **sigwait()**. Otherwise, the behavior is undefined.

If there are multiple signals queued for the selected signal number, **sigwait()** will return with the first queued signal and the remainder will remain queued. If any of multiple pending signals in the range **SIGRTMIN** to **SIGRTMAX** is selected, the lowest numbered signal will be returned. The selection order between realtime and nonrealtime signals, or between multiple pending nonrealtime signals, is unspecified.

If more than one thread in a process is in **sigwait()** for the same signal, only one thread will return from **sigwait()** with the signal number; which thread returns is undefined.

sigwaitinfo() has the same behavior as **sigwait()** if the *info* parameter is **NULL**. If the *info* parameter is not **NULL**, **sigwaitinfo()** has the same behavior as **sigwait()**, except that the selected signal number is returned in the *si_signo* field of the *info* parameter and the cause of the signal is returned in the *si_code* field. If any value is queued to the selected signal, the first such queued value will be dequeued and stored in the *si_value* member of *info* and the system resource used to queue the signal will be released and made available to queue other signals. If no value is queued, the contents of the *si_value* member is undefined. If no further signals are queued for the selected signal, the pending indication for that signal will be reset.

sigtimedwait() has the same behavior as **sigwaitinfo()** except that **sigtimedwait()** will only wait for the time interval specified by the *timeout* parameter if none of the signals specified by *set* are pending at the time of the call. If the *timeout* parameter specifies a zero-valued time interval, then **sigtimedwait()** will return immediately with an error if no signals in *set* are pending at the time of the call. If the *timeout* parameter is **NULL**, the behavior is undefined.

APPLICATION USAGE

For a given signal number, the sigwait family of routines should not be used in conjunction with **sigaction()** or any other functions which change signal action. If they are used together, the results are undefined.

S

Threads Considerations

The sigwait family of routines enable a thread to synchronously wait for signals. This makes the sigwait routines ideal for handling signals in a multithreaded process. The suggested method for signal handling in a multithreaded process is to have all threads block the signals of interest and dedicate one thread to call a sigwait function to wait for the signals. When a signal causes a sigwait function to return, the code to handle the signal can be placed immediately after the return from the sigwait routine. After the signal is handled, a sigwait function can again be called to wait for another signal.

In order to ensure that the dedicated thread handles the signal, it is essential that all threads, including the thread issuing the sigwait call, block the signals of interest. Otherwise, the signal could be delivered to a thread other than the dedicated signal handling thread. This could result in the default action being carried out for the signal. It is important that the thread issuing the sigwait call also block the signal. This will prevent signals from carrying out the default signal action while the dedicated signal handling thread is between calls to a sigwait function.

RETURN VALUE

Upon successful completion, **sigwait()** stores the signal number selected in the location pointed to by *sig* and returns with a value of **0** (zero). Otherwise, it returns an error number to indicate the error. The **errno** variable is NOT set if an error occurs.

Upon successful completion, **sigwaitinfo()** and **sigtimedwait()** will return the selected signal number. Otherwise a value of -1 is returned and **errno** is set to indicate the error.

ERRORS

If any of the following conditions occur, the sigwait family of routines will return the following error number:

[EAGAIN] **sigtimedwait()** was called and no signal in the *set* parameter was delivered within the time interval specified by the *timeout* parameter.

If any of the following conditions occur and the condition is detected, the sigwait family of routines will fail and return the following error number:

[EINVAL] *set* contains an invalid or unsupported signal number.

[EINVAL] **sigtimedwait()** was called and the *timeout* parameter specified a *tv_nsec* value less than zero or greater than or equal to 1000 million, or a *tv_sec* value less than zero or greater than or equal to 2147483648 (that is, a value too large to be represented as a signed 32-bit integer).

[EINTR] The wait was interrupted by an unblocked, caught signal.

[EFAULT] At least one of the *set*, *sig*, *info*, or *timeout* parameters references an illegal address.

AUTHOR

sigwaitinfo() and **sigtimedwait()** were derived from the IEEE POSIX P1003.1b standard.

sigwait() was derived from the IEEE POSIX P1003.1c standard.

SEE ALSO

pause(2), sigaction(2), sigpending(2), sigsuspend(2), pthread_sigmask(3T), signal(5).

STANDARDS CONFORMANCE

sigwait(): POSIX.1c

sigwaitinfo(): POSIX.1b

sigtimedwait(): POSIX.1b

NAME

socket() - create an endpoint for communication

SYNOPSIS

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

AF_CCITT Only

```
#include <x25/x25ccittproto.h>

int socket(int af, int type, int protocol);
```

DESCRIPTION

The **socket()** system call creates an endpoint for communication and returns a descriptor. The socket descriptor returned is used in all subsequent socket-related system calls.

The *af* parameter specifies an address family to be used to interpret addresses in later operations that specify the socket. These address families are defined in the include files **<sys/socket.h>** and **<x25/ccittproto.h>**. The only currently supported address families are:

AF_INET	(DARPA Internet addresses)
AF_UNIX	(path names on a local node)
AF_CCITT	(CCITT X.25 addresses)
AF_VME_LINK	(backplane communications on VMEbus)

The *type* specifies the semantics of communication for the socket. Currently defined types are:

SOCK_STREAM	Sequenced, reliable, two-way-connection-based byte streams.
SOCK_DGRAM	Datagrams (connectionless, unreliable messages of a fixed, typically small, maximum length; for AF_INET only).

protocol specifies a particular protocol to be used with the socket. Normally, only a single protocol exists to support a particular socket type using a given address family. However, many protocols may exist, in which case a particular protocol must be specified. The protocol number to use depends on the **communication domain** in which communication is to take place (see *services*(4) and *protocols*(4)). *protocol* can be specified as zero, which causes the system to choose a protocol type to use.

Sockets of type SOCK_STREAM are byte streams similar to pipes, except that they are full-duplex instead of half-duplex. A stream socket must be in a *connected* state before any data can be sent or received on it. A connection to another socket is created with a **connect()** or **accept()** call. Once connected, data can be transferred using some variant of the **send()** and **recv()** or the **read()** and **write()** calls. When a session is complete, use **close()** or **shutdown()** calls to terminate the connection.

TCP, the communications protocol used to implement SOCK_STREAM for AF_INET sockets, ensures that data is not lost or duplicated. If a peer has buffer space for data and the data cannot be successfully transmitted within a reasonable length of time, the connection is considered broken and the next **recv()** call indicates an error with **errno** set to [ETIMEDOUT]. If **SO_KEEPALIVE** is set and the connection has been idle for two hours, the TCP protocol sends "keepalive" packets every 75 seconds to determine whether the connection is active. These transmissions are not visible to users and cannot be read by a **recv()** call. If the remote system does not respond within 10 minutes (i.e., after 8 "keepalive" packets have been sent), the next socket call (e.g., **recv()**) returns an error with **errno** set to [ETIMEDOUT]. A **SIGPIPE** signal is raised if a process sends on a broken stream. This causes naive processes that do not handle the signal to exit. An end-of-file condition (zero bytes read) is returned if a process tries to read on a broken stream.

SOCK_DGRAM sockets allow sending of messages to correspondents named in **send()** calls. It is also possible to receive messages at such a socket with **recv()**.

The operation of sockets is controlled by socket level options set by the **setsockopt()** system call described by the *getsockopt*(2) manual entry. These options are defined in the file **<sys/socket.h>** and explained in the *getsockopt*(2) manual entry.

X.25 Only

Socket endpoints for communication over an X.25/9000 link can be in either address family, AF_INET or AF_CCITT. If the socket is in the AF_INET family, the connection behaves as described above. TCP is used if the socket type is SOCK_STREAM. UDP is used if the socket type is SOCK_DGRAM. In both cases, Internet protocol (IP) and the X.25-to-IP interface module are used.

If the socket is in the AF_CCITT address family, only the SOCK_STREAM socket type is supported. Refer to the topic "Comparing X.25 Level 3 Access to IP" in the *X.25 Programmer's Guide* for more details on the difference between programmatic access to X.25 via IP and X.25 Level 3.

If the socket is in the AF_CCITT family, the connection and all other operations pass data directly from the application to the X.25 Packet Level (level 3) without passing through a TCP or UDP protocol. Connections of the AF_CCITT family cannot use most of the socket level options described in *getsockopt*(2). However, AF_CCITT connections can use many X.25-specific **ioctl()** calls, described in *socketx25*(7).

DEPENDENCIES

AF_CCITT and AF_VME_LINK

Only the SOCK_STREAM type is supported.

RETURN VALUE

socket() returns the following values:

n	Successful completion. *n* is a valid file descriptor referring to the socket.
-1	Failure. **errno** is set to indicate the error.

ERRORS

If **socket()** fails, **errno** is set to one of the following values.

[EAFNOSUPPORT]	The specified address family is not supported in this version of the system.
[EHOSTDOWN]	The networking subsystem is not up.
[EINVAL]	SOCK_DGRAM sockets are currently not supported for the AF_UNIX or AF_VME_LINK address families.
[EMFILE]	The per-process descriptor table is full.
[ENFILE]	The system's table of open files is temporarily full and no more **socket()** calls can be accepted.
[ENOBUFS]	No buffer space is available. The socket cannot be created.
[ENOMEM]	No memory is available. The socket cannot be created.
[EPROTONOSUPPORT]	The specified protocol is not supported.
[EPROTOTYPE]	The type of socket and protocol do not match.
[ESOCKTNOSUPPORT]	The specified socket type is not supported in this address family.
[ETIMEDOUT]	Connection timed out.

FUTURE DIRECTION

Currently, the default behavior is the **HP-UX BSD Sockets**; however, it might be changed to **X/Open Sockets** in a future release. At that time, any **HP-UX BSD Sockets** behavior that is incompatible with **X/Open Sockets** might be obsoleted. Applications that conform to the X/Open specification now will avoid migration problems (see *xopen_networking*(7)).

S

MULTITHREAD USAGE

The **socket()** system call is thread-safe. It has a cancellation point; and it is async-cancel safe, async-signal safe, and fork-safe.

AUTHOR

socket() was developed by HP and the University of California, Berkeley.

SEE ALSO

accept(2), bind(2), connect(2), getsockname(2), getsockopt(2), ioctl(2), listen(2), recv(2), select(2), send(2), shutdown(2), af_ccitt(7F), af_vme_link(7F), socket(7), socketx25(7), tcp(7P), udp(7P), unix(7P), xopen_networking(7).

STANDARDS CONFORMANCE

socket(): XPG4

NAME

socketpair() - create a pair of connected sockets

SYNOPSIS

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

int socketpair(int af, int type, int protocol, int sv[2]);
```

DESCRIPTION

The **socketpair()** system call creates an unnamed pair of connected sockets and returns two file descriptors in *sv*[0] and *sv*[1]. The two sockets are indistinguishable. *af* specifies the address family. See *socket*(2). *type* specifies the semantics of communication for the socket. *protocol* specifies a particular protocol to be used. *protocol* can be specified as zero, which causes the system to choose a protocol type to use.

RETURN VALUE

socketpair() returns the following values:

- **0** Successful completion.
- **-1** Failure. **errno** is set to indicate the error.

ERRORS

If **socketpair()** fails, **errno** is set to one of the following values.

[EAFNOSUPPORT] The specified address family is not supported in this version of the system.

[EFAULT] The *sv* parameter is not valid.

[EMFILE] The per-process file descriptor table is full.

[ENFILE] The system file table is temporarily full.

[ENOBUFS] No buffer space is available for the operation to complete.

[EOPNOTSUPP] The specified protocol does not support creation of socket pairs.

[EPROTONOSUPPORT]
The specified protocol is not supported in this version of the system.

DEPENDENCIES

socketpair() is supported only for AF_UNIX.

FUTURE DIRECTION

Currently, the default behavior is the **HP-UX BSD Sockets**; however, it might be changed to **X/Open Sockets** in a future release. At that time, any **HP-UX BSD Sockets** behavior that is incompatible with **X/Open Sockets** might be obsoleted. Applications that conform to the X/Open specification now will avoid migration problems (see *xopen_networking*(7)).

MULTITHREAD USAGE

The **socketpair()** system call is thread-safe. It has a cancellation point; and it is async-cancel safe, async-signal safe, and fork-safe.

AUTHOR

socketpair() was developed by HP and the University of California, Berkeley.

SEE ALSO

read(2), socket(2), write(2), xopen_networking(7).

STANDARDS CONFORMANCE

socketpair(): XPG4

NAME

stat - get file status

SYNOPSIS

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

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

DESCRIPTION

The **stat()** function obtains information about the named file and writes it to the area pointed to by the *buf* argument. The path argument points to a pathname naming a file. Read, write or execute permission of the named file is not required, but all directories listed in the pathname leading to the file must be searchable. An implementation that provides additional or alternate file access control mechanisms may, under implementation-dependent conditions, cause **stat()** to fail. In particular, the system may deny the existence of the file specified by *path*.

The *buf* argument is a pointer to a **stat** structure, as defined in the header **<sys/stat.h>**, into which information is placed concerning the file.

The **stat()** function updates any time-related fields (as described in the definition of File Times Update in the XBD specification), before writing into the **stat** structure.

The structure members *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atime*, *st_ctime*, and *st_mtime* will have meaningful values for all file types defined in this document. The value of the member *st_nlink* will be set to the number of links to the file.

RETURN VALUE

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

ERRORS

The **stat()** function will fail if:

[EACCES]	Search permission is denied for a component of the *path* prefix.
[EIO]	An error occurred while reading from the file system.
[ELOOP]	Too many symbolic links were encountered in resolving path.
[ENAMETOOLONG]	The length of the *path* argument exceeds **{PATH_MAX}** or a pathname component is longer than **{NAME_MAX}**.
[ENOENT]	A component of *path* does not name an existing file or path is an empty string.
[ENOTDIR]	A component of the *path* prefix is not a directory.

The **stat()** function may fail if:

[ENAMETOOLONG]	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds **{PATH_MAX}**.
[EOVERFLOW]	A value to be stored would overflow one of the members of the *stat* structure.

SEE ALSO

fstat(2), lstat(2), <sys/stat.h>, <sys/types.h>.

CHANGE HISTORY

First released in Issue 1.

Derived from Issue 1 of the SVID.

Issue 4

The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- The type of argument *path* is changed from **char *** to **const char ***.
- In the DESCRIPTION section, (a) statements indicating the purpose of this interface and a paragraph defining the contents of **stat** structure members are added, and (b) the words "extended security controls" are replaced by "additional or alternate file access control mechanisms."

The following change is incorporated for alignment with the FIPS requirements:

- In the ERRORS section, the condition whereby **[ENAMETOOLONG]** will be returned if a pathname component is larger that **{NAME_MAX}** is now defined as mandatory and marked as an extension.

Another change is incorporated as follows:

- The header **<sys/types.h>** is now marked as optional (OH); this header need not be included on XSI-conformant systems.

Issue 4, Version 2

The ERRORS section is updated for X/OPEN UNIX conformance as follows:

- In the mandatory section, **EIO** is added to indicate that a physical I/O error has occurred, and **ELOOP** to indicate that too many symbolic links were encountered during pathname resolution.
- In the optional section, a second **ENAMETOOLONG** condition is defined that may report excessive length of an intermediate result of pathname resolution of a symbolic link.
- In the optional section, **EOVERFLOW** is added to indicate that a value to be stored in a member of the **stat** structure would cause overflow.

S

HP-UX EXTENSIONS

DESCRIPTION

If the chosen path name or file descriptor refers to a Multi-Level Directory (MLD), and the process does not have the multilevel effective privilege, the *i-node* number returned in *st_ino* is the *i-node* of the MLD itself.

The parameters for the **stat()** function are as follows:

path is a pointer to a path name of any file within the mounted file system.(All directories listed in the path name must be searchable.)

buf is a pointer to a **stat** structure, which is where the file status information is stored.

The **stat** structure contains the following members:

```
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 sysfs(2) */
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 */
long    st_blksize;   /* File system block size */
uint    st_acl:1;     /* Set if the file has optional */
                      /* access control list entries */
                      /* HFS File Systems only */
```

(Note that the position of items in this list does not necessarily reflect the order of the members in the structure.)

S

ERRORS

[EFAULT] *buf* or *path* points to an invalid address. The reliable detection of this error is implementation dependent.

[EOVERFLOW] The file size in bytes or the number of blocks allocated to the file cannot be represented correctly in the structure pointed to by *buf*.

NFS

The *st_basemode* and *st_acl* fields are zero on files accessed remotely. *st_acl* field is applicable to HFS File Systems only.

WARNINGS

Access Control Lists - HFS File Systems only

Access control list descriptions in this entry apply only to HFS file systems on standard HP-UX operating systems.

DEPENDENCIES

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.

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), stat64(2), sysfs(2), time(2), truncate(2), unlink(2), utime(2), write(2), acl(5), stat(5).

STANDARDS CONFORMANCE

stat(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

statfs, fstatfs - get file system statistics

SYNOPSIS

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

int statfs(const char *path, struct statfs *buf);

int fstatfs(int fildes, struct statfs *buf);
```

DESCRIPTION

statfs() returns status information for a mounted file system.

fstatfs() returns similar information for an open file.

The parameters for the **statfs()** and **fstatfs()** functions are as follows:

path is a pointer to a path name of any file within the mounted file system.

buf is a pointer to a **statfs()** structure, which is where the file system status information is stored.

fildes is a file descriptor for an open file, which is created with the successful completion of an **open()**, **creat()**, **dup()**, **fcntl()**, or **pipe()** system call (see *open*(2), *creat*(2), *dup*(2), *fcntl*(2), or *pipe*(2)).

The **statfs()** structure contains 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.  f_fsid[1] is the file system
                       type; see sysfs(2) */
```

The fields **f_blocks** , **f_bavail** and **f_bfree** are expressed in terms of blocks of size **f_bsize.**

A *file node* is a structure in the file system hierarchy that describes a file.

Fields that are undefined for a particular file system are set to −1.

RETURN VALUE

statfs() and **fstatfs()** return 0 upon successful completion; otherwise, they return −1 and set **errno** to indicate the error.

S

ERRORS

If **statfs()** fails, **errno** is set to one of the following values:

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

[EFAULT] *buf* or *path* point 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 during path-name translation.

[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.

If **fstatfs()** fails, **errno** is set to one of the following values:

[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).

S

NAME

statvfs, fstatvfs - get file system information

SYNOPSIS

```
#include <sys/types.h>
#include <sys/statvfs.h>
int statvfs (const char *path, struct statvfs *buf);
int fstatvfs (int fildes, struct statvfs *buf);
```

DESCRIPTION

statvfs() returns information about a mounted file system.

fstatvfs() returns similar information about an open file.

The parameters for the **statvfs()** and **fstatvfs()** functions are as follows:

path is a pointer to a path name of any file within the mounted file system.

buf is a pointer to a **statvfs()** structure, which is where the file system status information is stored.

fildes is a file descriptor for an open file, which is created with the successful completion of an **open()**, **creat()**, **dup()**, **fcntl()**, or **pipe()** system call (see *open*(2), *creat*(2), *dup*(2), *fcntl*(2), or *pipe*(2)).

The **statvfs()** structure contains the following members:

```
ulong f_bsize;              /* preferred file system block size */
ulong f_frsize;             /* fundamental file system block size */
ulong f_blocks;             /* total blocks of f_frsize on file system */
ulong f_size;               /* size of file system in f_frsize unit */
ulong f_bfree;              /* free blocks */
ulong f_bavail;             /* blocks available to non-superuser */
long  f_files;              /* total file nodes in file system */
long  f_ffree;              /* free file nodes in file system */
long  f_favail;             /* file nodes available to non-superuser */
long  f_fsid;               /* file system ID for file system */
                            /* type; see sysfs(2) */
char  f_basetype[FSTYPSZ];  /* file system type name is null-terminated */
long  f_flag;               /* bit mask of flags */
long  f_namemax             /* maximum file name length */
char  f_fstr[32];           /* file system specific string */
time_t  f_time;             /* Last time file system was written */
```

S

The field f_basetype contains a null-terminated file-system-type name.

The constant **[FSTYPSZ]** is defined in the header file **<statvfs.h>**.

The following flags can be returned in the f_flag field:

ST_LARGEFILES File system is enabled for large files.

ST_RDONLY File system is read-only.

ST_NOSUID File system does not support **setuid** and **setgid** semantics.

ST_EXPORTED File system is exported (NFS).

ST_QUOTA Quotas are enabled on this file system.

RETURN VALUE

statvfs() and **fstatvfs()** return 0 upon successful completion; otherwise, they return −1 and set **errno** to indicate the error.

ERRORS

If **statvfs()** fails, **errno** is set to one of the following values:

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

[ELOOP] Too many symbolic links are encountered during path-name translation.

[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.

If **fstatvfs()** fails, **errno** is set to the following value:

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

When both **statvfs()** and **fstatvfs()** fail, **errno** is set to one of the following values:

[EFAULT] *buf* points to an invalid address.

[EIO] An I/O error occurred while reading from or writing to the file system.

SEE ALSO

df(1M), fstatfs(2), fstatvfs64(2), quotactl(2), stat(2), statfs(2), statvfs64(2), sysfs(2), ustat(2).

NAME

stime() - set time and date

SYNOPSIS

```
#include <time.h>

int stime(const time_t *tp);
```

DESCRIPTION

The **stime()** system call sets the system time and date. *tp* points to the value of time as measured in seconds from 00:00:00 on January 1, 1970, Coordinated Universal Time (UTC).

RETURN VALUE

stime() returns the following values:

- **0** Successful completion.
- **-1** Failure. **errno** is set to indicate the error.

ERRORS

If **stime()** fails, **errno** is set to one of the following values.

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

SEE ALSO

date(1), gettimeofday(2), time(2).

STANDARDS CONFORMANCE

stime(): SVID2, SVID3, XPG2

S

NAME

stream - STREAMS enhancements to standard system calls

DESCRIPTION

The **open()**, **close()**, **read()**, **readv()**, **write()**, **writev()**, **ioctl()**, **select()**, and **signal()** system calls are enhanced to support STREAMS. The new functionality is described below for each system call.

Open Enhancements

When calling open for a STREAMS device, the *oflag* parameter can only be constructed from the **O_NONBLOCK** flag values that are OR-ed with the **O_RDONLY**, **O_WRONLY**, or **O_RDWR** flag values. The values of the other flags are not applicable to STREAMS devices and have no effect on them.

The values of the **O_NONBLOCK** flags affect the operations of STREAMS-based device drivers, when the **read()**, **write()**, **getmsg()**, **getpmsg()**, **putmsg()**, or **putpmsg()** functions are used. After the stream is open, these flags can be modified by calling **fcntl()** (see the *fcntl*(2) man page). The effects of the flags are device specific.

The open of a STREAMS device may fail for one or more of the following STREAMS-specific conditions:

EIO A hangup occurred while the **open()** function was attempting to open the stream.

EAGAIN The system was unable to allocate a stream.

ENODEV The device has not been generated into the system as a STREAMS device.

ENXIO The open routine of one of the modules or drivers in the stream failed.

Close Enhancements

When all file descriptors associated with a STREAMS device have been closed, the stream is dismantled. If the file descriptor is associated with a stream that is subject to persistent links, the **close()** function will succeed immediately, but the stream will remain open. See **I_PLINK** documentation in *streamio*(7). Dismantling includes popping any modules on the stream and closing the driver. If **O_NONBLOCK** flag is set, and there are no signals posted for the stream, the **close()** function waits for output to drain on each module's or driver's non-empty write queue. **close()** waits for each module or driver for the amount of time set by the **I_SETCLTIME ioctl()** (see the *streamio*(7) man page). The default is 15 seconds per module or driver. If the **O_NONBLOCK** flag is set, or there are any pending signals, the function does not wait for output to drain and dismantles the stream immediately. If a STREAMS device is closed, and the calling process had previously registered to recieve a **SIGPOLL** signal for events associated with that device (see "Signal Enhancements" below), **close()** unregisters the calling process for the events associated with the stream.

Read and Readv Enhancements

In this section, **read()** refers to both **read()** and **readv()**. For STREAMS devices, the **read()** function operates in accordance with the read mode of the file. STREAMS has three read modes: byte-stream mode, message-nondiscard mode, and message-discard mode. The default is byte-stream mode; however, the user can change this by issuing the **I_SRDOPT ioctl()** call. The user can also test for the current read mode by issuing the **I_GRDOPT ioctl()** call. See the *streamio(7)* man page for more information about these **ioctl()** calls. The **read()** function's behavior in each of the read modes of a STREAMS device is as follows:

- In byte-stream mode, the function retrieves data from the stream associated with the file descriptor until it has retrieved *nbyte* bytes, or until there is no more data to be retrieved.
- In message-nondiscard mode, the function retrieves data until it reaches a message boundary. If it does not retrieve all of the data in the message, it places the remaining data back on the stream. This data can be retrieved by a subsequent **read()**, **getmsg()**, or **getpmsg()** call.
- In message-discard mode, the function retrieves data until it has retrieved *nbytes*, or until it has reached a message boundary. However, unread data remaining in the message is discarded and is not available for reading by a subsequent **read()**, **getmsg()**, or **getpmsg()** call.

When attempting to read a STREAMS device and encountering a zero-byte message:

- If the read mode is byte-stream, the **read()** function returns the number of bytes of data read before encountering the zero-byte message. If data was read before receiving the zero-byte message, **read()** returns the zero-byte message to the stream so it can be processed by a subsequent **read()**, **getmsg()**, or **getpmsg()** call. If no data was read, **read()** consumes the message.

S

- If the read mode is message-discard or message-nondiscard, the **read()** function returns zero, and then consumes the message.

The **read()** function reads the data at the front of the stream head read queue. It reads both priority band and normal data.

The **read()** function processes control messages according to the STREAMS read flags: **RPROTNORM**, **RPROTDAT**, and **RPROTDIS**. The default is for **RPROTNORM** to be set; however, the user can change this by issuing the **I_SRDOPT ioctl()** call. The **read()** function's behavior for each read flag is described below:

- If **RPROTNORM** is set, a read from a stream can only process data messages. It cannot process any type of control message and fails if such a message is encountered at the stream head.
- If **RPROTDAT** is set, **read()** processes both data and control messages. The **read()** function delivers data in both data and control messages.
- If **RPROTDIS** is set, **read()** consumes any control messages and retrieves data from data messages only.

The following is also true for reads to STREAMS devices. If the **O_NONBLOCK** flag is clear, and no message is waiting to be read on the stream, the **read()** function blocks until a message arrives at the stream head. If the **O_NONBLOCK** flag is set, and no message is waiting to be read on the stream, the **read()** function fails and returns ERANGE.

A read from a STREAMS device may fail for one or more of the following STREAMS-specific conditions:

EAGAIN No message is waiting to be read on the stream, and the **O_NONBLOCK** flag is set.

EBADMSG A message is waiting to be read, but it is not a data message and the **RPROTNORM** flag is set.

EINVAL The stream is linked to a multiplexor.

A read from a STREAMS device also fails if an error message is received at the stream head. In this case, **errno** is set to the value returned in the error message.

If a hangup occurs on the stream being read, the **read()** function continues its operations until the stream read queues are empty. Thereafter, it returns a value of 0 (zero).

Write and Writev Enhancements

In this section, **write()** refers to both **write()** and **writev()**. When writing to a STREAMS device, the **write()** function sends ordinary, priority band zero, data. Other aspects of the **write()** function's behavior are determined by the packet size that the stream will accept.

If *nbytes* is not within the top module's minimum and maximum packet size range, **write()** will return ERANGE. Two exceptions exist, however, in which **write()** does not return an error. The first exception is if *nbytes* is too large and either the maximum packet size is infinite or the minimum packet size is less than or equal to zero. The second exception occurs if *nbytes* is too small and the minimum packet size is less than or equal to zero. With either exception, **write()** does not return ERANGE, and transfers the data.

S

The **write()** function may send the user's data buffer in multiple messages. The maximum amount of data that **write()** sends in one message is the lower value of the top module's maximum packet size and **STRMSGSZ**. If the maximum packet size is infinite, **write()** compares half of the top module's high water mark to **STRMSGSZ** instead. If the high water mark is less than or equal to zero, the page size is used.

If a zero-length buffer (*nbytes* is 0) is passed to **write()**, zero bytes are sent to the stream and zero bytes are returned.

The following is also true for writes to STREAMS devices. If the **O_NONBLOCK** flag is clear, and the stream cannot accept data (the stream head write queue is full due to flow control conditions), the **write()** function blocks until data can be accepted. If the **O_NONBLOCK** flag is set, and the stream cannot accept data, the **write()** function fails, and returns EAGAIN. If the **O_NONBLOCK** flag is set, and the stream cannot accept data, but part of the buffer has already been written, the **write()** function terminates and returns the number of bytes written.

A write to a STREAMS device may fail for one or more of the following STREAMS-specific conditions:

EAGAIN The **O_NONBLOCK** flag is set, and the stream cannot accept **write()** data because it is flow controlled.

EINVAL The **write()** function attempts to write to a stream that is linked below a multiplexor.

ENXIO A hangup occurs on a stream while the **write()** function is writing to the stream.

ERANGE The *nbytes* parameter is not within the allowable range.

The **write()** system call will also fail if an error message has been received at the stream head of the stream to which the **write()** function is attempting to write. In this case, the function returns with **errno** set to the value included in the error message.

Ioctl Enhancements

Refer to the *streamio*(7) man page for a description of STREAMS **ioctl()** functionality.

Select Enhancements

The **select()** system call checks the status of STREAMS devices. **select()** does not provide as much information for STREAMS devices as **poll()**. A program calls **select()** so that it can wait for events on both STREAMS and non-STREAMS devices. If **select()** returns an event for a STREAMS device, the program can call **poll()** to get more information. Refer to the *poll*(2) man page for more information about **poll()**.

select() returns a read event if a **poll() POLLIN**, **POLLERR**, **POLLNVAL** or **POLLHUP** event exists on the stream. In other words, **select()** returns a read event if a normal or priority band message is waiting to be read, if a read error exists at the stream head, if a write error exists at the stream head, if the stream is linked under a multiplexor, or if a hang-up has occurred.

select() returns a write event if a **poll() POLLOUT**, **POLLWRNORM**, **POLLERR**, or **POLLNVAL** event exists on the stream. This means that **select()** returns a write event if normal data can be written without blocking because of flow control, a read error exists at the stream head, a write error exists at the stream head, or the stream is linked under a multiplexor.

select() returns an exception event if a **poll() POLLPRI** event exists on the stream. More specifically, **select()** returns an exception event if a high-priority message is waiting to be read.

Signal Enhancements

A new signal, **SIGPOLL**, has been added for STREAMS. Processes register to receive a **SIGPOLL** signal for events that occur on a STREAMS device (see the *signal*(2) man page and **I_SETSIG** in the *streamio*(7) man page). The default action is to ignore the signal, not to terminate the process.

SEE ALSO

close(2), fcntl(2), getmsg(2), open(2), poll(2), putmsg(2), read(2), signal(2), select(2), write(2), streamio(7), and *STREAMS/UX for HP9000 Reference Manual*.

NAME

stty(), gtty() - control terminal device (Bell Version 6 compatibility)

SYNOPSIS

```
#include <sgtty.h>

int stty(int fildes, const struct sgttyb *argp);

int gtty(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** and **TCGETA** **ioctl()** calls described in *termio*(7).

DESCRIPTION

For certain status settings and status inquiries about terminal devices, the functions **stty()** and **gtty()** are equivalent to

```
ioctl(fildes, TIOCSETP, argp)
```

and

```
ioctl(fildes, TIOCGETP, argp)
```

respectively (see *ioctl*(2) and *termio*(7).

RETURN VALUE

gtty() and **stty()** return the following values:

0 Successful completion.
-1 Failure. **errno** is set to indicate the error.

ERRORS

If **gtty()** or **stty()** fails, **errno** is set to one of the following values:

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

[EFAULT] **argp** points to an invalid address.

SEE ALSO

stty(1), exec(2), ioctl(2), sttyV6(7), termio(7), tty(7).

S

NAME

swapon - add swap space for interleaved paging/swapping

SYNOPSIS

```
#include <unistd.h>

int swapon(const char *path, ...
           /* [int min,
               int limit,
               int reserve,]
              int priority */ );
```

Remarks

The ANSI C "**, ...**" construct denotes a variable length argument list whose optional and required members are given in the associated comment (**/* */**).

DESCRIPTION

The **swapon()** system call makes a block device or a directory named *path* available to the system for paging and swapping.

priority indicates the order in which the swap space from the device or file system is used. Space is taken from the lower-priority systems first.

swapon() can be used only by users who have appropriate privileges.

If *path* names a block device file

swapon() makes it available to the system at the specified *priority* for allocation for paging and swapping.

In this form, **swapon()** takes only two arguments: the *path* to the block device file, and the *priority*.

The device associated with *path* can be a device already known to the system, defined at system configuration time, or it can be a previously unspecified device.

If the device was already defined at system configuration time and also has a start and/or size defined for that swap device, these values are used.

Otherwise, if a filesystem exists on the device, swap is added following the filesystem, or if no filesystem exists, the complete device is used for swap.

See the appropriate system administrator's manual for information on how the size of the swap area is calculated.

If *path* names a directory

swapon() makes the blocks on the file system rooted at *path* available for paging and swapping.

The *min*, *limit*, and *reserve* arguments are passed and used only if the *path* argument names a directory.

min indicates the number of file system blocks to take from the file system when **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.

S

ERRORS

If **swapon()** fails, **errno** is set to one of the following values.

[EACCES]	A component of the path prefix denies search permission.
[EALREADY]	The device or directory associated with *path* already has swap turned on.
[EBUSY]	The device associated with *path* is already in use.
[EEXIST]	The device associated with *path* was specified at system configuration time to add swap at a specified location, but that location is within an existing file system on the device.
[EFAULT]	The LIF header on the device associated with *path* contains inconsistent directory data.

[EIO] Unable to read the device associated with *path*.

[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.

[ENODEV] The device associated with *path* does not exist.

[ENOENT] The system-imposed limit on the number of swap file entries has been reached.

[ENOSPC] There is is not enough available space on the specified file system or device.

[ENOSYS] The device associated with *path* was specified at system configuration time to add swap following the file system, but no file system was found.

[ENOTBLK] The *path* argument is not a block special file or the root directory of a file system.

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

[ENXIO] The device associated with *path* could not be opened.

[EPERM] The effective user ID is not a user with appropriate privileges.

[EROFS] The device associated with *path* is read-only.

WARNINGS

No means is available to stop swapping to a device.

The system allocates 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 a file system is usually slower than swapping to a device.

Once file system blocks have been allocated for swap space, the file system can not be unmounted unless the system is rebooted.

AUTHOR

swapon() was developed by the University of California, Berkeley.

SEE ALSO

swapon(1M).

S

NAME

symlink - make symbolic link to a file

SYNOPSIS

```
#include <unistd.h>

int symlink(const char *path1, const char *path2);
```

DESCRIPTION

The **symlink()** function creates a symbolic link. Its name is the pathname pointed to by *path2*, which must be a pathname that does not name an existing file or symbolic link. The contents of the symbolic link are the string pointed to by *path1*.

RETURN VALUE

Upon successful completion, **symlink()** returns 0. Otherwise, it returns −1 and sets **errno** to indicate the error.

ERRORS

The **symlink()** function will fail if:

[EACCES]	Write permission is denied in the directory where the symbolic link is being created, or search permission is denied for a component of the path prefix of *path2*.
[EEXIST]	The *path2* argument names an existing file or symbolic link.
[EIO]	An I/O error occurs while reading from or writing to the file system.
[ELOOP]	Too many symbolic links were encountered in resolving *path2*.
[ENAMETOOLONG]	The length of the *path2* argument exceeds {**PATH_MAX**}, or a pathname component is longer than {**NAME_MAX**}.
[ENOENT]	A component of *path2* does not name an existing file or *path2* is an empty string.
[ENOSPC]	The directory in which the entry for the new symbolic link is being placed cannot be extended because no space is left on the file system containing the directory, or the new symbolic link cannot be created because no space is left on the file system which will contain the link, or the file system is out of file-allocation resources.
[ENOTDIR]	A component of the path prefix of *path2* is not a directory.
[EROFS]	The new symbolic link would reside on a read-only file system.

The **symlink()** function may fail if:

[ENAMETOOLONG]	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {**PATH_MAX**}.

S

APPLICATION USAGE

Like a hard link, a symbolic link allows a file to have multiple logical names. The presence of a hard link guarantees the existence of a file, even after the original name has been removed. A symbolic link provides no such assurance; in fact, the file named by the *path1* argument need not exist when the link is created. A symbolic link can cross file system boundaries.

Normal permission checks are made on each component of the symbolic link pathname during its resolution.

SEE ALSO

chown(2), link(2), lstat(2), open(2), readlink(2), <unistd.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

HP-UX EXTENSIONS

ERRORS

If **symlink()** fails, **errno** is set to one of the following values.

[EFAULT] *path1* or *path2* points outside the process's allocated address space. The reliable detection of this error is implementation-dependent.

[EIO] An I/O error occurred while making the directory entry for *path2*, allocating the inode for *path2*, or writing out the link contents of *path2*.

[EIO] An I/O error occurred while making the directory entry or allocating the inode.

AUTHOR

symlink() was developed by the University of California, Berkeley.

SEE ALSO

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

STANDARDS CONFORMANCE

symlink(): AES, SVID3

S

NAME

sync - update disk

SYNOPSIS

```
#include <unistd.h>

void sync(void);
```

DESCRIPTION

sync() causes all information in memory that should be on disk to be written out. This includes modified file system meta-data and delayed block I/O.

It should be used by commands and programs that examine a file system, such as **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.

AUTHOR

sync() was developed by HP and AT&T Bell Laboratories.

SEE ALSO

sync(1M), fdatasync(2), fsync(2).

STANDARDS CONFORMANCE

sync(): SVID2, SVID3, XPG2

NAME

sysconf() - get configurable system variables

SYNOPSIS

```
#include <unistd.h>

long sysconf(int name);

int CPU_IS_PA_RISC(long cpuvers);
```

DESCRIPTION

The **sysconf()** system call provides a way for 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 the value returned:

Variable	Value for *name*	Value Returned
AES_OS_VERSION	**_SC_AES_OS_VERSION**	Version number of OSF/AES OSC supported
ARG_MAX	**_SC_ARG_MAX**	Maximum total length of the arguments for **exec()** in bytes, including environment data (see *exec*(2))
ATEXIT_MAX	**_SC_ATEXIT_MAX**	Maximum number of functions that can be registered with **atexit()** (see *atexit*(2))
BC_BASE_MAX	**_SC_BC_BASE_MAX**	Maximum ibase (input number radix) and obase (output number radix) allowed by **bc** (see *bc*(1))
BC_DIM_MAX	**_SC_BC_DIM_MAX**	Maximum number of elements in an array permitted by **bc** (see *bc*(1))
BC_SCALE_MAX	**_SC_BC_SCALE_MAX**	Maximum scale factor (number of digits to the right of the decimal point) allowed by **bc** (see *bc*(1))
BC_STRING_MAX	**_SC_BC_STRING_MAX**	Maximum length of strings allowed by **bc** (see *bc*(1))
CHILD_MAX	**_SC_CHILD_MAX**	Maximum number of simultaneous processes per user ID (see *fork*(2))
CLK_TCK	**_SC_CLK_TCK**	Number of clock intervals per second for **times()** (see *times*(2))
CLOCKS_PER_SEC	**_SC_CLOCKS_PER_SEC**	Number of clock ticks per second for **clock()** (see *clock*(3C))
COLL_WEIGHTS_MAX	**_SC_COLL_WEIGHTS_MAX**	Maximum number of weights that can be assigned to an entry of the **LC_COLLATE order** keyword in a **localedef** input file (see *localedef*(1M))
CPU_CHIP_TYPE	**_SC_CPU_CHIP_TYPE**	Encoding which indicates type of CPU chip employed in system. Bits 21-26 identify the model, bits 27-31 the revision. See "Precision I/O Architecture Specification" for encodings.
CPU_KEYBITS1	**_SC_CPU_KEYBITS1**	Processor Extensions (see below)
CPU_VERSION	**_SC_CPU_VERSION**	Version of CPU architecture (see below)
EXPR_NEST_MAX	**_SC_EXPR_NEST_MAX**	Maximum parenthesis nesting level for **expr** expressions (see *expr*(1))

S

HW_32_64_CAPABLE	**_SC_HW_32_64_CAPABLE**	Returns which kernel is supported on the hardware. The value returned is an encoding which may be interpreted using the _SYSTEM_SUPPORTS_ILP32OS() and _SYSTEM_SUPPORTS_LP64OS() macros defined in unistd.h. Example:
		
		long ret = sysconf(_SC_HW_32_64_CAPABLE);
		
		if (_SYSTEM_SUPPORTS_ILP32OS(ret) != 0) {
		  /* system supports 32-bit OS */
		}
		
		if (_SYSTEM_SUPPORTS_LP64OS(ret) != 0) {
		  /* system supports 64-bit OS */
		}

IO_TYPE	**_SC_IO_TYPE**	Type of I/O drivers the kernel supports, currently, only the value **IO_TYPE_CDIO**
KERNEL_BITS	**_SC_KERNEL_BITS**	Returns the number of bits used by the kernel for pointer and long data types. Current values include 32 and 64.
LIBC_VERSION	**_SC_LIBC_VERSION**	The version of libc that is in use by the application that is requesting this information. See below for details.
LINE_MAX	**_SC_LINE_MAX**	Maximum number of bytes in an input line (including the newline) for POSIX.2 utilities
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
PAGE_SIZE	**_SC_PAGE_SIZE**	Kernel memory page size
PASS_MAX	**_SC_PASS_MAX**	Maximum number of significant bytes in a password
POSIX_FSYNC	**_SC_FSYNC**	Positive if the File Synchronization option is supported (see *fsync*(2))
POSIX_JOB_CONTROL	**_SC_JOB_CONTROL**	Positive if the system supports POSIX job control; –1 otherwise
POSIX_PRIORITY_ SCHEDULING	**_SC_PRIORITY_ SCHEDULING**	Positive if the system supports POSIX.4 priority scheduling; –1 otherwise
POSIX_REALTIME_SIGNALS	**_SC_REALTIME_SIGNALS**	Positive if the system supports POSIX.4 realtime signal extensions; –1 otherwise
POSIX_SAVED_IDS	**_SC_SAVED_IDS**	Positive if each process has a saved set-user-ID and a saved set-group-ID; –1 otherwise
POSIX_SYNCHRONIZED_IO	**_SC_SYNCHRONIZED_IO**	Positive if the Synchronized IO option is supported (see *open*(2))
POSIX_TIMERS	**_SC_TIMERS**	Positive if the system supports POSIX.4 clocks and timers; –1 otherwise

S

POSIX_VERSION	**_SC_VERSION**	Approval date of the POSIX.1 Standard (such as 199009 for POSIX.1-1990) to which the system conforms. This value indicates the year (first four digits) and month (next two digits) that the standard was approved by the IEEE Standards Board.
POSIX2_C_BIND	**_SC_2_C_BIND**	Equal to 1 if the POSIX.2 C Language Bindings Option is available through the **c89** utility; –1 otherwise
POSIX2_C_DEV	**_SC_2_C_DEV**	Equal to 1 if the POSIX.2 C Language Development Utilities Option is supported; –1 otherwise
POSIX2_C_VERSION	**_SC_2_C_VERSION**	Current version of the POSIX.2 C Language Binding Option supported (same format as **_POSIX_VERSION**); –1 otherwise.
POSIX2_FORT_DEV	**_SC_2_FORT_DEV**	Equal to 1 if the POSIX.2 FORTRAN Development Utilities Option is supported; –1 otherwise
POSIX2_FORT_RUN	**_SC_2_FORT_RUN**	Equal to 1 if the POSIX.2 Fortran Runtime Utilities Option is supported; –1 otherwise
POSIX2_LOCALEDEF	**_SC_2_LOCALEDEF**	Equal to 1 if locales can be created with the POSIX.2 *localedef* utility; –1 otherwise
POSIX2_SW_DEV	**_SC_2_SW_DEV**	Equal to 1 if the POSIX.2 Software Development Utilities Option is supported; –1 otherwise
POSIX2_UPE	**_SC_2_UPE**	Equal to 1 if the POSIX.2 User Portability Utilities Option is supported; –1 otherwise
POSIX2_VERSION	**_SC_2_VERSION**	Current version of POSIX.2 (same format as **_POSIX_VERSION**)
POSIX_THREADS	**_SC_THREADS**	Positive if the implementation supports POSIX threads; -1 otherwise.
POSIX_THREAD_ ATTR_STACKADDR	**_SC_THREAD_ ATTR_STACKADDR**	Positive if the implementation supports the POSIX Thread Stack Address Attribute option; -1 otherwise.
POSIX_THREAD_ ATTR_STACKSIZE	**_SC_THREAD_ ATTR_STACKSIZE**	Positive if the implementation supports the POSIX Thread Stack Size Attribute option; -1 otherwise.
POSIX_THREAD_ PRIORITY_SCHEDULING	**_SC_THREAD_ PRIORITY_SCHEDULING**	Positive if the implementation supports the POSIX Thread Priority Scheduling option; -l otherwise.
POSIX_THREAD_ PRIO_INHERIT	**_SC_THREAD_ PRIO_INHERIT**	Positive if the implementation supports the POSIX Thread Priority Inheritance option; -l otherwise.
POSIX_THREAD_ PRIO_PROTECT	**_SC_THREAD_ PRIO_PROTECT**	Positive if the implementation supports the POSIX Thread Priority Protection option; -l otherwise.
POSIX_THREAD_ PROCESS_SHARED	**_SC_THREAD_PROCESS_ SHARED**	Positive if the implementation supports the POSIX Thread Process-Shared Synchronization option; -l otherwise.
POSIX_THREAD_ SAFE_FUNCTIONS	**_SC_THREAD_SAFE_ FUNCTIONS**	Positive if the implementation supports the POSIX Thread Thread-Safe Functions option; -l otherwise.

PTHREAD_ DESTRUCTOR_ ITERATIONS	**_SC_THREAD_ DESTRUCTOR_ ITERATIONS**	The number of attempts made to destroy a pthread's thread-specific data values on thread exit.
PTHREAD_KEYS_MAX	**_SC_THREAD_ KEYS_MAX**	The number of pthread data keys per process.
PTHREAD_STACK_MIN	**_SC_THREAD_ STACK_MIN**	Minimum size in bytes of pthread stack storage.
PTHREAD_THREADS_MAX	**_SC_THREAD_ THREADS_MAX**	Maximum number of pthreads that can be created per process.
PROC_RSRC_MGR	**_SC_PROC_RSRC_MGR**	Equal to 1 if the optional HP Process Resource Management (PRM) software is installed and configured; 0 otherwise (see *prmconfig*(1))
RE_DUP_MAX	**_SC_RE_DUP_MAX**	Maximum number of repeated occurrences of a regular expression permitted when using the interval notation **\{***m***,***n***\}** (see *regcomp*(3C))
RTSIG_MAX	**_SC_RTSIG_MAX**	Maximum number of realtime signals reserved for application use.
SECURITY_CLASS	**_SC_SECURITY_CLASS**	**SEC_CLASS-NONE** (No DoD security level supported)
SIGQUEUE_MAX	**_SC_SIGQUEUE_MAX**	Maximum number of queued signals that a process may send and have pending at the receiver(s) at any time.
STREAM_MAX	**_SC_STREAM_MAX**	Maximum number of stdio streams that one process can have open at one time
TIMER_MAX	**_SC_TIMER_MAX**	Maximum number of POSIX.4 timers per process, if POSIX.4 timers are supported; –1 otherwise
TZNAME_MAX	**_SC_TZNAME_MAX**	Maximum number of bytes in a time zone name for the **TZ** environment variable
XOPEN_CRYPT	**_SC_XOPEN_CRYPT**	Equal to 1 if the X/Open Encryption Feature Group is supported; –1 otherwise
XOPEN_ENH_I18N	**_SC_XOPEN_ENH_I18N**	Equal to 1 if the X/Open Enhanced Internationalization Feature Group is supported; –1 otherwise
XOPEN_SHM	**_SC_XOPEN_SHM**	Equal to 1 if the X/Open Shared Memory Feature Group is supported; –1 otherwise
XOPEN_VERSION	**_SC_XOPEN_VERSION**	Issue number of *X/Open Portability Guide* supported
XBS5_ILP32_ OFF32	**_SC_XBS5_ILP32_ OFF32**	A flag which denotes whether **_CS_XBS5_ILP32_OFF32_CFLAGS**, **_CS_XBS5_ILP32_OFF32_LDFLAGS**, **_CS_XBS5_ILP32_OFF32_LIBS** and **_CS_XBS5_ILP32_OFF32_LINTFLAGS** are supported by *confstr*(3C). A return value of -1 indicates they are not supported.

S

XBS5_ILP32_ OFFBIG	**_SC_XBS5_ILP32_ OFFBIG**	A flag which denotes whether **_CS_XBS5_ILP32_OFFBIG_CFLAGS**, **_CS_XBS5_ILP32_OFFBIG_LDFLAGS**, **_CS_XBS5_ILP32_OFFBIG_LIBS** and **_CS_XBS5_ILP32_OFFBIG_LINTFLAGS** are supported by *confstr*(3C). A return value of -1 indicates they are not supported.
XBS5_LP64_ OFF64	**_SC_XBS5_LP64_ OFF64**	A flag which denotes whether **_CS_XBS5_LP64_OFF64_CFLAGS**, **_CS_XBS5_LP64_OFF64_LDFLAGS**, **_CS_XBS5_LP64_OFF64_LIBS** and **_CS_XBS5_LP64_OFF64_LINTFLAGS** are supported by *confstr*(3C). A return value of -1 indicates they are not supported.
XBS5_LPBIG_ OFFBIG	**_SC_XBS5_LPBIG_ OFFBIG**	A flag which denotes whether **_CS_XBS5_LPBIG_OFFBIG_CFLAGS**, **_CS_XBS5_LPBIG_OFFBIG_LDFLAGS**, **_CS_XBS5_LPBIG_OFFBIG_LIBS** and **_CS_XBS5_LPBIG_OFFBIG_LINTFLAGS** are supported by *confstr*(3C). A return value of -1 indicates they are not supported.

Some of 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>**.

The possible values of the **CPU_VERSION** variable returned by **sysconf(_SC_CPU_VERSION)** and their meanings are:

Value	Meaning
CPU_PA_RISC1_0	HP Precision Architecture RISC Version 1.0
CPU_PA_RISC1_1	HP Precision Architecture RISC Version 1.1

The **CPU_IS_PA_RISC()** function classifies *cpuvers*, a value of the **CPU_VERSION** variable, as to its processor family.

The availability of architecture specific instructions is indicated by the key bit data returned by **sysconf(_SC_CPU_KEYBITS1)**. Upon successful completion, the data returned will be the logical OR of the defined values for the features supported.

The possible values returned by **sysconf(_SC_CPU_KEYBITS1)** and their meanings are shown in the following table.

Return Value	Instruction Supported
HARITH	Halfword parallel add, subtract, and average
HSHIFT	Halfword parallel shift-and-add

S

The format of the value returned by **sysconf(_SC_LIBC_VERSION)** is as follows:

XXyyZZZZqN

where

XX HP-UX major release number

yy HP-UX minor release number

ZZZZ Library specific number

q 0=32PA 1=64PA 2=32EM 3=64EM 4-9=Reserved

N 0 = archive library

1-9 = System V version of shared library

RETURN VALUE

Upon successful completion, **sysconf()** returns the value of the named variable. 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**, but does not change **errno**.

CPU_IS_PA_RISC() returns positive nonzero if *cpuvers* is an HP PA-RISC processor; zero if not.

ERRORS

If **sysconf()** fails, the value of **errno** (see *errno*(2)) is set to:

[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 ticks;
  ...
ticks = sysconf(_SC_CLK_TCK);
```

The following example determines if the current processor is an HP PA-RISC machine:

```
#include <unistd.h>
if (CPU_IS_PA_RISC(sysconf(_SC_CPU_VERSION)))
  ...
```

WARNINGS

CPU_IS_PA_RISC() is implemented as a macro.

Normally, the values returned from **sysconf()** do not change during the lifetime of the calling process. However, the value of the symbolic constant **_POSIX_VERSION** and thus the value of **sysconf(_SC_VERSION)** can vary under certain circumstances. If either of the feature test macros **_POSIX1_1988** or **_XPG3** is defined by the programmer prior to including **<unistd.h>**, the value of **_POSIX_VERSION** is defined as **198808**, in conformance with POSIX.1-1988, FIPS 151-1, and XPG3. Otherwise, the value of **_POSIX_VERSION** is defined as **199009**, in conformance with POSIX.1-1990.

Similarly, the value of the symbolic constant **_XOPEN_VERSION** and thus the value of **sysconf(_SC_XOPEN_VERSION)** can vary under certain circumstances. If the feature test macro **_XPG3** is defined by the programmer prior to including **<unistd.h>**, the value of **_XOPEN_VERSION** is defined as **3**, in conformance with XPG3. Otherwise, the value of **_XOPEN_VERSION** is defined as **4**, in conformance with XPG4.

See *stdsyms*(5) for more information about these feature test macros.

Any application that has a dependency on libdld.sl is a potential user of both archived and shared libc. Applications that comprise both archived and shared components where **sysconf(_SC_LIBC_VERSION)** may be invoked from both the archived and shared components may get inconsistent return values from **sysconf()**.

AUTHOR

sysconf() was developed by HP and POSIX.

CPU_IS_PA_RISC() was developed by HP.

SEE ALSO

getconf(1), atexit(2), exec(2), fork(2), getrlimit(2), pathconf(2), times(2), clock(3C), regcomp(3C), limits(5), stdsyms(5), unistd(5), x_open(5).

HP Process Resource Manager: prmconfig(1) in *HP Process Resource Manager User's Guide*.

STANDARDS CONFORMANCE

sysconf(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1, POSIX.2, POSIX.4

S

NAME

sysfs - get file system type information

SYNOPSIS

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

int sysfs(int opcode, const char *fsname);

int sysfs(int opcode, int fs_index, char *buf );

int sysfs(int opcode);
```

DESCRIPTION

sysfs is used to return information about the file system types configured in the system. The number arguments accepted by **sysfs** varies and depends on the *opcode*.

The current recognized *opcodes* and their functions are:

GETFSIND Translate *fsname*, a null-terminated file-system type identifier, into a file-system type index.

GETFSTYP Translate *fs_index*, a file-system type index, into a null-terminated file-system type identifier and write it into the buffer pointed to by *buf;* this buffer must be at least of size **FSTYPSZ** as defined in **<sys/fstyp.h>.** If there is no file-system type configured at fs_index, a null string is returned for the file-system type identifier.

GETNFSTYP Return one more than the largest file system type configured. This is not the number of file system types configured, because the type numbers may not be contiguous. See the example below.

RETURN VALUE

Upon successful completion, **sysfs()** returns the file-system type index if the *opcode* is **GETFSIND,** a value of 0 if the *opcode* is **GETFSTYP,** or the number of file system types configured if the *opcode* is **GETNFSTYP.** Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

ERRORS

sysfs fails if one or more of the following are true and sets **errno** to the value indicated:

EINVAL *fsname* points to an invalid file-system identifier; *fs_index* is negative or greater than the largest file-system type index; *opcode* is invalid.

EFAULT *buf* or *fsname* points to an invalid user address.

EXAMPLE

List the filesystem types configured in the system.

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

int max_type, error, i;
char name[FSTYPSZ];
max_type = sysfs(GETNFSTYP);
for (i = 0; i < max_type; i++) {
        error = sysfs(GETFSTYP, i, name);
        if (error == 0)
                my_print(name);
}
```

NAME

time - get time

SYNOPSIS

```
#include <time.h>

time_t time(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.

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.

ERRORS

[EFAULT] **time()** fails if *tloc* points to an illegal address. The reliable detection of this error is implementation dependent.

SEE ALSO

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

STANDARDS CONFORMANCE

time(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

t

NAME

timer_create(), timer_delete(), timer_settime(), timer_gettime(), timer_getoverrun() - timer operations

SYNOPSIS

```
#include <time.h>

int timer_create(
     clockid_t clock_id,
     struct sigevent *evp,
     timer_t *timerid
);

int timer_delete(
     timer_t timerid
);

int timer_settime(
     timer_t timerid,
     int flags,
     const struct itimerspec *value,
     struct itimerspec *ovalue
);

int timer_gettime(
     timer_t timerid,
     struct itimerspec *value
);

int timer_getoverrun(
     timer_t timerid
);
```

DESCRIPTION

timer_create()

The **timer_create()** function creates a per-process timer using the specified clock, **clock_id**, as the timing base. The **timer_create()** function returns, in the location referenced by **timerid**, a timer ID of type **timer_t** used to identify the timer in timer requests. This timer ID will be unique within the calling process until the timer is deleted. The particular clock, **clock_id**, is defined in **<time.h>**. The timer whose ID is returned will be in a disarmed state upon return from **timer_create()**.

The **evp** argument, if non-NULL, points to a *sigevent* structure. If the *sigev_notify* member of **evp** is SIGEV_SIGNAL, then the structure should also specify the signal number to be sent to the process on timer expiration. The signal to be sent is specified in the *sigev_signo* field of **evp**. If the *sigev_notify* member of *evp* is SIGEV_NONE, no notification is sent. If **evp** is NULL, then a default signal is sent to the process. The defaults for the clocks **CLOCK_REALTIME**, **CLOCK_VIRTUAL**, and **CLOCK_PROFILE** are **SIGALRM**, **SIGVTALRM**, and **SIGPROF**.

Per-process timers are not inherited by a child process across a **fork()** and are disarmed and deleted by an **exec()**.

timer_delete()

The **timer_delete()** function deletes the specified timer, **timerid**, previously created by the **timer_create()** function. If the timer is armed when **timer_delete()** is called, the behavior is as if the timer is automatically disarmed before removal. Any pending notifications from the timer remain.

timer_settime()

The **timer_settime()** function sets the time until the next expiration of the timer specified by **timerid** from the **it_value** member of the **value** argument and arms the timer if the **it_value** member of **value** is non-zero. If the specified timer was already armed when **timer_settime()** is called, this call resets the time until next expiration to the **value** specified. If the **it_value** member of **value** is zero, the timer is disarmed. Any pending notifications from the timer remain.

If the flag **TIMER_ABSTIME** is not set in the argument **flags**, **timer_settime()** behaves as if the time until next expiration is set equal to the interval specified by the **it_value** member of **value**. That is, the timer will expire in **it_value** nanoseconds from when the call is made.

If the flag **TIMER_ABSTIME** is set in the argument **flags**, **timer_settime()** behaves as if the time until next expiration is set equal to the difference between the absolute time specified by the **it_value** member of **value** and the current value of the clock associated with **timerid**. That is, the timer will expire when the clock reaches the value specified by the **it_value** member of **value**. If the specified time has already passed, the function will succeed and the expiration notification is made.

The reload value of the timer is set to the value specified by the **it_interval** member of **value**. When a timer is armed with a non-zero **it_interval**, a periodic (or repetitive) timer is specified.

Time values that are between two consecutive non-negative integer multiples of the resolution of the specified timer are rounded up to the larger multiple of the resolution. A quantization error will not cause the timer to expire earlier than the rounded-up time value.

If the argument **ovalue** is not NULL, the function **timer_settime()** stores, in the location referenced by **ovalue**, a value representing the previous amount of time before the timer would have expired or zero if the timer was disarmed, together with the previous timer reload value. The members of **ovalue** are subject to the resolution of the timer, and are the same values that would be returned by a **timer_gettime()** call at that point in time.

timer_gettime()

The **timer_gettime()** function stores the amount of time until the specified timer, **timerid**, expires and the timer's reload value into the space pointed to by the **value** argument. The **it_value** member of this structure will contain the amount of time before the timer expires, or zero if the timer is disarmed. This value is returned as the interval until timer expiration, even if the timer was armed with absolute time. The **it_interval** member of **value** will contain the reload value last set by **timer_settime()**.

timer_getoverrun()

Only a single signal is delivered to the process for a given timer at any point in time. When a timer for which a signal is still pending expires, no signal is delivered, and a timer overrun has occurred. When a timer expiration signal is delivered to a process, the **timer_getoverrun()** function returns the timer expiration count for the specified timer. The overrun count returned contains the number of extra timer expirations which occurred between the time the signal was generated and when it was delivered, up to but not including an implementation defined maximum of **DELAYTIMER_MAX**. If the number of such extra expirations is greater than or equal to **DELAYTIMER_MAX**, then the overrun count is set to **DELAYTIMER_MAX**. The value returned by **timer_getoverrun()** applies to the most recent expiration signal delivery for the timer. If no expiration signal has been delivered for the timer, the meaning of the overrun count returned is undefined.

RETURN VALUE

Upon successful completion, **timer_create()** returns zero and updates the location referenced by **timerid** to a **timer_t** which can be passed to the per-process timer calls. Otherwise, **timer_create()** returns −1 and sets **errno** to indicate the error. The value of **timerid** is undefined if an error occurs.

Upon successful completion, **timer_delete()** returns zero. Otherwise, **timer_delete()** returns −1 and sets **errno** to indicate the error.

Upon successful completion, **timer_settime()** returns zero and updates the location referenced by **ovalue**, if **ovalue** is non-NULL.

Upon successful completion, **timer_gettime()** returns zero and updates the location referenced by **value**, if **ovalue** is non-NULL. Otherwise, **timer_gettime()** returns −1 and sets **errno** to indicate the error.

Upon successful completion, **timer_getoverrun()** returns the timer expiration overrun count as explained above. Otherwise, **timer_getoverrun()** returns −1 and sets **errno** to indicate the error.

ERRORS

If any of the following conditions occur, the **timer_create()** function returns −1 and sets **errno** (see *errno*(2)) to the corresponding value:

- [EAGAIN] The system lacks sufficient signal queuing resources to honor the request.
- [EAGAIN] The calling process has already created all of the timers it is allowed by this implementation.

t

[EINVAL] The specified clock ID is not defined.

[EFAULT] The **timerid** or **evp** argument points to an invalid address.

[ENOSYS] The function **timer_create()** is not supported by this implementation.

If any of the following conditions occur, the **timer_delete()** function returns −1 and sets **errno** to the corresponding value:

[EINVAL] The timer ID specified by **timerid** is not a valid timer ID.

[ENOSYS] The function **timer_delete()** is not supported by this implementation.

If any of the following conditions occur, the **timer_settime()**, **timer_gettime()**, and **timer_getoverrun()** functions return −1 and set **errno** to the corresponding value:

[EINVAL] The **timerid** argument does not correspond to an ID returned by **timer_create()**, but not yet deleted by **timer_delete()**.

[EINVAL] The *value* structure passed to **timer_settime()** specified a nanosecond value less than zero or greater than or equal to 1000 million.

[EFAULT] The **value** or **ovalue** argument points to an invalid address.

[ENOSYS] The **timer_settime()**, **timer_gettime()**, and **timer_getoverrun()** functions are not supported by this implementation.

EXAMPLES

Create a timer, set it to go off in one minute, and deliver a **SIGUSR1** signal:

```
#include <signal.h>
#include <time.h>

timer_t timerid;
struct itimerspec one_minute = { {60, 0}, {0, 0} } ;

void handler()
{
     int overrun = timer_getoverrun(timerid);

     if (overrun == -1) {
          perror("handler: timer_getoverrun()");
          exit(1);
     }
     (void)printf("Timer expired, overrun count was %d,
               overrun);
}

int main()
{
     struct sigaction sigact;
     struct sigevent sigev;

     sigact.sa_handler = handler;
     sigemptyset(sigact.sa_mask);
     sigact.sa_flags = 0;

     if (sigaction(SIGUSR1, &sigact, (struct sigaction *)NULL)
          == -1) {
          perror("sigaction");
          exit(1);
     }

     sigev.sigev_notify = SIGEV_SIGNAL;
     sigev.sigev_signo = SIGUSR1;

     if (timer_create(CLOCK_REALTIME, &sigev, &timerid)
          == -1) {
          perror("timer_create");
          exit(1);
     }
```

t

```
        if (timer_settime(timerid, 0, &one_minute, (struct itimerspec
             == -1) {
             perror("timer_create");
             exit(1);
        }
        pause();
        if (timer_delete(timerid) == -1) {
             perror("timer_delete");
             exit(1);
        }
        return 0;
   }
```

AUTHOR

timer_create(), **timer_delete()**, **timer_settime()**, **timer_gettime()**, and **timer_getoverrun()** were derived from the proposed IEEE POSIX P1003.4 standard, draft 14.

SEE ALSO

clocks(2), getitimer(2).

STANDARDS CONFORMANCE

timer_create(): POSIX.4

timer_delete(): POSIX.4

timer_getoverrun(): POSIX.4

timer_gettime(): POSIX.4

timer_settime(): POSIX.4

t

NAME

times - get process and child process times

SYNOPSIS

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

clock_t times(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()** function (see *sysconf*(2)).

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.

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 (such as system start-up time). This point does not change from one invocation of **times()** to another. If **times()** fails, –1 is returned and **errno** is set to indicate the error.

Remarks

times() has a granularity of one tick. Processes which run less than one tick may not register any value.

ERRORS

[EFAULT] **times()** fails if *buffer* points to an illegal address. The reliable detection of this error is implementation dependent.

SEE ALSO

time(1), gettimeofday(2), exec(2), fork(2), sysconf(2), time(2), wait(2).

WARNINGS

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(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

ftruncate, truncate - truncate a file to a specified length

SYNOPSIS

```
#include <unistd.h>

int ftruncate(int fildes, off_t length);

int truncate(const char *path, off_t length);
```

DESCRIPTION

The **ftruncate()** function causes the regular file referenced by *fildes* to have a size of *length* bytes.

The **truncate()** function causes the regular file named by *path* to have a size of *length* bytes.

The effect of **ftruncate()** and **truncate()** on other types of files is unspecified. If the file previously was larger than *length*, the extra data is lost. If it was previously shorter than *length*, bytes between the old and new lengths are read as zeroes. With **ftruncate()**, the file must be open for writing; for **truncate()**, the process must have write permission for the file.

If the request would cause the file size to exceed the soft file size limit for the process, the request will fail and the implementation will generate the **SIGXFSZ** signal for the process.

These functions do not modify the file offset for any open file descriptions associated with the file. On successful completion, if the file size is changed, these functions will mark for update the *st_ctime* and *st_mtime* fields of the file, and if the file is a regular file, the **S_ISUID** and **S_ISGID** bits of the file mode may be cleared.

RETURN VALUE

Upon successful completion, **ftruncate()** and **truncate()** returns 0. Otherwise a –1 is returned, and **errno** is set to indicate the error.

ERRORS

The **ftruncate()** and **truncate()** functions will fail if:

[EINTR]	A signal was caught during execution.
[EINVAL]	The *length* argument was less than 0.
[EFBIG] or [EINVAL]	The *length* argument was greater than the maximum file size.
[EIO]	An I/O error occurred while reading from or writing to a file system.

The **ftruncate()** function will fail if:

[EBADF] or [EINVAL]	The *fildes* argument is not a file descriptor open for writing.
[EINVAL]	The *fildes* argument references a file that was opened without write permission.

The **truncate()** function will fail if:

[EACCES]	A component of the *path* prefix denies search permission, or write permission is denied on the file.
[EISDIR]	The named file is a directory.
[ELOOP]	Too many symbolic links were encountered in resolving *path*.
[ENAMETOOLONG]	The length of the specified pathname exceeds **PATH_MAX** bytes, or the length of a component of the pathname exceeds **NAME_MAX** bytes.
[ENOENT]	A component of **path** does not name an existing file or path is an empty string.
[ENOTDIR]	A component of the *path* prefix of path is not a directory.
[EROFS]	The named file resides on a read-only file system.

The **truncate()** function may fail if:

[ENAMETOOLONG]	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds **{PATH_MAX}**.

t

SEE ALSO

open(2), <unistd.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

t

HP-UX EXTENSIONS

SYNOPSIS

```
int truncate(const char *path, size_t length);
int ftruncate(int fildes, size_t length);
```

ERRORS

If truncate() fails, **errno** is set to one of the following values:

[EACCES] MAC access is denied on the file.

[EDQUOT] The user's disk quota block limit has been reached for this file system.

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

[EINVAL] *length* was greater than the maximum file size.

[ETXTBSY] The file is a pure procedure (shared text) file that is being executed.

If **ftruncate()** fails, **errno** is set to one of the following values:

[EDQUOT] The user's disk quota block limit has been reached for this file system.

AUTHOR

truncate() was developed by the University of California, Berkeley.

SEE ALSO

ftruncate64(2), open(2), truncate64(2).

STANDARDS CONFORMANCE

truncate(): AES ftruncate(): AES, SVID3

t

NAME

ttrace - tracing facility for multithreaded processes

SYNOPSIS

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

int ttrace (ttreq_t request,  pid_t pid, lwpid_t lwpid,
     uint64_t addr, uint64_t data, uint64_t addr2);
```

Remarks

While the posix API is defined and will not change, the present underlying system calls are not guaranteed to be compatible with future versions.

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

The **ttrace()** system call provides a means by which a process can control the execution of another process. Its primary use is for the implementation of breakpoint and event driven debugging; see *adb*(1) and *dde*(1). **ttrace()** is designed to function for both single and multithreaded traced processes. The traced process behaves normally until one of its threads encounters a signal (see *signal*(2) for the list), or an *event* (these are discussed in detail in the **EVENTS** section below) at which time the thread enters a stopped state and the tracing process is notified via **ttrace_wait()**.

The *request* argument determines the action to be taken by **ttrace()** and is one of the following:

TT_PROC_SETTRC

This request must be issued by a child process if it is to be traced by its parent.

For this request, the *pid*, *lwpid*, *addr*, and *addr2* arguments must be set to 0 (zero) and *data* must be set to **TT_VERSION**. Peculiar results occur if the parent does not expect to trace the child.

Note that it is critical for future backward compatibility that the **TT_VERSION** macro itself be used and not its value.

All other requests are to be used only by the tracing process. They are divided in two groups: requests that target a process and requests that target a specific thread within the process. For all process-wide requests (those prefixed by **TT_PROC_**), *pid* is the process ID of the traced process and *lwpid* must be set to zero.

The process-wide requests are:

TT_PROC_ATTACH

This request allows the calling process to trace the process identified by *pid*. If the executable image of process *pid* is NFS mounted, it is necessary that the mount point be a hard, non-interruptible mount point, for the request to complete successfully. The 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 the process *pid* unless the effective user ID of the tracing process is super-user.

t

When this call returns, the target process (all its threads) is stopped.

For this request, the *lwpid*, *addr* and *addr2* arguments must be set to zero and *data* must be **TT_VERSION** (see **TT_PROC_SETTRC** above).

TT_PROC_DETACH

This request detaches the traced process and allows it to continue executing. It behaves identically to **TT_PROC_CONTINUE** except that the process is no longer being traced after the call returns.

For this request, the *lwpid*, *addr*, *data* and *addr2* arguments must be set to zero.

TT_PROC_RDTEXT
TT_PROC_RDDATA

These requests allow reading from the target process text (**TT_PROC_RDTEXT**) or data space (**TT_PROC_RDDATA**).

The *addr* argument specifies the offset to be read from. The *data* argument specifies the number of bytes to read and the *addr2* argument specifies where to store that data in the tracing process.

The *lwpid* argument must be set to zero.

TT_PROC_WRTEXT
TT_PROC_WRDATA

These requests allow writing into the target process text (**TT_PROC_WRTEXT**) and data spaces (**TT_PROC_WRDATA**).

The *addr* argument specifies the offset to be written to. The *data* argument specifies the number of bytes to write.a The *addr2* argument specifies where to get the data in the tracing process.

The *lwpid* argument must be set to zero.

TT_PROC_STOP

This request causes the traced process (all its threads) to stop. If a thread was already stopped by the debugger prior to this call, its state is not modified.

The *lwpid*, *addr*, *data* and *addr2* arguments must be set to zero.

TT_PROC_CONTINUE

This request causes the entire traced process to resume execution. All threads that had been stopped directly (request) or indirectly (event) by the debugger are resumed with all their pending signals intact.

The *data*, *addr* and *addr2* arguments must be set to zero.

TT_PROC_GET_PATHNAME

This request is used by the calling process to access the path name of the executable file provided as a *path* or *file* argument to **exec()**. The request reads *data* bytes of data of the pathname string from the traced process' context into the data buffer in user space pointed to by *addr*.

In the typical case, *data* is equal to the value of the *ttexec_data_t.tts_len* member of the **ttstate_t** structure returned via the **TT_LWP_GET_STATE** or other **ttrace** requests returning a Lightweight Process (LWP or lwp) state. The length of the path does not include a terminating null character. The data is available during the entire life of the process.

The *lwpid* and *addr2* arguments must be set to zero.

TT_PROC_GET_EVENT_MASK

This request returns the process-wide event flags and signal mask values.

The *data* argument specifies the number of bytes to be read from the context of the traced process into the **ttevent_t** data structure in user space pointed to by *addr*.

The *lwpid* and *addr2* arguments must be set to zero.

The **ttevent_t** data structure is as follows:

```
typedef struct {
    sigset_t    tte_signals;
    ttevents_t  tte_events;
    tteopt_t    tte_opts;
} ttevent_t;
```

The options provided in *tte_opts* control the behavior of child processes produced by **fork()** and are as follows:

```
TTEO_NONE         = 0x0
TTEO_NOSTRCCHLD   = 0x1
TTEO_PROC_INHERIT = 0x2
TTEO_LWP_INHERIT  = 0x4
TTEO_NORM_SIGTRAP = 0x8
```

If **TTEO_NOSTRCCHLD** is set, the child process resulting from a **fork()** will not be traced. This makes it possible for a debugger to debug another debugger. The **TTEO_PROC_INHERIT** and **TTEO_LWP_INHERIT** options allow events to be inherited by child processes and/or threads. Refer to the **EVENTS** section below.

If **TTEO_NORM_SIGTRAP** is set, the SIGTRAP signal behaves normally. That is, it is getting delivered (the default behavior is to drop these signals).

t

TT_PROC_SET_EVENT_MASK

This request allows the tracing process to establish events and signals the traced process will respond to. Refer to the **EVENTS** section for a description of these events.

The *addr* argument is a pointer to a **ttevent_t** structure to be copied into the target process. The *data* argument specifies the number of bytes to be transferred.

The *lwpid* and *addr2* arguments must be set to zero.

TT_PROC_GET_FIRST_LWP_STATE

This request returns the **ttstate_t** structure associated with the first thread on the stopped list. It resets the list pointer to the first entry in the list. The **TT_PROC_GET_NEXT_LWP_STATE** request (see below) provides the means to examine the state of other stopped threads.

The *data* argument specifies the number bytes to be read from the context of the traced process into the **ttstate_t** data structure in user space pointed to by *addr*. The *lwpid* and *addr2* arguments must be zero.

The **ttstate_t** structure provides the debugger with the means to query the system for the state of a thread. It is established when a thread enters the debugger stopped state and, except for the **TTS_WAITEDFOR** bit, is invariant until the thread is resumed. Its layout is as follows:

```
typedef struct {
    pid_t        tts_pid;
    lwpid_t      tts_lwpid;
    uint64_t     tts_user_tid;
    ttevents_t   tts_event;
    ttsf_t       tts_flags;
    int          tts_scno;
    int          tts_scnargs;
    uint64_t     tts_scarg[SCALL_MAXARGS];
    union {
        ttexec_data_t      tts_exec;
        ttfork_data_t      tts_fork;
        ttsignal_data_t    tts_signal;
        ttthread_data_t    tts_thread;
        ttsyscall_data_t   tts_syscall;
        ttexit_data_t      tts_exit;
        char               tts_fill[128];
    } tts_u;
} ttstate_t;
```

tts_pid is the process ID.

tts_lwpid is the lwpid of the stopped thread.

t

tts_user_tid is the thread's user ID.

tts_event is the event that caused the stop (**TTEVT_NONE** if the thread stopped because of a **ttrace** command).

The **tts_flags** provide information about the state of the thread before it was stopped. The information specifies whether or not the thread has been waited for by **ttrace_wait()**, whether or not it is processing a system call, whether it is a 32-bit or a 64-bit process and whether the thread is in the **exit()** system call. The values are as follows:

```
TTS_WASSUSPENDED = 0x0001
TTS_WASSLEEPING  = 0x0002
TTS_WASRUNNING   = 0x0004
TTS_WAITEDFOR    = 0x0008
TTS_INSYSCALL    = 0x0010
TTS_IS32BIT      = 0x0020
TTS_ATEXIT       = 0x0040
```

The following three arguments provide information regarding the system call being executed when the thread was stopped. This information is valid only if the

TTS_INSYSCALL bit is set in **tts_flags**.

tts_scno is the system call number.

tts_scnargs is the number of arguments of the system call.

tts_scarg is the argument list of the system call.

The data associated with a **TTEVT_EXEC** event is as follows:

```
typedef struct {
    int     tts_pathlen;
} ttexec_data_t;
```

tts_pathlen is the length of the pathname of the **exec()** system call.

The data associated with a **TTEVT_FORK** or **TTEVT_VFORK** event is as follows:

```
typedef struct {
    pid_t   tts_fpid;
    lwpid_t tts_flwpid;
    int     tts_isparent;
} ttfork_data_t;
```

tts_fpid is the process ID of the other side of the fork.

tts_flwpid is the thread ID of the other side of the fork.

tts_isparent is zero for the child event and one for the parent.

The data associated with a **TTEVT_SIGNAL** event is as follows:

```
typedef struct {
    int             tts_signo;
    ttsigf_t        tts_sigflags;
    uint64_t        tts_sigaction;
    siginfo_t       tts_siginfo;
} ttsignal_data_t;
```

tts_signal is the signal number.

tts_sigflags is **TTSF_USERSIGINFO** if a **siginfo** was delivered with the signal, 0 otherwise.

tts_sigaction is the disposition of the signal.

tts_siginfo is the **siginfo**, if applicable.

The data associated with a **TTEVT_LWP_CREATE**, **TTEVT_LWP_TERMINATE** or **TTEVT_LWP_ABORT_SYSCALL** event is as follows:

```
typedef struct {
    lwpid_t         tts_target_lwpid;
} ttthread_data_t;
```

tts_target_lwpid is the *lwpid* of the targeted lwp.

The data associated with a **TTEVT_SYSCALL** event is as follows:

```
typedef struct {
    int64_t     tts_rval[2];
    int         tts_errno;
} ttsyscall_data_t;
```

The **tts_rval** fields are the return value(s) of the system call.

tts_errno is the error status if the system call failed.

The data associated with a **TTEVT_LWP_EXIT** event is as follows:

```
typedef struct {
    int         tts_exitcode;
} ttexit_data_t;
```

tts_exitcode is the exit code of the process.

t

TT_PROC_GET_NEXT_LWP_STATE

This request is identical to **TT_PROC_GET_FIRST_LWP_STATE** except that it returns the state for the next thread on the stopped list. As events cause threads to stop, they are added to this list. This provides a way for the tracing process to examine the state of all the stopped threads in the target process. Both these requests return either a 1 (one) if valid data is returned or 0 (zero) otherwise. Valid data is returned if the status is that there was a stopped thread for which to return.

TT_PROC_GET_MPROTECT

This request allows the debugger to obtain protection information for a page in the address space of the code being debugged. The *addr* argument specifies the address for which the protection is to be obtained. The *addr2* argument specifies the address of an integer in which the protection data will be copied.

For this request, the *lwpid* and *data* arguments must be set to zero.

TT_PROC_SET_MPROTECT

This requests allows the debugger to modify the protection of the address space of the code being debugged. The *addr* argument specifies the start address. The *data* argument specifies the extent (in bytes) of the space to be modified. The *addr2* argument contains the new protection. Note that protection changes affect whole pages (see *mprotect*(2) for more information).

For this request, the *lwpid* argument must be set to zero.

TT_PROC_SET_SCBM

This request allows the debugger to pass a bitmap to the kernel indicating which system calls should cause a debugger stop.

The *addr* argument must be set to **TTSCBM_SELECT** or **TTSCBM_UNSELECT** to indicate whether the bitmap represents a positive (meaning that the calls in the bitmap will result in a stop) or a negative (meaning that all calls except those in the bit map will result in a stop) list.

The *data* argument is the size of the bitmap, in bytes. A size of zero indicates that the current bitmap, if any, should be cleared.

The *addr2* argument is the user address where the bitmap is located. If *data* is zero, this value must be zero too.

The *lwpid* argument must be zero.

TT_PROC_EXIT

This request causes the traced process to terminate. It has the same consequence as **exit()** being invoked by one of the process threads. The *lwpid*, *addr*, *data* and *addr2* arguments must be zero.

TT_PROC_CORE

This request causes the traced process to generate a core file in the process's current working directory. The core file is named *core.pid* where *pid* is the process ID of the target process. The process's state is left unchanged. The *lwpid*, *addr*, *data* and *addr2* arguments must be zero.

All other requests are targeted to a specific thread in the traced process. Also, all other requests require both the *pid* of the traced process and an *lwpid* specifying a valid thread in the process being traced. These requests are prefixed by **TT_LWP_** and are as follows:

TT_LWP_STOP

This request causes the thread identified by *lwpid* to stop executing. If the thread is already stopped **by the debugger**, or **by an event**, an error is returned.

The *addr*, *data* and *addr2* arguments must be zero.

TT_LWP_CONTINUE

This request causes the thread identified by *lwpid* to resume execution or, rather, to return to the state it was in prior to being stopped **by the debugger**. If the thread had not previously been stopped by the debugger, an error is returned.

If *addr* is not **TT_NOPC**, that value is loaded in the program counter before execution is resumed. Unexpected behavior will result if this value is not within the same function since only the PC, not the context, is being modified.

If *data* is non-zero, it is expected to be a valid signal number and the thread will continue as if it had received this signal.

The *addr2* argument must be zero.

TT_LWP_SINGLE
This request causes the stopped thread identified by *lwpid* to resume execution for one machine instruction. It 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 the **TT_LWP_CONTINUE** request.

TT_LWP_GET_EVENT_MASK
This request is the same as **TT_PROC_GET_EVENT_MASK** except for the thread identified by *lwpid*.

TT_LWP_SET_EVENT_MASK
This request is the same as **TT_PROC_SET_EVENT_MASK** except for the thread identified by *lwpid*.

TT_LWP_GET_STATE
This calls returns the state of the thread identified by *lwpid*. If the thread was not previously stopped by the debugger or waiting to be continued after an event, an error is returned.

SECURITY FEATURES

For security reasons, **ttrace()** inhibits the set-user-ID facility on subsequent **exec()** calls.

EVENTS

As noted earlier, a tracing process can set event flags in the context of a traced process, or its individual threads, to cause the threads to respond to specific events during their execution. When an event flag is set in the context of the process, all threads in the process respond to the event. When set in the context of a thread, only the specific thread will respond to the event.

IMPORTANT: If an event is requested by the process, the event mask of the thread is not examined. For the event mask of the thread to be significant, the process event must be be unset. Similarly, if an event option is enabled in the process, the option for the thread is not considered. Event masks may be inherited across **fork()** using the **tte_opts** options in the **ttevent_t** structure. If **TTEO_PROC_INHERIT** is set, the child process inherits the event mask of its parent. If **TTEO_LWP_INHERIT** is set, the lwp inherits the event mask of the lwp that invoked **fork()**. If the latter is set, the lwp created by **lwp_create()** also inherits the event mask of the creating thread.

These events are:

TTEVT_SIGNAL This event flag indicates that the traced thread needs to examine signal mask bits when processing signals. This means that, by default, threads stop when receiving a signal. If the signal being processed has its mask bit set, signal processing continues as though the process were not traced: the traced thread is not stopped, and the tracing process is not notified of the signal. On the other hand, if the signal mask bit is not set for the signal being processed, the traced thread is stopped and the tracing process is notified via **ttrace_wait()**.

Note that the SIGKILL signal can never be unmasked. It behaves as though its mask bit were always set. This means that a SIGKILL signal cannot be used to stop a traced thread. The SIGTRAP signal is also special in that it is used to stop traced threads when they respond to a trap, such as a breakpoint or a single step. Consequently, masking SIGTRAP, even though allowed, will result in unexpected behavior in these conditions.

TTEVT_FORK This event flag indicates that the traced thread needs to take special action when it invokes **fork()**. When set, both the parent thread and the initial thread in the child process stop (after the child process is marked as a traced process and adopts its parent's debugger). Both threads log the fact that they stopped in response to a **TTEVT_FORK** event. The parent thread provides the pid of the child process in the appropriate portion of the **ttstate_t** structure. The initial thread of the child process provides the pid of the parent in the same location. See the **ttstate_t** structure description for further details.

t

TTEVT_VFORK This event flag indicates that the traced thread needs to take special action when it invokes **vfork()**. The behavior is identical to that of TTEVT_FORK but it is important to note that the caveats with respect to **vfork()**, continue to apply here. In particular, it needs to be remembered that when the child process stops, its parent is asleep, and that the child borrows the parent's address space until a call to **exec()** or an exit (either by a call to **exit()** or abnormally) takes place. Continuing the parent process before the above steps take place results in an error.

TTEVT_EXEC This event flag indicates that a traced thread needs to notify the debugger upon completion of loading the new executable file, in the **exec()** system call. The length of the pathname string (not including a null terminating character) is returned in the **ttstate_t** structure and the path may subsequently be obtained using the **TT_PROC_GET_PATHNAME** request.

TTEVT_SYSCALL_RETURN
This event flag indicates that the traced process will notify the debugger upon return of all system calls. The traced process will also provide the following information: the system call number, its number of arguments and all its arguments, its return value and its error return in the **ttstate_t** structure. If the system call is a **fork()**, **vfork()** or **exec()** and if, respectively, the **TTEVT_FORK**, **TTEVT_VFORK** or **TTEVT_EXEC** event is set, only the notification associated with these events is performed. See the **TT_PROC_SET_SCBM** request.

TTEVT_SYSCALL_ENTRY
This event flag requests notification of system call entry points. By default, all system calls stop at this event if it is selected. The information provided is the same as for **TTEVT_SYSCALL_RETURN** events but the return value and error are always zero.

TTEVT_SYSCALL_RESTART
Identical to **TTEVT_SYSCALL_ENTRY** but for system call restarts.

TTEVT_EXIT This event flag indicates that the traced process needs to notify the debugger action when it invokes **exit()**. When set, the traced thread stops while still potentially multithreaded.

TTEVT_LWP_CREATE
This event flag indicates that the debugger wants to be notified when the **lwp_create()** system call is invoked to create a thread. When set, the calling thread stops and provides the debugger with the *lwpid* of the newly created thread.

TTEVT_LWP_EXIT
This event flag indicates that the debugger wants to be notified when a thread is exiting via the **lwp_exit()** system call. The thread stops upon entry to the system call.

TTEVT_LWP_TERMINATE
This event flag indicates that the debugger wants to be notified when a caller thread invokes the **lwp_terminate()** call on a target thread. When set, the calling thread stops upon entering the system call and provides the *lwpid* of the thread to be terminated in the **ttstate_t** structure.

TTEVT_LWP_ABORT_SYSCALL
This event flag indicates that the debugger is to be notified when the **lwp_abort_syscall()** system call is invoked. The *lwpid* of the target thread is provided in the **ttstate_t** structure.

DEPENDENCIES

If the *addr* argument to a **TT_LWP_CONTINUE** or **TT_LWP_SINGLE** request is not **TT_NOPC**, 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.

Additional requests are available:

TT_LWP_RUREGS With this request, the words at offset *addr* in the **save_state** structure are returned to the tracing process. The *data* argument is the size of the read. The *addr2* argument points to the location in the debugger space where the data will be written. The *addr* argument must be word-aligned and *addr+data* must be less or equal to **sizeof (save_state_t)** (see **<machine/save_state.h>**).

NOTE: Only 4 and 8 bytes reads and writes are currently supported.

TT_LWP_WUREGS With this request, *data* bytes of data pointed to by *addr2* are written at offset *addr* in the **save_state** structure. Only these 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.

NOTE: Only 4 and 8 bytes reads and writes are currently supported.

ERRORS

If a request fails, **ttrace** returns -1 and *errno* is set to one of the following:

[EINVAL]	*request* is an illegal number.
[EINVAL]	A non-zero value has been passed in a parameter expecting a zero value or vice-versa.
[EINVAL]	The *data* argument of **TT_PROC_SETTRC** or **TT_PROC_ATTACH** is not **TT_VERSION**.
[EINVAL]	Size too large for data transfer.
[EINVAL]	Invalid signal number.
[EINVAL]	Misaligned request or not a word multiple (**TT_PROC_RDTEXT**, **TT_PROC_WRTEXT**).
[EINVAL]	Invalid signal (**TT_LWP_CONTINUE**, **TT_LWP_SINGLE**).
[EINVAL]	Invalid offset (**TT_LWP_RUREGS**, **TT_LWP_WUREGS**).
[EINVAL]	**ptrace()** and **ttrace()** requests are being mixed.
[EINVAL]	An offset in the **save_state** structure is not word-aligned.
[EINVAL]	An invalid register is targeted by **TT_LWP_WUREGS.**
[EINVAL]	The size argument to a **TT_PROC_GET_PATHNAME** is larger than **MAXPATHLEN.**
[EACCES]	The *pid* argument to the **TT_PROC_ATTACH** is the pid of the invoker.
[EACCES]	The process is already being traced.
[EACCES]	Attempting to trace a process whose binary resides on a soft/interruptible NFS mount point.
[EACCES]	The executable image of the process being attached resides across an interruptible NFS mount.
[EFAULT]	Invalid user address.
[EPERM]	The specified thread cannot be attached for tracing.
[ESRCH]	*pid* and/or *lwpid* identify a process or a thread to be traced that does not exist or has not executed a **ttrace()** with the **TT_PROC_SETTRC** request.
[EINTR]	Cannot suspend process or attach is interrupted (**TT_PROC_ATTACH**).
[EPROTO]	Attempting to stop a thread already stopped by the debugger.
[EPROTO]	Attempting to resume a thread not stopped by the debugger.
[EPROTO]	Attempting to read or write registers while the thread is not stopped.
[EPROTO]	Attempting to obtain the state of a thread which was not stopped by the debugger.
[EPROTO]	Invoked before an exec event took place (**TT_PROC_GET_PATHNAME**).
[EPROTO]	The process is exiting and the request is not allowed in this condition.
[EPROTO]	The debugger is attempting to modify wide registers after having modified narrow registers.
[EPROTO]	The debugger is attempting to modify the address space of a process in the middle of a vfork.
[ENODATA]	Data in this register is not readable or not writable at this time.

[EDEADLK] One thread of a multithreaded process (p1) has performed a **vfork()**, the child (p2) is stopped at the vfork event and the debugger is attempting to stop or resume a thread in the parent process (p1).

[ENOMEM] System is out of memory.

AUTHOR

ttrace was developed by HP.

SEE ALSO

adb(1), fork(2), vfork(2), exec(2), signal(2), wait(2), ttrace_wait(2).

STANDARDS CONFORMANCE

ttrace(): LOCAL

t

NAME

ttrace_wait - wait for ttrace event

SYNOPSIS

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

int ttrace_wait(pid_t pid, lwpid_t lwpid, ttwopt_t option,
     ttstate_t *tsp, size_t size);
```

DESCRIPTION

The **ttrace_wait()** system call provides a means to wait for a **ttrace()** event to occur. A tracing process (debugger) will normally invoke **ttrace_wait()** after a process or any of its threads has been set running.

ttrace_wait() synchronizes tracing requests directed at threads within the traced process. This mechanism differs from the process-oriented synchronization provided by **wait()** or **waitpid()** (see *wait*(2)).

The *pid* argument identifies the process-id of a traced process which the debugger expects to stop. If *pid* is a positive value, and *lwpid* is zero, then **ttrace_wait()** will wait for any thread in the traced process identified by *pid* to stop in response to an outstanding ttrace event. The information concerning the thread that hit the event point is available in the *ttstate_t* structure (see *ttrace*(2)).

The *lwpid* argument identifies the Lightweight Process (LWP) id of a thread in the traced process *pid* for which the debugger must wait to validate **ttrace()** request completion. If both *pid* and *lwpid* are non-zero values, **ttrace_wait()** suspends the calling process until the specified LWP in the traced process stops.

When multiple child processes are simultaneously traced, **ttrace_wait()** can be used to identify the process-id and LWP id of a thread which stopped in response to any outstanding **ttrace()** request established for the group of traced child processes. This is achieved by invoking **ttrace_wait()** with both *pid* and *lwpid* set to 0 (zero).

A zero *pid* and non-zero *lwpid* will return an error.

The *option* argument must specify either **TTRACE_WAITOK** or **TTRACE_NOWAIT**. These values control the synchronizing effect of **ttrace_wait()** on the calling process. The **TTRACE_NOWAIT** value causes **ttrace_wait()** to behave in non-blocking mode and return to the calling process immediately whether or not a pre-existing ttrace request completed on behalf of the tracing process. With **TTRACE_WAITOK**, **ttrace_wait()** suspends the calling process until the requested pid and/or LWP stop.

As mentioned above, the *tsp* argument references a *ttstate_t* structure (see *ttrace*(2)) which provides all the needed information regarding the stopped thread. The *size* argument specifies the size of the *ttstate_t* structure referenced by *addr*.

t

RETURN VALUE

If the call succeeds, **ttrace_wait()** will return 1 (one) if the event was never waited for, 0 (zero) otherwise. If the call fails, -1 is returned and **errno** is set to the appropriate value.

ERRORS

The **ttrace_wait()** system call fails if one or more of the following is true:

[EINVAL] *pid* is zero and *lwpid* is non-zero.

[EINVAL] The option is invalid.

[EINVAL] The *lwpid* is not controlled by process *pid*.

[ESRCH] The *pid* or *lwpid* do not identify an existing process (LWP).

[EACCES] The *pid* does not identify a process debugged by the invoking process.

[ECHILD] The process (LWP) died while it was waited for.

[EINTR] **`ttrace_wait()`** was interrupted by a signal.

[EFAULT] An invalid address was given for the kernel to write data into.

AUTHOR

`ttrace_wait()` was developed by HP.

SEE ALSO

ttrace(2), wait(2).

t

NAME

ualarm - set the interval timer

SYNOPSIS

```
#include <unistd.h>

useconds_t ualarm(useconds_t useconds, useconds_t interval);
```

DESCRIPTION

The **ualarm()** function causes the **SIGALRM** signal to be generated for the calling process after the number of real-time microseconds specified by the *useconds* argument has elapsed. When the *interval* argument is non-zero, repeated timeout notification occurs with a period in microseconds specified by the *interval* argument. If the notification signal, **SIGALRM**, is not caught or ignored, the calling process is terminated.

Implementations may place limitations on the granularity of timer values. For each interval timer, if the requested timer value requires a finer granularity than the implementation supports, the actual timer value will be rounded up to the next supported value.

Interactions between **ualarm()** and either **alarm()** or **sleep()** are unspecified.

RETURN VALUE

The **ualarm()** function returns the number of microseconds remaining from the previous **ualarm()** call. If no timeouts are pending or if **ualarm()** has not previously been called, **ualarm()** returns 0.

ERRORS

No errors are defined.

APPLICATION USAGE

The **ualarm()** function is a simplified interface to **setitimer()**, and uses the **ITIMER_REAL** interval timer.

SEE ALSO

alarm(2), getitimer(2), sleep(3C), <unistd.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

u

NAME

ulimit - get and set user limits

SYNOPSIS

```
#include <ulimit.h>

long ulimit(int cmd, ...);
```

Remarks

The ANSI C "**, ...**" construct denotes a variable length argument list whose optional [or required] members are given in the associated comment (**/* */**).

DESCRIPTION

ulimit() provides for control over process limits. Available values for *cmd* 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. The optional second argument is not used.

UL_SETFSIZE Set the file size limit of the process to the value of the optional second argument which is taken as a long. Any process can decrease this limit, but only a process with an effective user ID of super-user can 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 might not be attainable at a given time. The optional second argument is not used.

ERRORS

ulimit() fails if one or more of the following conditions is true.

[EINVAL] *cmd* is not in the correct range.

[EPERM] **ulimit()** fails and the limit is 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 to indicate the error.

SEE ALSO

brk(2), write(2).

STANDARDS CONFORMANCE

ulimit(): AES, SVID2, SVID3, XPG2, XPG3, XPG4

 umask(2)

NAME

umask - set and get file creation mask

SYNOPSIS

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

mode_t umask(mode_t cmask);
```

DESCRIPTION

umask() sets the process's file mode creation mask to **umask()** 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(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

umount - unmount a file system

SYNOPSIS

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

int umount(const 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 user with the appropriate privilege.

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 a user with appropriate privileges.

[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. Updating of **/etc/mnttab** is performed by the **mount** and **syncer** commands (see *mount*(1M) and *syncer*(1M) for more information).

SEE ALSO

mount(1M), syncer(1M), mount(2), vfsmount(2).

STANDARDS CONFORMANCE

umount(): SVID2, SVID3, XPG2

NAME

uname(), setuname() - get information about computer system; set node name (system name)

SYNOPSIS

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

int uname(struct utsname *name);

int setuname(const char *name, size_t namelen);
```

DESCRIPTION

uname()

The **uname()** system call places information identifying the computer system in the **utsname** structure pointed to by *name*.

The **utsname** structure, defined in **<sys/utsname.h>**, is set up as follows:

```
#define UTSLEN  9
#define SNLEN  15

char sysname[UTSLEN];
char nodename[UTSLEN];
char release[UTSLEN];
char version[UTSLEN];
char machine[UTSLEN];
char idnumber[SNLEN];
```

Each field is a null-terminated string.

The **sysname** field contains the name of the operating system, **HP-UX** on standard HP-UX systems.

The **nodename** field contains the name by which the computer system is known in a communications network.

The **release** field contains the release identifier of the operating system, such as **A.09.01**.

The **version** field contains additional information about the operating system. This value can change in future releases. The first character of the **version** field identifies the license level:

- **A** Two-user system
- **B** 16-user system
- **C** 32-user system
- **D** 64-user system
- **E** 8-user system
- **U** 128-user, 256-user, or unlimited-user system

The **machine** field contains the hardware and model identifiers of the computer system.

The **idnumber** field contains a unique identification number within that class of hardware, possibly a hardware or software serial number. This field contains a null string if there is no identification number.

setuname()

The **setuname()** system call sets the node name (system name), as returned in the **nodename** field of the **utsname** structure, to *name*, which has a length of *namelen* characters. This is usually executed by **/sbin/init.d/hostname** at system boot time. Names are limited to **UTSLEN - 1** characters; **UTSLEN** is defined in **<sys/utsname.h>**.

u

RETURN VALUE

uname() and **setuname()** return the following values:

- *n* Successful completion. *n* is a nonnegative value.
- **-1** Failure. **errno** is set to indicate the error.

ERRORS

If **uname()** or **setuname()** fails, **errno** is set to one of the following values.

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

[EPERM] **setuname()** was attempted by a process lacking appropriate privileges.

AUTHOR

uname() was developed by AT&T and HP.

SEE ALSO

hostname(1), uname(1), setuname(1M), gethostname(2), sethostname(2).

STANDARDS CONFORMANCE

uname(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

unlink - remove directory entry; delete file

SYNOPSIS

```
#include <unistd.h>

int unlink(const char *path);
```

DESCRIPTION

The **unlink()** system call 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, only the directory entry is removed immediately so that processes that do not already have the file open cannot access the file. After all processes close their references to the file, if there are no more links to the file, the space occupied by the file is then freed and the file ceases to exist.

RETURN VALUE

unlink() returns the following values:

0 Successful completion.
-1 Failure. **errno** is set to indicate the error.

ERRORS

If **unlink()** fails, **errno** is set to one of the following values:

[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	Write permission is denied on the directory containing the link to be removed.
[EACCES]	The process does not have read/write access permission to the parent directory.
[EBUSY]	The entry to be unlinked is the mount point for a mounted file system.
[EFAULT]	*path* points outside the process's allocated address space. 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.
[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.
[EPERM]	The directory containing the file to be removed has the sticky bit set and neither the containing directory nor the file to be removed are owned by the effective user ID.
[EPERM]	The named file is a directory and the effective user ID is not a user with appropriate privileges.
[EROFS]	The directory entry to be unlinked is part of a read-only file system.
[ETXTBSY]	The entry to be unlinked is the last link to a pure procedure (shared text) file that is being executed.

WARNINGS

If **unlink()** is used on a directory that is *not* empty (contains files other than **.** and **..**), the directory is unlinked, the files become orphans, and the directory link count is left with an inaccurate value unless they are linked by some other directory.

If **unlink()** is used on a directory that *is* empty (contains only the files **.** and **..**), the directory is unlinked, but the parent directory's link count is left with an inaccurate value.

In either of the above cases, the file system should be checked using **fsck** (see *fsck*(1M)). To avoid these types of problems, use **rmdir()** instead (see *rmdir*(2)).

SEE ALSO

rm(1), close(2), link(2), open(2), rmdir(2), remove(3C).

STANDARDS CONFORMANCE

unlink(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

usleep - suspend execution for an interval

SYNOPSIS

#include <unistd.h>

int usleep(useconds_t useconds);

DESCRIPTION

The **usleep()** function suspends the current process from execution for the number of microseconds specified by the *useconds* argument. Because of other activity, or because of the time spent in processing the call, the actual suspension time may be longer than the amount of time specified.

The *useconds* argument must be less than 1,000,000. If the value of *useconds* is 0, then the call has no effect.

The **usleep()** function uses the process' real-time interval timer to indicate to the system when the process should be woken up.

There is one real-time interval timer for each process. The **usleep()** function will not interfere with a previous setting of this timer. If the process has set this timer prior to calling **usleep()**, and if the time specified by *useconds* equals or exceeds the interval timer's prior setting, the process will be woken up shortly before the timer was set to expire.

Implementations may place limitations on the granularity of timer values. For each interval timer, if the requested timer value requires a finer granularity than the implementation supports, the actual timer value will be rounded up to the next supported value.

Interactions between **usleep()** and either **alarm()** or **sleep()** are unspecified.

RETURN VALUE

On successful completion, **usleep()** returns 0. Otherwise, it returns −1 and sets **errno** to indicate the error.

ERRORS

The **usleep()** function may fail if:

[EINVAL] The time interval specified 1,000,000 or more microseconds.

APPLICATION USAGE

The **usleep()** function is included for its historical usage. The **setitimer()** function is preferred over this function.

SEE ALSO

alarm(2), getitimer(2), sigaction(2), sleep(3C), <unistd.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

u

NAME

ustat() - get mounted file system statistics

SYNOPSIS

```
#include <ustat.h>

int ustat(dev_t dev, struct ustat *buf);
```

DESCRIPTION

The **ustat()** system call returns information about a mounted file system. *dev* is a device number identifying a device containing a mounted file system. *buf* is a pointer to a **ustat** structure (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 or null */
char     f_fpack[6];    /* Filsys pack name or null */
int      f_blksize;     /* Block size */
```

The value of **f_tfree** is the number of free blocks of size **f_blksize**.

RETURN VALUE

ustat() returns the following values:

- **0** Successful completion.
- **-1** Failure. **errno** is set to indicate the error.

ERRORS

If **ustat()** fails, **errno** is set to one of the following values.

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

[EINVAL] *dev* is not the device number of a device containing a mounted file system.

WARNINGS

For some file systems, the number of free inodes does not change. Such file systems will return **-1** in the field **f_tinode**.

For some file systems, the inodes can be dynamically allocated. For such file systems, the field **f_tinode** contains the number of free inodes at the current time.

AUTHOR

ustat() was developed by AT&T and HP.

SEE ALSO

touch(1), stat(2), statvfs(2), fs(4), fs_vxfs(4).

STANDARDS CONFORMANCE

ustat(): SVID2, SVID3, XPG2

NAME

utime() - set file access and modification times

SYNOPSIS

```
#include <utime.h>

int utime(const char *path, const struct utimbuf *times);
```

DESCRIPTION

The **utime()** system call 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 a user with appropriate privileges can use **utime()** this way.

The following times in the **utimbuf** structure defined in **<utime.h>** are measured in seconds since 00:00:00 UTC (Coordinated Universal Time), January 1, 1970.

```
time_t actime;     /* access time */
time_t modtime;    /* modification time */
```

RETURN VALUE

utime() returns the following values:

- **0** Successful completion.
- **-1** Failure. **errno** is set to indicate the error.

ERRORS

If **utime()** fails, **errno** is set to one of the following values.

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

[EACCES] The effective user ID is not a user with appropriate privileges, and not the owner of the file, *times* is a NULL pointer, and write access is denied.

[EFAULT] *times* is not a NULL pointer, and it 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.

[ENOENT] The named file does not exist.

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

[EPERM] The effective user ID is not a user with appropriate privileges. and not the owner of the file, and *times* is not a NULL pointer.

[EROFS] The file system containing the file is mounted read-only.

DEPENDENCIES

NFS

utime() may return [EPERM] when invoked on a remote file owned by a superuser, even if the invoking user has write permission on the file.

SEE ALSO

touch(1), stat(2).

STANDARDS CONFORMANCE

utime(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

utimes - set file access and modification times

SYNOPSIS

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

int utimes(const char *path, const struct timeval times[2]);
```

DESCRIPTION

The **utimes()** function sets the access and modification times of the file pointed to by the *path* argument to the value of the *times* argument. The **utimes()** function allows time specifications accurate to the microsecond.

For **utimes()**, the *times* argument is an array of **timeval** structures. The first array member represents the date and time of last access, and the second member represents the date and time of last modification. The times in the **timeval** structure are measured in seconds and microseconds since the Epoch, although rounding toward the nearest second may occur.

If the *times* argument is a null pointer, the access and modification times of the file are set to the current time. The effective user ID of the process must be the same as the owner of the file, or must have write access to the file or appropriate privileges to use this call in this manner. Upon completion, **utimes()** will mark the time of the last file status change, *st_ctime*, for update.

RETURN VALUE

Upon successful completion, 0 is returned. Otherwise, −1 is returned and **errno** is set to indicate the error, and the file times will not be affected.

ERRORS

The utimes() function will fail if:

[EACCES] Search permission is denied by a component of the *path* prefix; or the *times* argument is a null pointer and the effective user ID of the process does not match the owner of the file and write access is denied.

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

[ENAMETOOLONG] The length of the *path* argument exceeds **{PATH_MAX}** or a pathname component is longer than **{NAME_MAX}**.

[ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

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

[EPERM] The *times* argument is not a null pointer and the calling process' effective user ID has write access to the file but does not match the owner of the file and the calling process does not have the appropriate privileges.

[EROFS] The file system containing the file is read-only.

The **utimes()** function may fail if:

[ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result whose length exceeds **{PATH_MAX}.**

u

SEE ALSO

<sys/time.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

NAME

vfork - spawn new process; share virtual memory

SYNOPSIS

```
#include <unistd.h>

pid_t vfork(void);
```

REMARKS

vfork() is a higher performance version of **fork()** that is provided on some systems where a performance advantage can be attained.

If the calling process is multi-threaded, the newly created child process will only contain one thread. This one thread will be a copy of the thread calling **vfork()**.

vfork() differs from **fork()** only in that the child process can share code and data with the calling process (parent process). This speeds 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 that relies upon the differences between **fork()** and **vfork()** is not portable across HP-UX systems.

All HP-UX implementations must provide the entry **vfork()**, but it is permissible for them to treat it identically to **fork**. On some implementations the two are not distinguished because the **fork()** implementation is as efficient as possible. Other versions may do the same to avoid the overhead of supporting 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()** (see *exec*(2)), the data space copied from the parent to the child by **fork()** is not used. This is particularly inefficient in a paged environment, making **vfork** is particularly useful. Depending upon the size of the parent's data space, **vfork()** 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()** or abnormally (see *exec*(2) and *exit*(2)). 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()** flushes and closes standard I/O channels, thereby damaging 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 is created if any of the following conditions are encountered:

[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.

V

DEPENDENCIES

Series 800

Process times for the parent and child processes within the [**vfork**,**exec**] window may be inaccurate.

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()** (see *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()** or **sigvector()** is used (see *sigaction*(2) and *sigvector*(2)).

AUTHOR

vfork() was developed by the University of California, Berkeley.

SEE ALSO

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

V

NAME

vfsmount - mount a file system

SYNOPSIS

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

int vfsmount(int type,
             const char *dir,
             int flags,
             caddr_t data);
```

Remarks

This routine is included only for compatibility with past releases. It works only with UFS (HFS), NFS, and CDFS file systems. For maximum portability and improved functionality, new applications should use the **mount()** system call (see *mount*(2)).

DESCRIPTION

The **vfsmount()** system call attaches a file system to a directory. After a successful return, references to directory *dir* 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.

type indicates the type of the file system. It must be one of the types described below. **vfsmount()** does not check that the file system is actually of type *type*; if *type* is incorrect, **vfsmount()** may cause the process to hang. To prevent such problems, **statfsdev()** (see *statfsdev*(3C)) should be called before **vfsmount()** to check the file system type, which **statfsdev()** places in the **f_fsid[1]** field of the **statfs** structure that it returns.

The *flags* argument determines whether the file system can be written to. It also controls whether programs from the mounted file system are allowed to have set-user-ID execution. Physically write-protected and magnetic tape file systems must be mounted read-only. Failure to do so results 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-user-ID programs not permitted.

data is a pointer to a structure containing arguments specific to the value contained in *type*. The following values for *type* are defined in **<sys/mount.h>**:

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.

MOUNT_UFS Mount a local HFS file system. *data* points to a structure of the following format:

```
struct ufs_args {
     char    *fspec;
     int      flags;
};
```

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 of **mount()** (see *mount*(2)).

flags points to a bit map that sets options. The following values of the bits are defined in **<sys/mount.h>**:

MS_DELAY Specify that the writes to disks are to be delayed till the buffer needs to be reused. This is the default on Series 800 systems, as it was prior to release 10.0.

MS_BEHIND Specify that the writes to disks are to be done asynchronously, where possible, without waiting for completion. This is the default on Series 700 systems, as it was prior to release 10.0.

V

MS_BEHIND and **MS_DELAY** are mutually exclusive.

MS_NO_FSASYNC Specify that rigorous posting of file system metadata is to be used. This is the default.

MS_FSASYNC Specify that relaxed posting of file system metadata is to be used. This may lead to better performance for certain applications; but there is increased potential for data loss in case of a crash.

MS_FSASYNC and **MS_NO_FSASYNC** are mutually exclusive.

NOTES

The **MOUNT_NFS** type is no longer supported through this interface.

RETURN VALUE

vfsmount() returns the following values:

0 Successful completion.
-1 Failure. No file system is mounted. **errno** is set to indicate the error.

ERRORS

If **vfsmount()** fails, **errno** is set to one of the following values.

[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 superblock 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.

[EINVAL] *type* is not **MOUNT_UFS**, or **MOUNT_CDFS**.

[EIO] An I/O error occurred while reading from or writing to the file system.

[EIO] An attempt was made to mount a physically write protected or magnetic tape file system as read-write.

[ELOOP] Too many symbolic links were encountered while 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* is longer than **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 message can occur only during a local mount.

V

[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 (indicating that no device driver exists for the associated hardware).

[EPERM] The caller does not have appropriate privileges.

DEPENDENCIES

NFS

If **vfsmount()** fails, **errno** can also be set to one of the following values.

[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.

See *mountd*(1M), *getfh*(2), and *inet*(7F) for more information.

WARNINGS

The **mount** command (see *mount*(1M)) is preferred over **vfsmount()** because **mount** supports all mounting options that are available from **vfsmount()** directly, plus **mount** 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(1M), mount(2), umount(2).

NAME

wait, waitpid - wait for child process to stop or terminate

SYNOPSIS

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

pid_t wait(int *stat_loc);

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

DESCRIPTION

The **wait()** and **waitpid()** functions allow the calling process to obtain status information pertaining to one of its child processes. Various options permit status information to be obtained for child processes that have terminated or stopped. If status information is available for two or more child processes, the order in which their status is reported is unspecified.

The **wait()** function will suspend execution of the calling process until status information for one of its terminated child processes is available, or until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. If status information is available prior to the call to **wait()**, return will be immediate.

The **waitpid()** function will behave identically to **wait()**, if the *pid* argument is (pid_t)*–1* and the *options* argument is 0. Otherwise, its behaviour will be modified by the values of the *pid* and *options* arguments.

The *pid* argument specifies a set of child processes for which status is requested. The **waitpid()** function will only return the status of a child process from this set:

- If *pid* is equal to (*pid_t*)–1, status is requested for any child process. In this respect, **waitpid()** is then equivalent to **wait()**.
- If pid is greater than 0, it specifies the process ID of a single child process for which status is requested.
- If pid is 0, status is requested for any child process whose process group ID is equal to that of the calling process.
- If *pid* is less than (*pid_t*)–1, status is requested for any child 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, defined in the header **<sys/wait.h>**.

WCONTINUED — The **waitpid()** function will report the status of any continued child process specified by *pid* whose status has not been reported since it continued from a job control stop.

WNOHANG — The **waitpid()** function will not suspend execution of the calling process if status is not immediately available for one of the child processes specified by *pid*.

WUNTRACED — The status of any child processes specified by *pid* that are stopped, and whose status has not yet been reported since they stopped, will also be reported to the requesting process.

If the calling process has **SA_NOCLDWAIT** set or has **SIGCHLD** set to **SIG_IGN**, and the process has no unwaited for children that were transformed into zombie processes, it will block until all of its children terminate, and **wait()** and **waitpid()** will fail and set **errno** to **ECHILD**.

If **wait()** or **waitpid()** return because the status of a child process is available, these functions will return a value equal to the process ID of the child process. In this case, if the value of the argument *stat_loc* is not a null pointer, information will be stored in the location pointed to by *stat_loc*. If and only if the status returned is from a terminated child process that returned 0 from **main()** or passed 0 as the status argument to **_exit()** or **exit()**, the value stored at the location pointed to by *stat_loc* will be 0. Regardless of its value, this information may be interpreted using the following macros, which are defined in **<sys/wait.h>** and evaluate to integral expressions; the *stat_val* argument is the integer value pointed to by *stat_loc*.

WIFEXITED(stat_val)
Evaluates to a non-zero value if status was returned for a child process that terminated normally.

WEXITSTATUS(stat_val)
If the value of **WIFEXITED(stat_val)** is non-zero, this macro evaluates to the low-order 8 bits of the *status* argument that the child process passed to **_exit()** or **exit()**, or the value the child process returned from **main()**.

WIFSIGNALED(stat_val)
Evaluates to non-zero value if status was returned for a child process that terminated due to the receipt of a signal that was not caught (see <*signal.h*>).

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 of the child process.

WIFSTOPPED(stat_val)
Evaluates to a non-zero value if status was returned for a child process that is currently stopped.

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 child process to stop.

WIFCONTINUED(stat_val)
Evaluates to a non-zero value if status was returned for a child process that has continued from a job control stop.

If the information pointed to by *stat_loc* was stored by a call to **waitpid()** that specified the **WUNTRACED** flag and did not specify the **WCONTINUED** flag, exactly one of the macros **WIFEXITED(*stat_loc)**, **WIFSIGNALED(*stat_loc)**, and **WIFSTOPPED(*stat_loc)** will evaluate to a non-zero value.

If the information pointed to by *stat_loc* was stored by a call to **waitpid()** that specified the **WUNTRACED** and **WCONTINUED** flags, exactly one of the macros **WIFEXITED(*stat_loc)**, **WIFSIGNALED(*stat_loc)**, **WIFSTOPPED(*stat_loc)**, and **WIFCONTINUED (*stat_loc)** will evaluate to a non-zero value.

If the information pointed to by *stat_loc* was stored by a call to **waitpid()** that did not specify the **WUNTRACED** or **WCONTINUED** flags, or by a call to the **wait()** function, exactly one of the macros **WIFEXITED(*stat_loc)** and **WIFSIGNALED(*stat_loc)** will evaluate to a non-zero value.

If the information pointed to by *stat_loc* was stored by a call to **waitpid()** that did not specify the **WUNTRACED** flag and specified the **WCONTINUED** flag, by a call to the **wait()** function, exactly one of the macros **WIFEXITED(*stat_loc)**, **WIFSIGNALED(*stat_loc)**, and **WIFCONTINUED(*stat_loc)** will evaluate to a non-zero value.

There may be additional implementation-dependent circumstances under which **wait()** or **waitpid()** report status. This will not occur unless the calling process or one of its child processes explicitly makes use of a non-standard extension. In these cases the interpretation of the reported status is implementation-dependent.

If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes will be assigned a new parent process ID corresponding to an implementation-dependent system process.

W

RETURN VALUE

If **wait()** or **waitpid()** returns because the status of a child process is available, these functions will return a value equal to the process ID of the child process for which status is reported. If **wait()** or **waitpid()** returns due to the delivery of a signal to the calling process, −1 will be returned and **errno** will be set to **EINTR**. If **waitpid()** was invoked with **WNOHANG** set in *options*, it has at least one child process specified by *pid* for which status is not available, and status is not available for any process specified by *pid*, 0 will be returned. Otherwise, *(pid_t)*−1 will be returned, and **errno** will be set to indicate the error.

ERRORS

The **wait()** function will fail if:

[ECHILD] The calling process has no existing unwaited-for child processes.

[EINTR] The function was interrupted by a signal. The value of the location pointed to by *stat_loc* is undefined.

The **waitpid()** function will fail if:

[ECHILD] The process or process group specified by *pid* does not exist or is not a child of the calling process.

[EINTR] The function was interrupted by a signal. The value of the location pointed to by *stat_loc* is undefined.

[EINVAL] The *options* argument is not valid.

APPLICATION USAGE

Threads Considerations

In a multi-threaded application, only the calling thread is suspended by **wait()** or **waitpid()**.

wait() and **waitpid()** will not return until all threads in the process have reached the desired state. For example, **wait()** and **waitpid()** will not return until all threads have terminated. If the **WUNTRACED** or **WCONTINUED** options are specified for **waitpid()**, the function will not return until all threads have stopped or continued respectively.

SEE ALSO

exec(2), exit(2), fork(2), wait3(2), waitid(2), <sys/types.h>, <sys/wait.h>.

CHANGE HISTORY

First released in Issue 1.

Derived from Issue 1 of the SVID.

Issue 4

The following change is incorporated for alignment with the ISO POSIX-1 standard:

- Text describing conditions under which 0 will be returned when **WNOHANG** is set in *options* is added to the RETURN VALUE section.

Other changes are incorporated as follows:

- The header **<sys/types.h>** is now marked as optional (OH); this header need not be included on XSI-conformant systems.
- Error return values throughout the DESCRIPTION and RETURN VALUE sections are changed to show the proper casting (that is, (*pid_t*)–1.
- The words "If the implementation supports job control" are removed from the description of **WUNTRACED**. This is because job control is defined as mandatory for Issue 4 conforming implementations.

Issue 4, Version 2

The following changes are incorporated in the DESCRIPTION for X/OPEN UNIX conformance:

- The **WCONTINUED** *options* flag and the **WIFCONTINUED(stat_val)** macro are added.
- Text following the list of options flags explains the implications of setting the **SA_NOCLDWAIT** signal flag, or setting **SIGCHILD** to **SIG_IGN**.
- Text following the list of macros, which explains what macros return non-zero values in certain cases, is expanded and the value of the **WCONTINUED** flag on the previous call to **waitpid()** is taken into account.

HP-UX EXTENSIONS

NAME

wait(), waitpid() - wait for child or traced process to stop or terminate

DESCRIPTION

wait()

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.

WCOREDUMP(*stat_loc)
: If the value of **WIFSIGNALED(*stat_loc)** is nonzero, this macro evaluates to a nonzero value if a "core image" was produced (see *signal*(5)).

waitpid()

WNOWAIT
: Keep the process whose status is returned in **stat_loc* in a waitable state. The process may be waited for again with identical results, provided the state of the process doesn't change in the interim.

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()**) 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.

Notes

Earlier HP-UX versions documented the bit encodings of the status returned by **wait()** rather than the macros **WCOREDUMP**, **WEXITSTATUS**, **WIFEXITED**, **WIFSIGNALED**, **WIFSTOPPED**, **WSTOPSIG**, and **WTERMSIG**. 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 **WIFEXITED**, **WIFSIGNALED**, and **WIFSTOPPED** have the same definitions as the correspondingly 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

If **wait()** or **waitpid()** fails, **errno** is set to one of the following values.

[EACCES]
: The calling process of **waitpid()** does not have read permission to the *pid*.

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

WARNINGS

The behavior of **wait()** and **waitpid()** is affected if the **SIGCLD** signal is set to **SIG_IGN**. See the WARNINGS section of *signal*(5). Signal handlers that cause system calls to be restarted can affect the **EINTR** condition described above (see *bsdproc*(3C), *sigaction*(2), and *sigvector*(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(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

waitpid(): AES, SVID3, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

wait3 - wait for child process to change state

SYNOPSIS

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

pid_t wait3 (int *stat_loc, int options,
     struct rusage *resource_usage);
```

DESCRIPTION

The **wait3()** function allows the calling process to obtain status information for specified child processes.

The following call:

```
wait3(stat_loc, options, resource_usage);
```

is equivalent to the call:

```
waitpid((pid_t)-1, stat_loc, options);
```

except that on successful completion, if the *resource_usage* argument to **wait3()** is not a null pointer, the **rusage** structure that the third argument points to is filled in for the child process identified by the return value.

RETURN VALUE

See *wait*(2).

ERRORS

In addition to the error conditions specified on **waitpid()**, under the following conditions, **wait3()** may fail and set **errno** to:

[ECHILD]	The calling process has no existing unwaited-for child processes, or if the set of processes specified by the argument *pid* can never be in the states specified by the argument *options*.

APPLICATION USAGE

Threads Considerations

In a multi-threaded application, only the calling thread is suspended by **wait3()**.

wait3() will not return until all threads in the process have reached the desired state. For example, **wait3()** will not return until all threads have terminated. If the **WUNTRACED** or **WCONTINUED** options are specified, **wait3()** will not return until all threads have stopped or continued respectively.

SEE ALSO

exec(2), exit(2), fork(2), pause(2), <sys/wait.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

HP-EXTENSIONS

SYNOPSIS

```
pid_t wait3(int *stat_loc, int options, int *reserved);
```

DESCRIPTION

The **wait3()** system call is equivalent to **waitpid()** with the value of *pid* equal to zero. The third parameter to **wait3()**, *reserved*, is currently unused and must always be a null pointer.

ERRORS

If **wait3()** fails, **errno** is set to one of the following values.

[EINVAL]	The *options* argument to **waitpid()** or **wait3()** is invalid.
[EINVAL]	**wait3()** was passed a nonnull pointer value for its third argument.

WARNINGS

The behavior of **wait3()** is affected if the **SIGCLD** signal is set to **SIG_IGN**. See the WARNINGS section of *signal*(5). Signal handlers that cause system calls to be restarted can affect the EINTR condition described above (see *bsdproc*(3C), *sigaction*(2), and *sigvector*(2)).

AUTHOR

wait3() was 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).

W

NAME

waitid - wait for child process to change state

SYNOPSIS

#include <sys/wait.h>

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);

DESCRIPTION

The **waitid()** function suspends the calling process until one of its children changes state. It records the current state of a child in the structure pointed to by *infop*. If a child process changed state prior to the call to **waitid()**, **waitid()** returns immediately.

The *idtype* and *id* arguments are used to specify which children **waitid()** will wait for.

If *idtype* is **P_PID**, **waitid()** will wait for the child with a process ID equal to *(pid_t)pid*.

If *idtype*is **P_PGID**, **waitid()** will wait for any child with a process group ID equal to *(pid_t)pid*.

If *idtype*is **P_ALL**, **waitid()** will wait for any children and id is ignored.

The *options* argument is used to specify which state changes **waitid()** will wait for. It is formed by OR-ing together one or more of the following flags:

WEXITED Wait for processes that have exited.

WSTOPPED Status will be returned for any child that has stopped upon receipt of a signal.

WCONTINUED Status will be returned for any child that was stopped and has been continued.

WNOHANG Return immediately if there are no children to wait for.

WNOWAIT Keep the process whose status is returned in *infop* in a waitable state. This will not affect the state of the process; the process may be waited for again after this call completes.

The *infop* argument must point to a **siginfo_t** structure. If **waitid()** returns because a child process was found that satisfied the conditions indicated by the arguments *idtype* and *options*, then the structure pointed to by *infop* will be filled in by the system with the status of the process. The *si_signo* member will always be equal to **SIGCHLD**.

RETURN VALUE

If **waitid()** returns due to the change of state of one of its children, 0 is returned. Otherwise, −1 is returned and **errno** is set to indicate the error.

ERRORS

The **waitid()** function will fail if:

[ECHILD] The calling process has no existing unwaited-for child processes.

[EINTR] The **waitid()** function was interrupted due to the receipt of a signal by the calling process.

[EINVAL] An invalid value was specified for *options*, or *idtype* and *id* specify an invalid set of processes.

APPLICATION USAGE

Threads Considerations

In a multi-threaded application, only the calling thread is suspended by **waitid()**.

waitid() will not return until all threads in the process have reached the desired state. For example, if the **WEXITED**, **WSTOPPED** or **WCONTINUED** options are specified, **waitid()** will not return until all threads in the process have terminated, stopped or continued respectively.

SEE ALSO

exec(2), exit(2), wait(2), <sys/wait.h>.

W

CHANGE HISTORY

First released in Issue 4, Version 2.

NAME

write, writev - write on a file

SYNOPSIS

```
#include <unistd.h>

ssize_t write(int fildes, const void *buf, size_t nbyte);

#include <sys/uio.h>

ssize_t writev(int fildes, const struct iovec *iov, int iovcnt);
```

DESCRIPTION

The **write()** function attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the open file descriptor, *fildes*.

If *nbyte* is 0, **write()** will return 0 and have no other results if the file is a regular file; otherwise, the results are unspecified.

On a regular file or other file capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file offset associated with *fildes*. Before successful return from **write()**, the file offset is incremented by the number of bytes actually written. On a regular file, if this incremented file offset is greater than the length of the file, the length of the file will be set to this file offset. If the **O_SYNC** flag of the file status flags is set and *fildes* refers to a regular file, a successful **write()** does not return until the data is delivered to the underlying hardware. On a file not capable of seeking, writing always takes place starting at the current position. The value of a file offset associated with such a device is undefined.

If the **O_APPEND** flag of the file status flags is set, the file offset will be set to the end of the file prior to each write and no intervening file modification operation will occur between changing the file offset and the write operation.

If a **write()** requests that more bytes be written than there is room for (for example, the *ulimit* or the physical end of a medium), only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes will return 20. The next write of a non-zero number of bytes will give a failure return (except as noted below) and the implementation will generate a **SIGXFSZ** signal for the process.

If **write()** is interrupted by a signal before it writes any data, it will return –1 with **errno** set to **EINTR**.

If **write()** is interrupted by a signal after it successfully writes some data, it will return the number of bytes written.

If the value of *nbyte* is greater than **{SSIZE_MAX}**, the result is implementation-dependent.

After a **write()** to a regular file has successfully returned:

- Any successful **read()** from each byte position in the file that was modified by that write will return the data specified by the **write()** for that position until such byte positions are again modified.
- Any subsequent successful **write()** to the same byte position in the file will overwrite that file data.

Write requests to a pipe or FIFO will be handled the same as a regular file with the following exceptions:

- There is no file offset associated with a pipe, hence each write request will append to the end of the pipe.
- Write requests of **{PIPE_BUF}** bytes or less will not be interleaved with data from other processes doing writes on the same pipe. Writes of greater than **{PIPE_BUF}** bytes may have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the **O_NONBLOCK** flag of the file status flags is set.
- If the **O_NONBLOCK** flag is clear, a write request may cause the process to block, but on normal completion it will return *nbyte*.
- If the **O_NONBLOCK** flag is set, **write()** requests will be handled differently, in the following ways:

- The **write()** function will not block the process.
- A write request for **{PIPE_BUF}** or fewer bytes will have the following effect: If there is sufficient space available in the pipe, **write()** will transfer all the data and return the number of bytes requested. Otherwise, **write()** will transfer no data and return –1 with **errno** set to **EAGAIN**.
- A write request for more than **{PIPE_BUF}** bytes will case one of the following:
  a. When at least one byte can be written, transfer what it can and return the number of bytes written. When all data previously written to the pipe is read, it will transfer at least **{PIPE_BUF}** bytes.
  b. When no data can be written, transfer no data and return –1 with **errno** set to **EAGAIN**.

When attempting to write to a file descriptor (other than a pipe or FIFO) that supports non-blocking writes and cannot accept the data immediately:

- If the **O_NONBLOCK** flag is clear, **write()** will block until the data can be accepted.
- If the **O_NONBLOCK** flag is set, **write()** will not block the process. If some data can be written without blocking the process, **write()** will write what it can and return the number of bytes written. Otherwise, it will return –1 and **errno** will be set to **EAGAIN**.

Upon successful completion, where **nbyte** is greater than 0, **write()** will mark for update the *st_ctime* and *st_mtime* fields of the file, and if the file is a regular file, the **S_ISUID** and **S_ISGID** bits of the file mode may be cleared.

If *fildes* refers to a STREAM, the operation of **write()** is determined by the values of the minimum and maximum *nbyte* range ("packet size") accepted by the STREAM. These values are determined by the top-most STREAM module. If *nbyte* falls within the packet size range, *nbyte* bytes will be written. If *nbyte* does not fall within the range and the minimum packet size value is 0, **write()** will break the buffer into maximum packet size segments prior to sending the data downstream (the last segment may contain less than the maximum packet size). If *nbyte* does not fall within the range and the minimum value is non-zero, **write()** will fail with **errno** set to **ERANGE**. Writing a zero-length buffer (*nbyte* is 0) to a STREAMS device sends 0 bytes with 0 returned. However, writing a zero-length buffer to a STREAMS-based pipe or FIFO sends no message and 0 is returned. The process may issue **I_SWROPT ioctl()** to enable zero-length messages to be sent across the pipe or FIFO.

When writing to a STREAM, data messages are created with a priority band of 0. When writing to a STREAM that is not a pipe or FIFO:

- If **O_NONBLOCK** is clear, and the **STREAM** cannot accept data (the **STREAM** write queue is full due to internal flow control conditions), **write()** will block until data can be accepted.
- If **O_NONBLOCK** is set and the **STREAM** cannot accept data, **write()** will return –1 and set **errno** to **[EAGAIN]**.
- If **O_NONBLOCK** is set and part of the buffer has been written while a condition in which the **STREAM** cannot accept additional data occurs, **write()** will terminate and return the number of bytes written.

In addition, **write()** and **writev()** will fail if the **STREAM** head had processed an asynchronous error before the call. In this case, the value of **errno** does not reflect the result of **write()** or **writev()** but reflects the prior error.

The **writev()** function is equivalent to **write()**, but gathers the output data from the *iovcnt* buffers specified by the members of the *iov* array: *iov*[*0*], *iov*[*1*], ..., *iov*[*iovcnt*–1]. *iovcnt* is valid if greater than 0 and less than or equal to **{IOV_MAX}**, defined in **<limits.h>**.

Each *iovec* entry specifies the base address and length of an area in memory from which data should be written. The **writev()** function will always write a complete area before proceeding to the next.

If *fildes* refers to a regular file and all of the *iov_len* members in the array pointed to by *iov* are 0, **writev()** will return 0 and have no other effect. For other file types, the behaviour is unspecified.

If the sum of the *iov_len* values is greater than **SSIZE_MAX**, the operation fails and no data is transferred.

RETURN VALUE

Upon successful completion, **write()** will return the number of bytes actually written to the file associated with *fildes*. This number will never be greater than *nbyte*. Otherwise, –1 is returned and **errno** is set to indicate the error.

Upon successful completion, **writev()** returns the number of bytes actually written. Otherwise, it returns a value of –1, the file-pointer remains unchanged, and **errno** is set to indicate an error.

ERRORS

The **write()** and writev() functions will fail if:

- [EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor and the process would be delayed in the **write()** operation.
- [EBADF] The *fildes* argument is not a valid file descriptor open for writing.
- [EFBIG] An attempt was made to write a file that exceeds the implementation-dependent maximum file size or the process' file size limit.
- [EINTR] The write operation was terminated due to the receipt of a signal, and no data was transferred.
- [EIO] A physical I/O error has occurred.
- [EIO] The process is a member of a background process group attempting to write to its controlling terminal, **TOSTOP** is set, the process is neither ignoring nor blocking **SIGTTOU** and the process group of the process is orphaned. This error may also be returned under implementation-dependent conditions.
- [ENOSPC] There was no free space remaining on the device containing the file.
- [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by any process, or that only has one end open. A **SIGPIPE** signal will also be sent to the process.
- [ERANGE] The transfer request size was outside the range supported by the **STREAMS** file associated with *fildes*.

The **writev()** function will fail if:

- [EINVAL] The sum of the *iov_len* values in the *iov* array would overflow an *ssize_t*.

The **write()** and **writev()** functions may fail if:

- [EINVAL] The **STREAM** or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.
- [ENXIO] A request was made of a non-existent device, or the request was outside the capabilities of the device.
- [ENXIO] A hangup occurred on the **STREAM** being written to.

A write to a **STREAMS** file may fail if an error message has been received at the **STREAM** head. In this case, **errno** is set to the value included in the error message.

The **writev()** function may fail and set **errno** to:

- [EINVAL] The *iovcnt* argument was less than or equal to 0, or greater than **{IOV_MAX}**.

SEE ALSO

chmod(2), creat(2), dup(2), fcntl(2), getrlimit(2), lseek(2), open(2), pipe(2), ulimit(2), <limits.h>, <stropts.h>, <sys/uio.h>, <unistd.h>.

W

CHANGE HISTORY

First released in Issue 1.

Derived from Issue 1 of the SVID.

Issue 4

The following changes are incorporated for alignment with the ISO POSIX-1 standard:

- The type of the argument *buf* is changed from *char ** to *const void**, and the type of the argument *byte* is changed from *unsigned size_t*.
- The DESCRIPTION section is changed:

- to indicate that writing at end-of-file is atomic
- to identify that **{SSIZE_MAX}** is now used to determine the maximum value of *nbyte*
- to indicate the consequences of activities after a call to the **write()** function
- To improve clarity, the text describing operations on pipes or FIFOs when **O_NONBLOCK** is set is restructured.

Other changes are incorporated as follows:

- The header **<unistd.h>** is added to the SYNOPSIS section.
- Reference to *ulimit* in the DESCRIPTION section is marked as an extension.
- Reference to the process' file size limit and the **ulimit()** function are marked as extensions in the description of the **EFBIG** error.
- The **ENXIO** error is marked as an extension.
- The APPLICATION USAGE section is removed.
- The description of **EINTR** is amended.

Issue 4, Version 2

The following changes are incorporated for X/OPEN UNIX conformance:

- The **writev()** function is added to the SYNOPSIS.
- The DESCRIPTION is updated to describe the reading of data from **STREAMS** files, an operational description of the **writev()** function is included, and a statement is added indicating that **SIGXFSZ** will be generated if an attempted write operation would cause the maximum file size to be exceeded.
- The RETURN VALUE section is updated to describe values returned by the **writev()** function.
- The ERRORS section has been restructured to describe errors that apply to both **write()** and writev() apart from those that apply to **writev()** specifically. The **EIO**, **ERANGE**, and **EINVAL** errors are also added.

HP-UX EXTENSIONS

DESCRIPTION

The **iovec** structure is defined in **/usr/include/sys/uio.h**.

For ordinary files, if the **O_DSYNC** file status flag is set, the write does not return until both the file data and the file attributes required to retrieve the data are physically updated. If the **O_SYNC** flag is set, the behavior is identical to that for **O_DSYNC**, with the addition that all file attributes changed by the write operation (including access time, modification time and status change time) are also physically updated prior to returning to the calling process.

For block special files, if the **O_DSYNC** or the **O_SYNC** flag 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.

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.

For character special devices, if the **stopio()** call was used on the same device after it was opened, **write()** returns **-1**, sets **errno** to [EBADF], and issues the **SIGHUP** signal to the process.

write() clears the **potential** and **granted** privilege vectors on the file.

If the write is performed by any user other than the owner or a user who has appropriate privileges, **write()** clears the set-user-ID, set-group-ID, and sticky bits on all nondirectory files. If the write is performed by the owner or a user who has appropriate privileges, the behavior is file-system dependent. In some file systems, the write clears the set-user-ID, set-group-ID, and sticky bits on a nondirectory file. In other file systems, the write does not clear these bits on a nondirectory file.

For directories, **write()** does not clear the set-user-ID, set-group-ID, and sticky bits.

W

ERRORS

If **write()** or **writev()** fails, the file offset remains unchanged and **errno** is set to one of the following values.

[EAGAIN] Enforcement-mode file and record locking was set, **O_NDELAY** was set, and there was a blocking record lock.

[EDEADLK] A resource deadlock would occur as a result of this operation (see *lockf*(2) and *fcntl*(2)).

[EDQUOT] User's disk quota block limit has been reached for this file system.

[EFBIG] The file is a regular file and *nbyte* is greater than zero and the starting position is greater than or equal to the offset maximum established in the open file description associated with *fildes*.

[ENOLCK] The system record lock table is full, preventing the write from sleeping until the blocking record lock is removed.

[ENOSPC] Not enough space on the file system. The process does not possess the **limit** effective privilege to override this restriction.

If **writev()** fails, the file offset remains unchanged and **errno** is set to one of the following values:

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

[EINVAL] One of the *iov_len* values in the *iov* array is negative.

If **write()** or **writev()** fails, the file offset is updated to reflect the amount of data transferred and **errno** is set to one of the following values.

[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()** attempts to write *mybufsize* bytes to the file from the buffer to which *mybuf* points.

```
#include <string.h>

int fildes;
size_t mybufsize;
ssize_t nbytes;
char *mybuf = "aeiou and sometimes y";
mybufsize = (size_t)strlen (mybuf);
nbytes = write (fildes, (void *)mybuf, mybufsize);
```

WARNINGS

Check *signal*(5) for the appropriateness of signal references on systems that support **sigvector**() (see *sigvector*(2)). **sigvector**() 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, the University of California, Berkeley, and SecureWare Inc.

SEE ALSO

mkfs(1M) creat(2), dup(2), fcntl(2), lockf(2), lseek(2), open(2), pipe(2), sigvector(2), ulimit(2), ustat(2), signal(5).

STANDARDS CONFORMANCE

write(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, POSIX.4

Section 4

File Formats

Section 4

File Formats

NAME

intro - introduction to file formats

DESCRIPTION

This section outlines the formats of various files. The C **struct** declarations for the file formats are given where applicable. Usually, these structures can be found in directories **/usr/include** or **/usr/include/sys**.

SEE ALSO

hier(5), Introduction(9).

a

NAME

a.out - assembler and link editor output

SYNOPSIS

#include <a.out.h> (for SOM files)

#include <elf.h> (for ELF files)

DESCRIPTION

PA32 SOM a.out

The file name **a.out** is the default file name for the output file from the assembler (see *as*(1)), compilers, and the linker (see *ld*(1)). The assembler and compilers create relocatable object files, ready for input to the linker. The linker creates executable object files and shared library files.

An object file consists of a file header, auxiliary headers, space dictionary, subspace dictionary, symbol table, relocation information, compiler records, space string table, symbol string table, and the data for initialized code and data. Not all of these sections are required for all object files. The file must begin with the file header, but the remaining sections do not have to be in any particular order; the file header contains pointers to each of the other sections of the file.

A relocatable object file, created by the assembler or compiler, must contain at least the following sections: file header, space dictionary, subspace dictionary, symbol table, relocation information, space string table, symbol string table, and code and data. It may also contain auxiliary headers and compiler records. Relocatable files generally contain unresolved symbols. The linker combines relocatable files and searches libraries to produce an executable file. The linker can also be used to combine relocatable files and produce a new relocatable file as output, suitable for input to a subsequent linker run.

An executable file, created by the linker, typically contains the following sections: file header, an HP-UX auxiliary header, space dictionary, subspace dictionary, symbol table, space string table, symbol string table, and code and data. The linker also copies any auxiliary headers and compiler records from the input files to the output file. If the file has been stripped (see *strip*(1)), it will not contain a symbol table, symbol string table, or compiler records. An executable file must not contain any unresolved symbols.

A shared library file, created by the linker, contains the same sections found in an executable file, with additional information added to the code section of the file. This additional information contains a header, export table, import table, and dynamic relocation records to be used by the dynamic loader.

Programs consist of two loadable spaces: a shared, non-writable, code space named **$TEXT$**; and a private, writable, data space named **$PRIVATE$**. A program may contain another loadable, private space named **$THREAD_SPECIFIC$**. A program may contain other unloadable spaces that contain data needed by development tools. For example, symbolic debugging information is contained in a space named **$DEBUG$** or **$PINFO$**. The linker treats loadable and unloadable spaces exactly the same, so the full generality of symbol resolution and relocation is available for the symbolic debugging information.

Spaces have an addressing range of 4,294,967,296 (2^32) bytes. Each loadable space is divided into four 1,073,741,824 (2^30) byte quadrants. The HP-UX operating system places all code in the first quadrant of the **$TEXT$** space, all data in the second quadrant of the **$PRIVATE$** space, and all shared library code in the third quadrant of shared memory space.

Each space is also divided into logical units called subspaces. When the linker combines relocatable object files, it groups all subspaces from the input files by name, then arranges the groups within the space by a sort key associated with each subspace. Subspaces are not architecturally significant; they merely provide a mechanism for combining individual parts of spaces independently from many input files. Some typical subspaces in a program are shown in the following table:

$SHLIB_INFO$	Information needed for dynamic loading
$MILLICODE$	Code for millicode routines
LIT	Sharable literals
$CODE$	Code
$UNWIND$	Stack unwind information
$GLOBAL$	Outer block declarations for Pascal
$DATA$	Static initialized data
$COMMON$	FORTRAN common
BSS	Uninitialized data
$TBSS$	Thread local storage

Subspaces can be initialized or uninitialized (although typically, only **BSS** and **$TBSS$** are uninitialized). The subspace dictionary entry for an initialized subspace contains a file pointer to the initialization data, while the entry for an uninitialized subspace contains only a 32-bit pattern used to initialize the entire area at load time.

In a relocatable file, initialized code and data often contain references to locations elsewhere in the file, and to unresolved symbols defined in other files. These references are patched at link time using the relocation information. Each entry in the relocation information (a "fixup") specifies a location within the initialized data for a subspace, and an expression that defines the actual value that should be placed at that location, relative to one or two symbols.

The linker summarizes the subspace dictionary in the HP-UX auxiliary header when creating an executable file. HP-UX programs contain only three separate sections: one for the code, one for initialized data, and one for uninitialized data. By convention, this auxiliary header is placed immediately following the file header.

When an **a.out** file is loaded into memory for execution, three areas of memory are set up: the **a.out** code is loaded into the first quadrant of a new, sharable space; the data (initialized followed by uninitialized) is loaded into the second quadrant of a new, private space; and a stack is created beginning at a fixed address near the middle of the second quadrant of the data space.

If the **a.out** file uses shared libraries, then the dynamic loader **/usr/lib/dld.sl** is loaded into memory and called to map into memory all shared libraries requested by the program. The shared library text is loaded into the third quadrant of the shared memory space, and the shared library data is allocated in the second quadrant of the data space.

The file format described here is a common format for all operating systems designed for HP's Precision Architecture. Therefore, there are some fields and structures that are not used on HP-UX or have been reserved for future use.

File Header

The format of the file header is described by the following structure declaration from **<filehdr.h>**.

```
struct header {
    short int    system_id;                /* system id */
    short int    a_magic;                  /* magic number */
    unsigned int version_id;               /* a.out format version */
    struct       sys_clock file_time;      /* timestamp */
    unsigned int entry_space; /* index of space containing entry point */
    unsigned int entry_subspace;           /* subspace index of entry */
    unsigned int entry_offset;             /* offset of entry point */
    unsigned int aux_header_location;      /* file ptr to aux hdrs */
    unsigned int aux_header_size;          /* sizeof aux hdrs */
    unsigned int som_length;               /* length of object module */
    unsigned int presumed_dp;   /* DP value assumed during compilation */
    unsigned int space_location;           /* file ptr to space dict */
    unsigned int space_total;              /* # of spaces */
    unsigned int subspace_location;        /* file ptr to subsp dict */
    unsigned int subspace_total;           /* # of subspaces */
    unsigned int loader_fixup_location;    /* space reference array */
    unsigned int loader_fixup_total;       /* # of space reference recs */
    unsigned int space_strings_location;   /* file ptr to sp. strings */
    unsigned int space_strings_size;       /* sizeof sp. strings */
    unsigned int init_array_location;      /* location of init pointers */
    unsigned int init_array_total;         /* # of init pointers */
    unsigned int compiler_location;        /* file ptr to comp recs */
    unsigned int compiler_total;           /* # of compiler recs */
    unsigned int symbol_location;          /* file ptr to sym table */
    unsigned int symbol_total;             /* # of symbols */
    unsigned int fixup_request_location;   /* file ptr to fixups */
    unsigned int fixup_request_total;      /* # of fixups */
    unsigned int symbol_strings_location;  /* file ptr to sym strings */
    unsigned int symbol_strings_size;      /* sizeof sym strings */
    unsigned int unloadable_sp_location;   /* file ptr to debug info */
    unsigned int unloadable_sp_size;       /* size of debug info */
```

```
    unsigned int checksum;                /* header checksum */
};
```

The timestamp is a two-word structure as shown below. If unused, both fields are zero.

```
struct sys_clock {
    unsigned int secs;
    unsigned int nanosecs;
};
```

a

Auxiliary Headers

The auxiliary headers are contained in a single contiguous area in the file, and are located by a pointer in the file header. Auxiliary headers are used for two purposes: to attach users' version and copyright strings to an object file, and to contain the information needed to load an executable program. In an executable program, the HP-UX auxiliary header must precede all other auxiliary headers. The following declarations are found in **<aouthdr.h>**.

```
struct aux_id {
    unsigned int mandatory : 1; /* linker must understand aux hdr info */
    unsigned int copy : 1;      /* copy aux hdr without modification */
    unsigned int append : 1;    /* merge multiple entries of same type */
    unsigned int ignore : 1;    /* ignore aux hdr if type unknown */
    unsigned int reserved : 12; /* reserved */
    unsigned int type : 16;     /* aux hdr type */
    unsigned int length;        /* sizeof rest of aux hdr */
};

/* Values for the aux_id.type field */
#define HPUX_AUX_ID            4
#define VERSION_AUX_ID         6
#define COPYRIGHT_AUX_ID       9
#define SHLIB_VERSION_AUX_ID  10

struct som_exec_auxhdr {         /* HP-UX auxiliary header */
    struct   aux_id som_auxhdr;  /* aux header id */
    long     exec_tsize;         /* text size */
    long     exec_tmem;          /* start address of text */
    long     exec_tfile;         /* file ptr to text */
    long     exec_dsize;         /* data size */
    long     exec_dmem;          /* start address of data */
    long     exec_dfile;         /* file ptr to data */
    long     exec_bsize;         /* bss size */
    long     exec_entry;         /* address of entry point */
    long     exec_flags;         /* loader flags */
    long     exec_bfill;         /* bss initialization value */
};

/* Values for exec_flags */
#define TRAP_NIL_PTRS    01

struct user_string_aux_hdr {       /* Version string auxiliary header */
    struct aux_id header_id;       /* aux header id */
    unsigned int  string_length;   /* strlen(user_string) */
    char          user_string[1];  /* user-defined string */
};

struct copyright_aux_hdr {       /* Copyright string auxiliary header */
    struct aux_id  header_id;      /* aux header id */
    unsigned int   string_length;  /* strlen(user_string) */
    char           copyright[1];   /* user-defined string */
};

struct shlib_version_aux_hdr {
    struct aux_id  header_id;      /* aux header id */
    short          version;        /* version number */
};
```

a

Space Dictionary

The space dictionary consists of a sequence of space records, as defined in **<spacehdr.h>**.

```
struct space_dictionary_record {
    union name_pt name;                 /* index to space name */
    unsigned int is_loadable: 1;        /* space is loadable */
    unsigned int is_defined: 1;         /* space is defined within file */
    unsigned int is_private: 1;         /* space is not sharable */
    unsigned int has_intermediate_code: 1;  /* contains intermediate
                                               code */
    unsigned int is_tspecific: 1;       /* space is $thread_specific$ */
    unsigned int reserved: 11;          /* reserved */
    unsigned int sort_key: 8;           /* sort key for space */
    unsigned int reserved2: 8;          /* reserved */
    int          space_number;          /* space index */
    int          subspace_index;        /* index to first subspace */
    unsigned int subspace_quantity;     /* # of subspaces in space */
    int          loader_fix_index;      /* index into loader fixup array */
    unsigned int loader_fix_quantity;     /* # of loader fixups in space */
    int          init_pointer_index;    /* index into init pointer array */
    unsigned int init_pointer_quantity;     /* # of init ptrs */
};
```

The strings for the space names are contained in the space strings table, which is located by a pointer in the file header. Each entry in the space strings table is preceded by a 4-byte integer that defines the length of the string, and is terminated by one to five null characters to pad the string out to a word boundary. Indices to this table are relative to the start of the table, and point to the first byte of the string (not the preceding length word). The union defined below is used for all such string pointers; the character pointer is defined for programs that read the string table into memory and wish to relocate in-memory copies of space records.

```
union name_pt {
    char         *n_name;
    unsigned int n_strx;
};
```

Subspace Dictionary

The subspace dictionary consists of a sequence of subspace records, as defined in **<scnhdr.h>**. Strings for subspace names are contained in the space strings table.

```
struct subspace_dictionary_record {
    int           space_index;            /* index into space dictionary */
    unsigned int  access_control_bits: 7;   /* access and priv levels
                                               of subsp */
    unsigned int  memory_resident: 1;     /* lock in memory during exec */
    unsigned int  dup_common: 1;      /* duplicate data symbols allowed */
    unsigned int  is_common: 1;           /* initialized common block */
    unsigned int  is_loadable: 1;         /* subspace is loadable */
    unsigned int  quadrant: 2;            /* quadrant in space subsp
                                             should reside in */
    unsigned int  initially_frozen: 1;    /* lock in memory
                                             when OS booted */
    unsigned int  is_first: 1;            /* must be first subspace */
    unsigned int  code_only: 1;           /* subspace contains only code */
    unsigned int  sort_key: 8;            /* subspace sort key */
    unsigned int  replicate_init: 1;      /* init values to be replicated
                                             to fill subsp len */
    unsigned int  continuation: 1;        /* subspace is a continuation */
    unsigned int  is_tspecific: 1;        /* subspace contains TLS */
    unsigned int  reserved: 5;            /* reserved */
    int           file_loc_init_value;    /* file location or init value */
    unsigned int  initialization_length;  /* length of initialization */
    unsigned int  subspace_start;         /* starting offset */
    unsigned int  subspace_length;        /* total subspace length */
```

```
    unsigned int  reserved2: 16;        /* reserved */
    unsigned int  alignment: 16;        /* alignment required */
    union name_pt name;                 /* index of subspace name */
    int           fixup_request_index;  /* index to first fixup */
    unsigned int  fixup_request_quantity;   /* # of fixup requests */
};
```

Symbol Table

The symbol table consists of a sequence of entries described by the structure shown below, from **<syms.h>**. Strings for symbol and qualifier names are contained in the symbol strings table, whose structure is identical with the space strings table.

```
struct symbol_dictionary_record {
    unsigned int   hidden: 1;          /* symbol not visible to loader */
    unsigned int   secondary_def: 1;   /* secondary def symbol */
    unsigned int   symbol_type: 6;     /* symbol type */
    unsigned int   symbol_scope: 4;    /* symbol value */
    unsigned int   check_level: 3;     /* type checking level */
    unsigned int   must_qualify: 1;    /* qualifier required */
    unsigned int   initially_frozen: 1;    /* lock in memory
                                              when OS booted */
    unsigned int   memory_resident: 1;  /* lock in memory during exec */
    unsigned int   is_common: 1;       /* common block */
    unsigned int   dup_common: 1;    /* duplicate data symbols allowed */
    unsigned int   xleast: 2;          /* MPE-only */
    unsigned int   arg_reloc: 10;      /* parameter relocation bits */
    union name_pt  name;               /* index to symbol name */
    union name_pt  qualifier_name;     /* index to qual name */
    unsigned int   symbol_info;        /* subspace index */
    unsigned int   symbol_value;       /* symbol value */
};

/* Values for symbol_type */
#define ST_NULL      0     /* unused symbol entry */
#define ST_ABSOLUTE  1     /* non-relocatable symbol */
#define ST_DATA      2     /* initialized data symbol */
#define ST_CODE      3     /* generic code symbol */
#define ST_PRI_PROG  4     /* program entry point */
#define ST_SEC_PROG  5     /* secondary prog entry point*/
#define ST_ENTRY     6     /* procedure entry point */
#define ST_STORAGE   7     /* storage request */
#define ST_STUB      8     /* MPE-only */
#define ST_MODULE    9     /* Pascal module name */
#define ST_SYM_EXT   10    /* symbol extension record */
#define ST_ARG_EXT   11    /* argument extension record */
#define ST_MILLICODE 12    /* millicode entry point */
#define ST_PLABEL    13    /* MPE-only */
#define ST_OCT_DIS   14    /* Used by OCT only--ptr to translated code */
#define ST_MILLI_EXT 15    /* address of external millicode */
#define ST_TSTORAGE  16    /* TLS common symbol */

/* Values for symbol_scope */
#define SS_UNSAT     0     /* unsatisfied reference */
#define SS_EXTERNAL  1     /* import request to external symbol */
#define SS_LOCAL     2     /* local symbol */
#define SS_UNIVERSAL 3     /* global symbol */
```

The meaning of the symbol value depends on the symbol type. For the code symbols (generic code, program entry points, procedure and millicode entry points), the low-order two bits of the symbol value encode the execution privilege level, which is not used on HP-UX, but is generally set to 3. The symbol value with those bits masked out is the address of the symbol (which is always a multiple of 4). For data symbols, the symbol value is simply the address of the symbol. For thread local storage symbols (not commons), the symbol value is the thread local storage offset in a library or executable file, and is the size of the symbol if in a relocatable object file. For storage requests and thread local storage commons, the symbol value is the number of bytes requested; the linker allocates space for the largest request for each symbol in the **BSS** or **$TBSS$** subspaces, unless a local or universal symbol is found for that symbol (in which case the storage request is treated like an unsatisfied reference).

a

If a relocatable file is compiled with parameter type checking, extension records follow symbols that define and reference procedure entry points and global variables. The first extension record, the **symbol extension record**, defines the type of the return value or global variable, and (if a procedure or function) the number of parameters and the types of the first three parameters. If more parameter type descriptors are needed, one or more **argument extension records** follow, each containing four more descriptors. A check level of 0 specifies no type checking; no extension records follow. A check level of 1 or more specifies checking of the return value or global variable type. A check level of 2 or more specifies checking of the number of parameters, and a check level of 3 specifies checking the types of each individual parameter. The linker performs the requested level of type checking between unsatisfied symbols and local or universal symbols as it resolves symbol references.

```
union arg_descriptor {
    struct {
    unsigned int    reserved: 3;    /* reserved */
    unsigned int    packing: 1;     /* packing algorithm used */
    unsigned int    alignment: 4;   /* byte alignment */
    unsigned int    mode: 4;        /* type of descriptor and its use */
    unsigned int    structure: 4;   /* structure of symbol */
    unsigned int    hash: 1;        /* set if arg_type is hashed */
    int             arg_type: 15;   /* data type */
    }  arg_desc;
    unsigned int    word;
};

struct symbol_extension_record {
    unsigned int    type: 8;                 /* always ST_SYM_EXT */
    unsigned int    max_num_args: 8;         /* max # of parameters */
    unsigned int    min_num_args: 8;         /* min # of parameters */
    unsigned int    num_args: 8;             /* actual # of parameters */
    union arg_descriptor symbol_desc;        /* symbol type desc. */
    union arg_descriptor argument_desc[3];   /* first 3 parameters */
};

struct argument_desc_array {
    unsigned int    type: 8;                 /* always ST_ARG_EXT */
    unsigned int    reserved: 24;            /* reserved */
    union arg_descriptor argument_desc[4];   /* next 4 parameters */
};
```

The **alignment** field in **arg_descriptor** indicates the minimum alignment of the data, where a value of **n** represents 2ˆ**n** byte alignment. The values for the **mode**, **structure**, and **arg_type** (when the data type is not hashed) fields in **arg_descriptor** are given in the following table.

a

Value	mode	structure	arg_type
0	any	any	any
1	value parm	scalar	void
2	reference parm	array	signed byte
3	value-result	struct	unsigned byte
4	name	pointer	signed short
5	variable	long ptr	unsigned short
6	function return	C string	signed long
7	procedure	Pascal string	unsigned long
8	long ref parm	procedure	signed dbl word
9		function	unsigned dbl word
10		label	short real
11			real
12			long real
13			short complex
14			complex
15			long complex
16			packed decimal
17			struct/array

For procedure entry points, the parameter relocation bits define the locations of the formal parameters and the return value. Normally, the first four words of the parameter list are passed in general registers (**r26-r23**) instead of on the stack, and the return value is returned in **r29**. Floating-point parameters in this range are passed instead in floating-point registers (**fr4-fr7**) and a floating-point value is returned in **fr4**. The parameter relocation bits consist of five pairs of bits that describe the first four words of the parameter list and the return value. The leftmost pair of bits describes the first parameter word, and the rightmost pair of bits describes the return value. The meanings of these bits are shown in the following table.

Bits	Meaning
00	No parameter or return value
01	Parameter or return value in general register
10	Parameter or return value in floating-point register
11	Double-precision floating-point value

For double-precision floating-point parameters, the odd-numbered parameter word should be marked **11** and the even-numbered parameter word should be marked **10**. Double-precision return values are simply marked **11**.

Every procedure call is tagged with a similar set of bits (see "Relocation Information" below), so that the linker can match each call with the expectations of the procedure entry point. If the call and entry point mismatch, the linker creates a stub that relocates the parameters and return value as appropriate.

Relocation Information

Each initialized subspace defines a range of fixups that apply to the data in that subspace. A fixup request is associated with every word that requires relocation or that contains a reference to an unsatisfied symbol. In relocatable object files created prior to HP-UX Release 3.0 on Series 800 systems, each fixup request is a five-word structure describing a code or data word to be patched at link time. Object files created on Release 3.0 or later contain variable-length fixup requests that describe every byte of the subspace. The *version_id* field in the file header distinguishes these two formats; the constant **VERSION_ID** is found in older object files, and the constant **NEW_VERSION_ID** is found in newer ones.

In older object files, fixups can compute an expression involving zero, one, or two symbols and a constant, then extract a field of bits from that result and deposit those bits in any of several different formats (corresponding to the Precision Architecture instruction set). The *fixup_request_index* field in the subspace dictionary entry indexes into the fixup request area defined by the file header and the *fixup_request_quantity* field refers to the number of fixup requests used for that subspace. The structure of a fixup request is contained in **<reloc.h>**.

```
struct fixup_request_record {
    unsigned int  need_data_ref: 1;    /* reserved */
    unsigned int  arg_reloc: 10;       /* parameter relocation bits */
    unsigned int  expression_type: 5;  /* how to compute value */
    unsigned int  exec_level: 2;       /* reserved */
    unsigned int  fixup_format: 6;     /* how to deposit bits */
```

```
    unsigned int  fixup_field: 8;     /* field to extract */
    unsigned int  subspace_offset;    /* subspace offset of word */
    unsigned int  symbol_index_one;   /* index of first symbol */
    unsigned int  symbol_index_two;   /* index of second symbol */
    int           fixup_constant;     /* constant */
};
/* Values for expression_type */
#define e_one     0   /* symbol1 + constant */
#define e_two     1   /* symbol1 - symbol2 + constant */
#define e_pcrel   2   /* symbol1 - pc + constant */
#define e_con     3   /* constant */
#define e_plabel  7   /* symbol1 + constant */
#define e_abs     18  /* absolute, 1st sym index is address */
/* Values for fixup_field (assembler mnemonics shown) */
#define e_fsel    0   /* F’: no change */
#define e_lssel   1   /* LS’: inverse of RS’ */
#define e_rssel   2   /* RS’: rightmost 11 bits, signed */
#define e_lsel    3   /* L’: leftmost 21 bits */
#define e_rsel    4   /* R’: rightmost 11 bits */
#define e_ldsel   5   /* LD’: inverse of RD’ */
#define e_rdsel   6   /* RD’: rightmost 11 bits, filled left with ones */
#define e_lrsel   7   /* LR’: L’ with "rounded" constant */
#define e_rrsel   8   /* RR’: R’ with "rounded" constant */
#define e_nsel    9   /* N1’: set all bits to zero: for id of 3-inst
                             code gen sequence */
/* Values for fixup_format (typical instructions shown) */
#define i_exp14   0   /* 14-bit immediate (LDW, STW) */
#define i_exp21   1   /* 21-bit immediate (LDIL, ADDIL) */
#define i_exp11   2   /* 11-bit immediate (ADDI, SUBI) */
#define i_rel17   3   /* 17-bit pc-relative (BL) */
#define i_rel12   4   /* 12 bit pc-relative (COMBT, COMBF, etc.) */
#define i_data    5   /* whole word */
#define i_none    6
#define i_abs17   7   /* 17-bit absolute (BE, BLE) */
#define i_milli   8   /* 17-bit millicode call (BLE) */
#define i_break   9   /* reserved (no effect on HP-UX) */
```

In newer object files, relocation entries consist of a stream of bytes. The *fixup_request_index* field in the subspace dictionary entry is a byte offset into the fixup dictionary defined by the file header, and the *fixup_request_quantity* field defines the length of the fixup request stream, in bytes, for that subspace. The first byte of each fixup request (the opcode) identifies the request and determines the length of the request.

In general, the fixup stream is a series of linker instructions that governs how the linker places data in the **a.out** file. Certain fixup requests cause the linker to copy one or more bytes from the input subspace to the output subspace without change, while others direct the linker to relocate words or resolve external references. Still others direct the linker to insert zeroes in the output subspace or to leave areas uninitialized without copying any data from the input subspace, and others describe points in the code without contributing any new data to the output file.

The include file **<reloc.h>** defines constants for each major opcode. Many fixup requests use a range of opcodes; only a constant for the beginning of the range is defined. The meaning of each fixup request is described below. The opcode ranges and parameters for each fixup are described in the table further below.

R_NO_RELOCATION Copy L bytes with no relocation.

R_ZEROES Insert L zero bytes into the output subspace.

R_UNINIT Skip L bytes in the output subspace.

R_RELOCATION Copy one data word with relocation. The word is assumed to contain a 32-bit pointer relative to its own subspace.

R_DATA_ONE_SYMBOL Copy one data word with relocation relative to an external symbol whose symbol index is S.

a

R_DATA_PLABEL Copy one data word as a 32-bit procedure label, referring to the symbol S. The original contents of the word should be 0 (no static link) or 2 (static link required).

R_SPACE_REF Copy one data word as a space reference. This fixup request is not currently supported.

R_REPEATED_INIT Copy L bytes from the input subspace, replicating the data to fill M bytes in the output subspace.

R_PCREL_CALL Copy one instruction word with relocation. The word is assumed to be a pc-relative procedure call instruction (for example, **BL**). The target procedure is identified by symbol S, and the parameter relocation bits are R.

R_ABS_CALL Copy one instruction word with relocation. The word is assumed to be an absolute procedure call instruction (for example, **BLE**). The target procedure is identified by symbol S, and the parameter relocation bits are R.

R_DP_RELATIVE Copy one instruction word with relocation. The word is assumed to be a dp-relative load or store instruction (for example, **ADDIL**, **LDW**, **STW**). The target symbol is identified by symbol S. The linker forms the difference between the value of the symbol S and the value of the symbol **$global$**. By convention, the value of **$global$** is always contained in register 27. Instructions may have a small constant in the displacement field of the instruction.

R_DLT_REL Copy one instruction word with relocation. The word is assumed to be a register-18-relative load or store instruction (for example, **LDW**, **LDO**, **STW**). The target symbol is identified by symbol S. The linker computes a linkage table offset relative to register 18 (reserved for a linkage table pointer in position-independent code) for the symbol S.

R_CODE_ONE_SYMBOL Copy one instruction word with relocation. The word is assumed to be an instruction referring to symbol S (for example, **LDIL**, **LDW**, **BE**). Instructions may have a small constant in the displacement field of the instruction.

R_MILLI_REL Copy one instruction word with relocation. The word is assumed to be a short millicode call instruction (for example, **BLE**). The linker forms the difference between the value of the target symbol S and the value of symbol 1 in the module's symbol table. By convention, the value of symbol 1 should have been previously loaded into the base register used in the BLE instruction. The instruction may have a small constant in the displacement field of the instruction.

R_CODE_PLABEL Copy one instruction word with relocation. The word is assumed to be part of a code sequence forming a procedure label (for example, **LDIL**, **LDO**), referring to symbol S. The **LDO** instruction should contain the value 0 (no static link) or 2 (static link required) in its displacement field.

R_BREAKPOINT Copy one instruction word conditionally. On HP-UX, the linker always replaces the word with a **NOP** instruction.

R_ENTRY Define a procedure entry point. The stack unwind bits, U, and the frame size, F, are recorded in a stack unwind descriptor.

R_ALT_ENTRY Define an alternate procedure entry point.

R_EXIT Define a procedure exit point.

R_BEGIN_TRY Define the beginning of a try/recover region.

R_END_TRY Define the end of a try/recover region. The offset R defines the distance in bytes from the end of the region to the beginning of the recover block.

R_BEGIN_BRTAB Define the beginning of a branch table.

R_END_BRTAB Define the end of a branch table.

R_AUX_UNWIND Define an auxiliary unwind table. **CN** is a symbol index of the symbol that labels the beginning of the compilation unit string table. **SN** is the offset, relative to the **CN** symbol, of the scope name string. **SK** is an integer specifying the scope kind.

a

R_STATEMENT Define the beginning of statement number N.

R_SEC_STATEMENT Define the beginning of a secondary statement number N.

R_DATA_EXPR Pop one word from the expression stack and copy one data word from the input subspace to the output subspace, adding the popped value to it.

R_CODE_EXPR Pop one word from the expression stack, and copy one instruction word from the input subspace to the output subspace, adding the popped value to the displacement field of the instruction.

R_FSEL Use an F´ field selector for the next fixup request instead of the default appropriate for the instruction.

R_LSEL Use an L-class field selector for the next fixup request instead of the default appropriate for the instruction. Depending on the current rounding mode, L´, LS´, LD´, or LR´ may be used.

R_RSEL Use an R-class field selector for the next fixup request instead of the default appropriate for the instruction. Depending on the current rounding mode, R´, RS´, RD´, or RR´ may be used.

R_N_MODE Select round-down mode (L´/R´). This is the default mode at the beginning of each subspace. This setting remains in effect until explicitly changed or until the end of the subspace.

R_S_MODE Select round-to-nearest-page mode (LS´/RS´). This setting remains in effect until explicitly changed or until the end of the subspace.

R_D_MODE Select round-up mode (LD´/RD´). This setting remains in effect until explicitly changed or until the end of the subspace.

R_R_MODE Select round-down-with-adjusted-constant mode (LR´/RR´). This setting remains in effect until explicitly changed or until the end of the subspace.

R_DATA_OVERRIDE Use the constant V for the next fixup request in place of the constant from the data word or instruction in the input subspace.

R_TRANSLATED Toggle "translated" mode. This fixup request is generated only by the linker during a relocatable link to indicate a subspace that was originally read from an old-format relocatable object file.

R_COMP1 Stack operations. The second byte of this fixup request contains a secondary opcode. In the descriptions below, A refers to the top of the stack and B refers to the next item on the stack. All items on the stack are considered signed 32-bit integers.

R_PUSH_PCON1	Push the (positive) constant V.
R_PUSH_DOT	Push the current virtual address.
R_MAX	Pop A and B, then push max(A, B).
R_MIN	Pop A and B, then push min(A, B).
R_ADD	Pop A and B, then push A + B.
R_SUB	Pop A and B, then push B – A.
R_MULT	Pop A and B, then push A * B.
R_DIV	Pop A and B, then push B / A.
R_MOD	Pop A and B, then push B % A.
R_AND	Pop A and B, then push A & B.
R_OR	Pop A and B, then push A | B.
R_XOR	Pop A and B, then push A XOR B.
R_NOT	Replace A with its complement.
R_LSHIFT	If C = 0, pop A and B, then push B << A. Otherwise, replace A with A << C.
R_ARITH_RSHIFT	If C = 0, pop A and B, then push B >> A. Otherwise, replace A with A >> C. The shifting is done with sign extension.
R_LOGIC_RSHIFT	If C = 0, pop A and B, then push B >> A. Otherwise, replace A with A >> C. The shifting is done with zero fill.

a

	R_PUSH_NCON1	Push the (negative) constant V.
R_COMP2	More stack operations.	
	R_PUSH_PCON2	Push the (positive) constant V.
	R_PUSH_SYM	Push the value of the symbol S.
	R_PUSH_PLABEL	Push the value of a procedure label for symbol S. The static link bit is L.
	R_PUSH_NCON2	Push the (negative) constant V.
R_COMP3	More stack operations.	
	R_PUSH_PROC	Push the value of the procedure entry point S. The parameter relocation bits are R.
	R_PUSH_CONST	Push the constant V.

R_PREV_FIXUP The linker keeps a queue of the last four unique multi-byte fixup requests. This is an abbreviation for a fixup request identical to one on the queue. The queue index X references one of the four; X = 0 refers to the most recent. As a side effect of this fixup request, the referenced fixup is moved to the front of the queue.

R_N0SEL Indicates that the following fixup is applied to the first of a three-instruction sequence to access data, generated by the compilers to enable the importing of shared library data.

R_N1SEL Uses a (N´) field selector for the next fixup request. This indicates that zero bits are to be used for the displacement on the instruction. This fixup is used to identify three-instruction sequences to access data (for importing shared library data).

R_LINETAB Defines the beginning of a line table. CU is a symbol index of the symbol that labels the beginning of the line table. SM is the offset relative to the CU symbol. ES designates the version information for the current line table.

R_LINETAB_ESC Defines an escape entry to be entered into the line table. ES designates the escape entry entered in the table. M designates the number of R_STATEMENT fixups to be interpreted as raw 8-bit table data.

R_LTP_OVERRIDE Override the following fixup, which is expected to be a R_DATA_ONE_SYMBOL fixup to copy one data word without relocation when building a shared library. The absolute byte offset of the symbol relative to the linkage table pointer is copied. If the linker is building a complete executable, the absolute virtual address is copied.

R_COMMENT Fixup used to pass comment information from the compiler to the linker. This fixup has a 5 byte argument that can be skipped and ignored by applications.

R_TP_OVERRIDE Override the next one of these fixups seen: R_DP_RELATIVE, R_DLT_REL, or R_DATA_ONE_SYMBOL, to use the thread local storage offset when fixing the instruction. This fixup is also used to catch thread local storage symbol mismatches.

R_RESERVED Fixups in this range are reserved for internal use by the compilers and linker.

The following table shows the mnemonic fixup request type and length and parameter information for each range of opcodes. In the parameters column, the symbol D refers to the difference between the opcode and the beginning of the range described by that table entry; the symbols B1, B2, B3, and B4 refer to the value of the next one, two, three, or four bytes of the fixup request, respectively.

Mnemonic	Opcodes	Length	Parameters
R_NO_RELOCATION	0-23	1	L = (D+1) * 4
	24-27	2	L = (D<<8 + B1 + 1) * 4
	28-30	3	L = (D<<16 + B2 + 1) * 4
	31	4	L = B3 + 1
R_ZEROES	32	2	L = (B1 + 1) * 4
	33	4	L = B3 + 1
R_UNINIT	34	2	L = (B1 + 1) * 4
	35	4	L = B3 + 1
R_RELOCATION	36	1	none
R_DATA_ONE_SYMBOL	37	2	S = B1
	38	4	S = B3
R_DATA_PLABEL	39	2	S = B1
	40	4	S = B3
R_SPACE_REF	41	1	none
R_REPEATED_INIT	42	2	L = 4; M = (B1 + 1) * 4
	43	3	L = (B1 + 1) * 4; M = (B1 + 1) * L
	44	5	L = (B1 + 1) * 4; M = (B3 + 1) * 4
	45	8	L = B3 + 1; M = B4 + 1
R_PCREL_CALL	48-57	2	R = rbits1(D); S = B1
	58-59	3	R = rbits2(D<<8 + B1); S = B1
	60-61	5	R = rbits2(D<<8 + B1); S = B3
R_ABS_CALL	64-73	2	R = rbits1(D); S = B1
	74-75	3	R = rbits2(D<<8 + B1); S = B1
	76-77	5	R = rbits2(D<<8 + B1); S = B3
R_DP_RELATIVE	80-111	1	S = D
	112	2	S = B1
	113	4	S = B3
R_DLT_REL	120	2	S = B1
	121	4	S = B3
R_CODE_ONE_SYMBOL	128-159	1	S = D
	160	2	S = B1
	161	4	S = B3
R_MILLI_REL	174	2	S = B1
	175	4	S = B3
R_CODE_PLABEL	176	2	S = B1
	177	4	S = B3
R_BREAKPOINT	178	1	none
R_ENTRY	179	9	U,F = B8 (U is 37 bits; F is 27 bits)
	180	6	U = B5 >> 3; F = pop A
R_ALT_ENTRY	181	1	none
R_EXIT	182	1	none
R_BEGIN_TRY	183	1	none
R_END_TRY	184	1	R = 0
	185	2	R = sign_extend(B1) * 4
	186	4	R = sign_extend(B3) * 4
R_BEGIN_BRTAB	187	1	none
R_END_BRTAB	188	1	none
R_STATEMENT	189	2	N = B1
	190	3	N = B2
	191	4	N = B3
R_DATA_EXPR	192	1	none
R_CODE_EXPR	193	1	none
R_FSEL	194	1	none
R_LSEL	195	1	none
R_RSEL	196	1	none
R_N_MODE	197	1	none
R_S_MODE	198	1	none
R_D_MODE	199	1	none
R_R_MODE	200	1	none
R_DATA_OVERRIDE	201	1	V = 0
	202	2	V = sign_extend(B1)

	203	3	**V = sign_extend(B2)**
	204	4	**V = sign_extend(B3)**
	205	5	**V = B4**
R_TRANSLATED	206	1	none
R_AUX_UNWIND	207	12	**CU,SN,SK = B11** (CU is 24 bits; SN is 32 bits; SK is 32 bits)
R_COMP1	208	2	**OP = B1; V = OP & 0x3f; C = OP & 0x1f**
R_COMP2	209	5	**OP = B1; S = B3; L = OP & 1;** **V = ((OP & 0x7f) << 24) | S**
R_COMP3	210	6	**OP = B1; V = B4;** **R = ((OP & 1) << 8) | (V >> 16);** **S = V & 0xffffff**
R_PREV_FIXUP	211-214	1	**X = D**
R_N0SEL	216	1	none
R_N1SEL	217	1	none
R_SEC_STMT	215	1	none
R_LINETAB	218	9	**ES = B1; CU = B3; SM = B4**
R_LINETAB_ESC	219	3	**ES = B1; M = B1**
R_LTP_OVERRIDE	220	1	none
R_COMMENT	221	6	**OP = B1; V = B2 to B6**
R_TP_OVERRIDE	222	1	none
R_RESERVED	224-255		reserved

Parameter relocation bits are encoded in the fixup requests in two ways, noted as *rbits1* and *rbits2* in the above table.

The first encoding recognizes that the most common procedure calls have only general register arguments with no holes in the parameter list. The encoding for such calls is simply the number of parameters in general registers (0 to 4), plus 5 if there is a return value in a general register.

The second encoding is more complex. The 10 argument relocation bits are compressed into 9 bits by eliminating some impossible combinations. The encoding is the combination of three contributions. The first contribution is the pair of bits for the return value, which are not modified. The second contribution is 9 if the first two parameter words together form a double-precision parameter; otherwise, it is 3 times the pair of bits for the first word plus the pair of bits for the second word. Similarly, the third contribution is formed based on the third and fourth parameter words. The second contribution is multiplied by 40, the third is multiplied by 4, then the three are added together.

Compiler Records

Compiler records are placed in relocatable files by each compiler or assembler to identify the version of the compiler that was used to produce the file. These records are copied into the executable file by the linker, but are strippable. The structure of a compiler record is shown below. All strings are contained in the symbol string table.

The format of the compilation record is described by the following structure declaration from **<compunit.h>**.

```
struct compilation_unit {
    union name_pt     name;           /* entry name */
    union name_pt     language_name;  /* language used */
    union name_pt     product_id;     /* compiler ID */
    union name_pt     version_id;     /* compiler version */
    unsigned int      reserved: 31;   /* reserved */
    unsigned int      chunk_flag: 1;  /* MPE-only */
    struct sys_clock  compile_time;   /* time file was compiled */
    struct sys_clock  source_time;    /* time file was last modified */
};
```

PA64 ELF a.out

The file name **a.out** is the default output file name from the link editor, *ld*(1). The link editor will make an **a.out** executable if there were no errors in linking. The output file of the assembler, *as*(1), also follows the format of the **a.out** file although its default file name is different.

Programs that manipulate ELF files may use the library that *elf*(3E) describes. An overview of the file format follows. For more complete information, see the references given below.

```
         Linking View                    Execution View
 _________________________       _________________________
          ELF header                      ELF header
 |_______________________|       |_______________________|
 |Program header table   |       |  Program header table |
 |       optional        |       |                       |
 |_______________________|       |  _____________________|
 |      Section 1        |       |                       |
 |_______________________|       |                       |
 |         . . .         |       |      Segment 1        |
 |                       |       |                       |
 |_______________________|       |  _____________________|
 |      Section n        |       |                       |
 |_______________________|       |                       |
 |         . . .         |       |      Segment 2        |
 |                       |       |                       |
 |_______________________|       |  _____________________|
 |         . . .         |       |         . . .         |
 |_______________________|       |  _____________________|
 |  Section header table |       |  Section header table |
 |                       |       |       optional        |
 |_______________________|       |  _____________________|
```

An ELF header resides at the beginning and holds a "road map" describing the file's organization. Sections hold the bulk of object file information for the linking view: instructions, data, symbol table, relocation information, and so on. Segments hold the object file information for the program execution view. As shown, a segment may contain one or more sections.

A program header table, if present, tells the system how to create a process image. Files used to build a process image (execute a program) must have a program header table; relocatable files do not need one. A section header table contains information describing the file's sections. Every section has an entry in the table; each entry gives information such as the section name, the section size, and so on. Files used during linking must have a section header table; other object files may or may not have one.

Although the figure shows the program header table immediately after the ELF header, and the section header table following the sections, actual files may differ. Moreover, sections and segments have no specified order. Only the ELF header has a fixed position in the file.

When an **a.out** file is loaded into memory for execution, three logical segments are set up: the text segment, the data segment (initialized data followed by uninitialized, the latter actually being initialized to all 0's), and a stack. The text segment is not writable by the program; if other processes are executing the same **a.out** file, the processes will share a single text segment.

The data segment starts at the next maximal page boundary past the last text address. (If the system supports more than one page size, the "maximal page" is the largest supported size.) When the process image is created, the part of the file holding the end of text and the beginning of data may appear twice. The duplicated chunk of text that appears at the beginning of data is never executed; it is duplicated so that the operating system may bring in pieces of the file in multiples of the actual page size without having to realign the beginning of the data section to a page boundary. Therefore, the first data address is the sum of the next maximal page boundary past the end of text plus the remainder of the last text address divided by the maximal page size. If the last text address is a multiple of the maximal page size, no duplication is necessary. The stack is automatically extended as required. The data segment is extended as requested by the *brk*(2) system call.

FILES

<a.out.h>
<aouthdr.h>
<compunit.h>
<filehdr.h>
<reloc.h>
<scnhdr.h>
<spacehdr.h>
<syms.h>

a

SEE ALSO

System Tools

as(1)	Translate assembly code to machine code
cc(1)	Invoke the HP-UX C compiler
ld(1)	Invoke the link editor

Miscellaneous

crt0(3)	Execution startup routine
elf(3E)	For the ELF a.out only
end(3C)	Symbol of the last locations in program
magic(4)	Magic number for HP-UX implementations
nm(1)	Print name list of object file
strip(1)	Strip symbol and line number information from an object file

a

NAME

acct - per-process accounting file format

SYNOPSIS

#include <sys/acct.h>

DESCRIPTION

Files produced as a result of calling **acct()** (see *acct*(2)) have records in the form defined by **<sys/acct.h>**, whose contents are:

```
typedef ushort comp_t; /* "floating point":
                                     13-bit fraction, 3-bit exponent */
struct  acct {
    char    ac_flag;        /* Accounting flag */
    char    ac_stat;        /* Exit status */
    uid_t   ac_uid;         /* Accounting user ID */
    gid_t   ac_gid;         /* Accounting group ID */
    dev_t   ac_tty;         /* control typewriter */
    time_t  ac_btime;       /* Beginning time */
    comp_t  ac_utime;       /* acctng user time in clock ticks */
    comp_t  ac_stime;       /* acctng system time in clock ticks */
    comp_t  ac_etime;       /* acctng elapsed time in clock ticks */
    comp_t  ac_mem;         /* memory usage in clicks */
    comp_t  ac_io;          /* chars trnsfrd by read/write */
    comp_t  ac_rw;          /* number of block reads/writes */
    char    ac_comm[8];     /* command name */
};
#define AFORK   01          /* has executed fork, but no exec */
#define ASU     02          /* used super-user privileges */
#define ACCTF   0300        /* record type: 00 = acct */
```

In **ac_flag**, the **AFORK** flag is turned on by each **fork()** and turned off by an **exec()** (see *fork*(2) and *exec*(2)). The **ac_comm** field is inherited from the parent process and is reset by any **exec()**. Each time the system charges the process with a clock tick, it also adds to **ac_mem** the current process size, computed as follows:

(data size) + (text size) + (number of in-core processes sharing text) +
sum of ((shared memory segment size) / (number of in-core processes attached to segment))

For systems with virtual memory, the text, data, and shared memory sizes refer to the resident portion of the memory segments. The value of **ac_mem**/(**ac_stime**+**ac_utime**) can be viewed as an approximation to the mean process size, as modified by text-sharing.

The **tacct** structure, which resides with the source files of the accounting commands, represents the total accounting format used by the various accounting commands:

```
/*
* total accounting (for acct period), also for day
*/
struct tacct {
    uid_t           ta_uid;       /* userid */
    char            ta_name[8];   /* login name */
    float           ta_cpu[2];    /* cum. cpu time, p/np (mins) */
    float           ta_kcore[2];  /* cum kcore-minutes, p/np */
    float           ta_con[2];    /* cum. connect time, p/np, mins */
    float           ta_du;        /* cum. disk usage */
    long            ta_pc;        /* count of processes */
    unsigned short  ta_sc;        /* count of login sessions */
    unsigned short  ta_dc;        /* count of disk samples */
    short           ta_fee;       /* fee for special services */
};
```

WARNINGS

The **ac_mem** value for a short-lived command gives little information about the actual size of the command because **ac_mem** can be incremented while a different command (such as the shell) is being

a

executed by the process.

Kernel internal structures may change from release to release without warning. Applications directly relying on these structures are not supported.

SEE ALSO

acct(2), acct(1M), acctcom(1M), exec(2), fork(2).

STANDARDS CONFORMANCE

acct: SVID2, SVID3, XPG2

a

NAME

ar - common archive file format

SYNOPSIS

#include <ar.h>

DESCRIPTION

The **ar** command is used to concatenate several files into an archive file (see *ar*(1)). Archives are used mainly as libraries to be searched by the link editor (see *ld*(1)).

Each archive begins with the archive magic string.

```
#define   ARMAG    "!<arch>\n"       /* magic string */
#define   SARMAG   8                 /* length of magic string */
```

Following the archive magic string are the archive file members. Each file member is preceded by a file member header which is of the following format:

```
#define   ARFMAG        "‘\n"    /* header trailer string */
#define   AR_NAME_LEN   16       /* ar_name size, includes ‘/’ */

struct ar_hdr   /* archive file member header - printable ascii */
{
    char    ar_name[16];    /* file member name - ‘/’ terminated */
    char    ar_date[12];    /* file member date - decimal */
    char    ar_uid[6];      /* file member user id - decimal */
    char    ar_gid[6];      /* file member group id - decimal */
    char    ar_mode[8];     /* file member mode - octal */
    char    ar_size[10];    /* file member size - decimal */
    char    ar_fmag[2];     /* ARFMAG - string to end header */
};
```

All information in the file member headers is in printable ASCII. The numeric information contained in the headers is stored as decimal numbers (except for **ar_mode** which is in octal). Thus, if the archive contains printable files, the archive itself is printable.

The contents of the **ar_name** field are slash (**/**) terminated and blank-padded. The **ar_date** field is the modification date of the file at the time of its insertion into the archive. Common format archives can be moved from system to system as long as the portable archive command **ar** is used. Note that older versions of **ar** did not use the common archive format, and those archives cannot be read or written by the common archiver.

Each archive file member begins on an even byte boundary; a new-line character is inserted between files if necessary. Nevertheless, the size given reflects the actual size of the file exclusive of padding.

Notice there is no provision for empty areas in an archive file. If the archive symbol table exists, the first file in the archive has a zero-length name (i.e., **ar_name[0] == ’/’** and **ar_name[1] == ’ ’**). The contents of this archive member are machine-dependent. Refer to the appropriate *a.out*(4) manual entry for more information.

Each archive which contains object files (see *a.out*(4)) may include an archive symbol table. This symbol table is used by the link editor (see *ld*(1)) to determine which archive members must be loaded during the link edit process. The archive symbol table (if it exists) is always the first member in the archive (but is never listed) and is automatically created and/or updated by **ar**.

If a member with a file name greater than 15 bytes exists within the archive, then the archive will also contain an additional special member to store the long file name string table. The special string table member has a zero length name where **ar_name[0] == ’/’** and **ar_name[1] == ’/’**.

If a special string table exists, it will precede all non-special archive members. If both a symbol table member and a string table member exist then the symbol table member will always precede the string table member.

Each entry in the string table is followed by a slash and a new-line character. The offset of the table begins at zero. If an archive member name exceeds 15 bytes, then the **ar_name** entry in the member's header does not contain a name, instead it contains the offset into the string table preceded by a slash.

For example, the member name **thisverylongfilename.o** contains **/0** in the **ar_name** field. This value represents the offset into the string table. The member name

a

yetanotherlongfilename.o contains **/27** in the **ar_name** field. The long name string table would have the following format:

	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	t	h	i	s	i	s	a	v	e	r
10	y	l	o	n	g	f	i	l	e	n
20	a	m	e	.	o	/	\n	y	e	t
30	a	n	o	t	h	e	r	l	o	n
40	g	f	i	l	e	n	a	m	e	.
50	o	/	\n							

SEE ALSO

System Tools:

ar(1) create archived libraries
ld(1) invoke the link editor

Miscellaneous:

a.out(4) assembler, compiler, and linker output
magic(4) magic number for HP-UX implementations
ranlib(1) regenerate an archive symbol table
strip(1) strip symbol and line number information from an object file

CAVEATS

strip removes the archive symbol table member from the archive (see *strip*(1)). The archive symbol table must be restored by using the **-ts** option of the **ar** command or the *ranlib*(1) command before the archive can be used with the **ld** link editor.

a

NAME

arraytab - disk array configuration table

DESCRIPTION

Arraytab is a table of supported configurations for HP SCSI disk array products. Each table entry includes a set of parameter values that specify an array configuration. The array configuration table is located in **/etc/hpC2400/arraytab**.

HP SCSI disk array devices are highly configurable. The physical disk mechanisms in an array can be grouped in special ways to provide various levels of data redundancy, and data read/write performance. These levels are known as **RAID** (for *R*edundant *A*rray of *I*nexpensive *D*isks) levels.

Using a process called **striping**, data from each read or write operation can be distributed across multiple physical disk mechanisms to provide load balancing and/or to add data redundancy for protection against the failure of physical disk mechanisms. Striping is done in increments of the physical disk block size for all RAID levels except RAID_3 (which uses byte striping). The stripe size, also known as **segment size**, establishes the degree of data spread across the set of disk mechanisms.

Logical disks are created by defining address regions that include all or part of the address space of a disk group. Each logical disk are separately addressable. For example:

Physical Block Address	Physical Drive 1	2	3		
0	X	X	X	|	
	X	X	X	|	Logical Drive 0
	X	X	X	|	
.	Y	Y	Y	|	
.	Y	Y	Y	|	Logical Drive 1
.	Y	Y	Y	|	
	Z	Z	Z	|	
	Z	Z	Z	|	Logical Drive 2
N	Z	Z	Z	|	

In this example, 3 physical drives have been grouped into a single RAID group (1 vertical partition). Three logical disks have then been formed by partitioning the composite logical address space (in blocks) into 3 logical regions.

A logical configuration which has more than one logical partition per physical disk group is called a **sub-LUN**. If the logical partition includes the entire address space of the disk group, the logical partition is called a **regular LUN**.

Each array configuration requires two types of specifications—physical specifications, and logical specifications. A physical specification determines which disk mechanisms form the groups. A logical configuration specifies the type and location of each physical disk mechanism (in the array) that is to be used within the logical partition. The logical configuration also specifies the size and characteristics of the logical partition.

Raid Levels

The disk array can be configured using one of the following RAID levels, depending on the I/O requirements of the system, and the degree of data availability required. Data availability (redundancy) is achieved at the expense of storage capacity, and possibly performance.

RAID_0: This level provides no data redundancy, however disks may be grouped in a set, and data striped across the disk set to provide load balancing.

A special case exists when a drive group of size 1 is defined (independent mode). In this case the physical disk mechanisms appear to the system as they would if there were no array controller. The array controller is transparent, providing only address selection among the disks connected to it. When configured in this manner the disks operate independently for every I/O request.

RAID_1: This level provides disk mirroring. Two sets of disks maintain identical copies of the data. By choosing the number of disks in each set larger than one, data can be striped across the disks in each set (RAID_0) to provide better load balancing; the redundant disk sets provide availability.

a

RAID_3: This level uses byte striping across a set of *n* drives, with an additional drive maintaining an XOR parity check byte for each byte of data. The resulting logical disk sector size is *n* times the sector size of one disk. Data can be recovered, if a drive fails, by using the redundancy of the parity drive while operating in a "degraded" mode. Since reads and writes to the individual mechanisms are accomplished in parallel, long I/O requests to the array complete in 1/*n*th the time, exclusive of the access time, allowing higher bandwidth I/O rates. Because the mechanisms operate in concert during the input/output operation, only one I/O may process at a time. Disks configured in RAID_3 have access time characteristics of a single disk, but are capable of transferring data at higher rates. This mode is most useful with long I/O requests.

RAID_5: This level uses block striping across a set of *n* drives. XOR parity information is maintained across the set of the drives on a block basis, such that the failure of any one drive allows continued operation in a "degraded" mode. While degraded, data from the failed drive is reconstructed from the parity information, and the data on the remaining disks. Unlike RAID_3, block sizes can be the same as for a single disk; however, write performance suffers when write requests are less than *n* blocks, because read-modify-write operations must be done on the data drive, and the parity drive. Because the XOR parity data is maintained on a block basis, drive mechanisms can operate independently, allowing multiple I/O requests to process concurrently on the set of disks. This mode is most useful for short I/O requests. This mode allows parallel processing of I/Os requests across the set of disks, however data transfer rates are equivalent to those of a single disk.

CONFIGURATION TABLE

Entries in the configuration table are formed from a number of fields, each terminated by a ":" character. The fields are organized as shown below:

Drive Group Name (Physical Configuration Name)
Drive List

. . .

Drive List

Logical Configuration Name
Logical partition configuration

Logical partition configuration

. . .

Logical partition configuration

Each part of the specification is terminated by a 'New Line' character. The fields are generally composed of an identifier token, followed by parameter value or values, separated by "**#**". Comments may also be placed within the file by leading the field with "**#**". All following characters up to 'New Line' will be ignored. A character may be escaped by immediately preceding it with "****". Logical configurations and physical configurations may appear in any order, provided the syntax requirements are met. Physical disk configuration labels must be unique within the table. Logical configuration labels need not be unique. However, configurations with non-unique labels should have different parameter values for the array controller type field, or specify a different disk group. Logical disk configurations are searched sequentially— the first labeled specification which matches will be used. The following list describes the arraytable parameters and their use.

Name *Type* **Description**

ct *str* Array Controller Type. This parameter must be specified in at least one logical partition of a logical configuration. The field consists of the concatenated vendor ID and product ID strings which are returned by the SCSI Inquiry message to the array controller, with "_" separating these two strings. This field defines array product for which this configuration may be used. For example, **HP_C2425D** or **HP_C2430D**. **dl** *num* Physical Drive list. Each drive group consists of 1 or more lists of disk mechanisms, each specified by the array channel number, the channel ID of the disk mechanism on the channel, and a disk identifier label, respectively. A drive list may have up to 5 drives listed. The order of the drives in the list determines the order in which data is placed on the drives. This order is defined by the drive sequence label **d***N*, where *N* is a number from 0 to 4. Subsequent lists may be used to create drive groups larger than 5 disks. The disk identifier label is a string formed from the vendor ID and product ID strings returned from a SCSI Inquiry message, separated by "_". Certain constraints are made for the drive groups and drive lists, depending upon the number of drives and the RAID level chosen. See restrictions below.

a

lp *num* Logical partition within the logical configuration. A logical configuration will have one or more logical partitions, with each logical partition consisting of a portion or the whole of a drive group (See LUN type). Address space is allocated to each logical partition in the order in which it is found in the table, and begin start from the beginning block of the disk group. A logical partition number corresponds with the SCSI logical unit (LUN) number.

lt *str* Logical partition or LUN type. A logical partition may be either "regular LUN" (**reg**) or "sub-LUN" (**sub**). A sub-LUN allows configuring multiple logical disks for a group of disks, each to an arbitrary capacity. A regular LUN allows a logical disk capacity of the composite disk capacity of a group of drives, or 2 GByte, whichever is smaller. When the regular LUN option is used, the capacity parameter is ignored by the array controller. Additional logical drives may be configured to use the remaining capacity beyond 2 GByte if the regular LUN mode is chosen.

bs *num* Block size of the logical partition or LUN in bytes. This value must be specified in increments of the native disk mechanism sector size. Currently supported values are 512, 1024, 2048, 4096 bytes.

cv *num* Capacity of the logical partition or LUN in blocks. If this value is set to 0, the array will configure as many blocks as are available (not previously configured in another LUN).

ss *num* Segment size. The size in bytes of a contiguous segment of the logical address space which will reside on a single physical disk. This allows controlling how many disks are involved with a single I/O request. If I/O requests are mostly random, single block requests, this value should be set to the block size. If the I/O requests are typically more than a single sequential block, then this value should be set to the number of bytes which minimizes the number of disks necessary to service most I/Os. The value must be an integral number of the block size.

is *num* The size in bytes of the first segment of the LUN. This allows this area to be set to a size different than the remainder of the disk, an area typically used as the boot block for some systems. This must be an integral number of the block size. If there are no special requirements, this parameter should be set to 0.

rl *str* RAID level. Acceptable strings are { **RAID_0**, **RAID_1**, **RAID_3**, **RAID_5**}. The RAID modes are described above.

gn *str* Group name. This is the label used to identify the physical drive group or configuration to be used with the logical configuration.

gs *num* Number of physical drives in the drive group.

rs *num* Reconstruction size. This is the number of logical disk blocks which will be reconstructed in one operation when a drive data set is being repaired. A larger value will cause the reconstruction to complete more quickly (and efficiently), but will cause longer delays in processing other I/O requests.

rf *num* Reconstruction frequency. This is the period of time between reconstruction operations, specified in 0.1 Sec. (see Reconstruction Size). This parameter is useful in systems which do not do I/O request queuing to allow I/Os to process smoothly while reconstructing the data set.

lf *num* LUN configuration flags. There are 16 possible LUN configuration flags. Currently only 6 of these flags are defined. It is not recommended that these fields be altered. The flags are used to enable certain features of the array controller for the specified LUN. The flags may be set by specifying the hexadecimal value for all the flags. The flags are defined as follows:

Bit 0	*off*	Not used.
Bit 1	*on*	Automatic reconstruction disable. Enabled allows the array controller to automatically begin data restruction when the replacement of a failed disk is detected.
Bit 2	*off*	Not used.

Bit 3	*off*	Not used.
Bit 4	*on*	Asynchronous Event Notification polling enable.
Bit 5	*on*	Parity verification enable.
Bit 6	*on*	Write with parity verification enable.
Bit 7	*off*	Not used.
Bit 8	*off*	Mode Sense: Current. Current values are accessed during mode sense. This bit should not be set concurrently with Bit 9.
Bit 9	*off*	Mode Sense: Saved. Saved values are accessed during mode sense. This bit should not be set concurrently with Bit 10.
Bit 10-15	*off*	Not used.

RAID LEVEL RESTRICTIONS:

The following restrictions apply to RAID configurations for the array:

RAID_0:

- No disk list may contain more than 1 disk per channel
- For groups larger than 5 disks, additional lists are defined and data is accessed in the order of definition.

RAID_1:

In this mode the lists define the set of disks for data, and the set of disks which form the mirrored pair.

- Two lists must be specified.
- The two lists must be of equal length.
- No list may contain more than 1 disk per channel
- Corresponding entries in the two lists (these form a mirrored disk pair) cannot be on the same channel.

RAID_3:

- There must be an odd number of disks in the disk list.
- Disks in the disk list must be on separate channels.
- The first disk of the set must be on channel 1, followed in order by the other channels. Thus a 3 disk set will use channels 1 through 3.
- The disk on the last channel is the parity disk. (Channel 3 for 3 disk configuration, channel 5 for 5 disk configuration.)
- Maximum configuration is 1 list of 5 disks.

RAID_5:

- The disk list cannot contain more than 1 disk per channel.
- Maximum configuration is 1 list of 5 disks.

EXAMPLE:

```
PGroup1: dl#0: d0#1#0#HP_02425: d1#2#0#HP_02425: d2#3#0#HP_02425:

LConfig: lp#0: gs#3: gn#PGroup1: r#RAID_3: is#0: ss#8192:\
         cv#204994: ct#HP_C2425D

         lp#1: gs#3: gn#PGroup1: r#RAID_3: is#0: ss#8192:\
         cv#8192: ct:#HP_C2425D
```

FILE SYSTEM CONSIDERATIONS:

The performance of the disk array will depend heavily upon the RAID level used, and the application. In addition, the disk array configuration parameters should be chosen with consideration of the parameters used for the file system in use on the array.

WARNING:

The configurations found in **/etc/hpC2400/arraytab** have been chosen and certified by HP for proper operation on HP systems. Use of configurations other than these have NOT been certified for proper operation, and cannot be warranted.

For configurations using logical partitions exceeding 2 GB it is necessary that the 2 GB governor flag be turned off in the array controller. See *see*(1M).

a

DEPENDENCIES:

Series 700:

LUN address 6 and 7 are reserved for use with array management utilities, and should not be configured.

Series 800:

LUN address 6 and 7 are reserved for use with array management utilities, and should not be configured.

Only RAID levels 0 (Independent), 3, and 5 are supported.

RAID 0 configurations must span only a single disk (Independent mode) and result in separate addressable logical partitions, one for each physical disk.

RAID 3 and RAID 5 configurations must result in a single logical partition, which span all disks on the array.

AUTHOR:

arraytab was developed by HP.

FILES

/etc/hpC2400/arraytab

SEE ALSO

newarray(1M), mkfs(1M), buildfs(1M), cfl(1M), fs(4), see(1M).

a

NAME

audeventstab - define and describe audit system events

DESCRIPTION

The **/usr/audit/audeventstab** file lists audit event numbers, corresponding mnemonic names, and brief explanations of each event. Blank lines and comments (beginning with a **#** character) are allowed. Each non-comment, non-blank line in this file contains three parts:

event	Audit event number in decimal: a single field separated by whitespace.
name	Corresponding mnemonic name: a single field separated by whitespace.
explanation	Remainder of the line, following a **#** character.

For kernel-generated audit events, event numbers match kernel-internal system call numbers, and event names are system call names. For events from self-auditing programs, names are macros defined in <**sys/audit.h**>.

EXAMPLES

To extract a list of event numbers and names from the file by stripping comments and ignoring blank lines:

```
tab='	'
sed < /usr/audit/audeventstab -e 's/#.*//' -e "/^[ $tab]*$/d"
```

AUTHOR

audeventstab was developed by HP.

FILES

/usr/audit/audeventstab

SEE ALSO

audisp(1M), audevent(1M).

NAME

audit - file format and other information for auditing

SYNOPSIS

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

DESCRIPTION

Audit records are generated when users make security-relevant system calls, as well as by self-auditing processes that call **audwrite()** (see *audwrite*(2)). Access to the auditing system is restricted to superuser.

Each audit record consists of an audit record header and a record body. The record header is comprised of time, process ID, error, event type, and record body length. The time refers to the time the audited event completes in either success or failure; the process ID belongs to the process being audited; the event type is a field identifying the type of audited activity; the length is the record body length expressed in bytes. The exact format of the header is defined in <**sys/audit.h**> as follows:

```
struct audit_hdr {
    u_long ah_time;        /* date/time (tv_sec of timeeval) */
    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 record body is the variable-length component of an audit record containing more information about the audited activity. For records generated by system calls, the body contains the parameters of the system calls; for records generated by self-auditing processes, the body consists of a high-level description of the event (see *audwrite*(2)).

The records in the audit file are compressed to save file space. When a process is audited the first time, a *pid* identification record (PIR) is written into the audit file containing information that remains constant throughout the lifetime of the process. This includes the parent's process ID, audit ID, real user ID, real group ID, effective user ID, effective group ID, and the terminal ID (tty). The PIR is entered only once per process per audit file, and is also defined in <**sys/audit.h**> as follows:

```
struct pir_body {          /* pir-related info */
    short   ppid;          /* parent process ID */
    int32_t aid;           /* audit ID */
    uid_t   ruid;          /* user_ID */
    gid_t   rgid;          /* group ID */
    uid_t   euid;          /* effective user_ID */
    gid_t   egid;          /* effective group_ID */
    dev_t   tty;           /* tty number */
};
```

Information accumulated in an audit file is analyzed and displayed by **audisp** (see *audisp*(1M)).

Whenever auditing is turned on, a "current" audit file is required and a "next" audit file (for backup) is recommended (see *audsys*(1M) and *audomon*(1M)). When the "current" audit file is full and the "next" audit file is available, the auditing system switches files automatically.

AUTHOR

audit was developed by HP.

SEE ALSO

audsys(1M), audevent(1M), audisp(1M), audomon(1M), audwrite(2), getevent(2), setevent(2).

a

NAME

authcap - security databases for trusted systems

SYNOPSIS

/tcb/files/auth/*
/tcb/files/auth/system/*

DESCRIPTION

All security-relevant databases are stored in an ASCII format in the file system. This format is converted to binary structures by support routines described in Section 3 manual entries. This manual entry describes the format of these databases, and describes the philosophy of conversion into data structures.

Hierarchy Structure

The complete database resides in two hierarchies: **/tcb/files/auth/*** and **/tcb/files**. The first hierarchy contains the Protected Password database, and has subdirectories with single letter names, each of which is a starting letter for user names. Within each of these directories are regular files, each containing an *authcap*(4) format file containing the Protected Password entry for a particular user. Thus, all user names beginning with **x** have their respective authentication and identity information in a file in directory **/tcb/files/auth/x**.

Directories within **/tcb/files/auth/system** and **/tcb/files** contain system-wide information. Global system settings reside in directory **/tcb/files/auth/system**. Terminal and device assignment files are located in directory **/tcb/files**.

The following database files reside in directory **system**:

default	Default Control

The following database files reside in directory **/tcb/files**:

ttys	Terminal Control
devassign	Device Assignment

File Format

Each data file (**/tcb/files/auth/system** and **/tcb/files**)has the same format. Each file consists of one virtual line, optionally split into multiple physical lines with the \ character present at the end of all lines except the last. For example, the line

```
smk:u_name=smk:u_id#16:u_pwd=a78/a1.eitfn6:chkent:
```

can be split into:

```
smk:u_name=smk:u_id#16:\
     :u_pwd=a78/a1.eitfn6:\
     :chkent:
```

Note that all capabilities must be immediately preceded and followed with the **:** separator. Multiple line entries require **:** at the end of each line and at the beginning of each continuation line in the entry. Continuation lines are indented by a tab character. Multiple entries are separated by a new-line character that is not preceded by a continuation character:

```
daa:u_name=daa:u_id#75:u_maxtries#9:chkent:
smk:u_name=smk:u_id#76:u_maxtries#5:chkent:
```

Line Format

The format of a line is briefly as follows:

```
name:cap1:cap2:cap3:...:capn:chkent:
```

The entry is referenced by the name. The end of the name part of the entry is terminated by the **:** character.

At the end of each entry is the *chkent* field. This is used as an integrity check on each entry. The *authcap*(3) routines reject all entries that do not contain the *chkent* terminator.

Each entry has 0 or more capabilities, each terminated with the **:** character. Each capability has a unique name. Numeric capabilities have the format:

*id***#***num*

a

where *num* is a decimal or (0-preceded) octal number. Boolean capabilities have the format:

id

or

id@

where the first form signals the presence of the capability and the second form signals the absence of the capability. String capabilities have the format:

`id=string`

where string is 0 or more characters. The `\` and `:` characters are escaped as `\\` and `\:` respectively.

File Locking

All databases use a lock file, the existence of which means that the file is currently being rewritten. Occasionally, the lock files remain after a system crash and must be removed manually. The lock file is formed by appending `-t` to the database file name.

Fields/Flags

All databases are converted into structures by programs. The data structures consist of two substructures, each of which has one member for each field in the database entry. The *field* structure contains a field value (for example, a number, a boolean flag, a directory string, or a mask), while the flag value (one bit) indicates the presence or absence of the field in that entry.

AUTHOR

SecureWare Inc.

SEE ALSO

default(4), devassign(4), getdvagent(3), getprdfent(3), getprpwent(3), getprtcent(3), prpwd(4), ttys(4).

b

NAME

/stand/bootconf - boot device configuration table

DESCRIPTION

This file contains the address and disk layout type of the system's boot devices or *lif* volumes. It is used by the **Software Distributor** and HP-UX kernel control scripts (fileset **OS-Core.KERN-RUN**) to determine how and where to update the initial boot loader. Normally the kernel's **checkinstall** script queries the system's hardware and creates the file. In rare cases when either the system configuration cannot be automatically determined or additional and/or alternate boot devices should be automatically updated, the administrator must edit the **/stand/bootconf** file manually.

There is one line in the file for each boot device. Each line contains the following blank-separated fields in the order shown:

disk type A flag indicating how the file system(s) on the disk are laid out. The flag must be one of the following:

- **l** Indicates that the root disk is in LVM format. If LVM mirrors are used, then each of the "mirrors" must have its own line.
- **p** Indicates that the root disk has Series 800-style hard partitions and that the boot volume is is section 6.
- **w** Indicates that the root disk is in the Release 9.X Series 700-style "whole disk" format with no partitions, but boot and swap space are reserved outside the file system.

device file The absolute path of the device special file that accesses the physical device where the boot area is located. For LVM root disks, the device special file is the physical volume(s) returned by the **vgdisplay -v** command. For Series 800 hard partitions, this is the device special file that points to section six of the disk. For Series 700-style "whole disks" this is the device file that references the entire disk.

Blank lines are permitted. Any line beginning with a **#** is considered to be a comment.

DIAGNOSTICS

The Software Distributor log file **/var/adm/sw/swagent.log** contains diagnostic messages under the **OS-Core.KERN-RUN** fileset if the **bootconf** file is incorrect. Most of the messages are self-explanatory; a few warrant additional explanation:

... is either empty or improperly formatted...
If there are no other messages about **bootconf**, the file is probably empty. Otherwise, the file is not in the proper format, and the other messages will explain what the problem is.

device file **... does not contain a valid boot LIF ...**
The specified device file does not point to a disk where there is a *lif* which contains the file **HPUX**.

... has an invalid character in the flag field...
Some character other than **#**, **l**, **p**, or **w** is in the first field of a line.

... contains contradictory boot LIF types...
As of release 10.0, the boot areas in **/stand/bootconf** must all be on the same type of disk layout.

... has unrecognized extra characters...
There are characters after the *device file* specification.

EXAMPLES

The boot area is on an LVM root disk:

```
# Boot Device configuration file
# This file contains information regarding the location
# of the boot LIF. It is used by the KERN-RUN fileset to
# update the boot kernel.
l /dev/dsk/c2t7d0
```

The system has LVM mirroring on root (the device files indicate that the system is running on a 9.0 release being prepared for updating to 10.0):

```
# Boot Device configuration file
# This file contains information regarding the location
# of the boot LIF. It is used by the KERN-RUN fileset to
# update the boot kernel.
l /dev/dsk/c1d0s2
l /dev/dsk/c4d0s2
l /dev/dsk/c5d0s2
```


The boot area is on a hard partitioned disk:

```
# Boot Device configuration file
# This File contains information regarding the location
# of the boot LIF. It is used by the KERN-RUN fileset to
# update the boot kernel.
p /dev/dsk/0s0
```

The boot area is on a whole disk layout:

```
# Boot Device configuration file
# This File contains information regarding the location
# of the boot LIF. It is used by the KERN-RUN fileset to
# update the boot kernel.
w /dev/dsk/6s0
```

WARNINGS

All of the boot devices in the file must have the same disk layout.

AUTHOR

bootconf was developed by the Hewlett-Packard Company.

FILES

/stand/bootconf

SEE ALSO

mediainit(1), hpux(1M), mkboot(1M), vgdisplay(1M), lif(4).

Software Distributor documentation.

NAME

cdnode - format of a CDFS cdnode

SYNOPSIS

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

DESCRIPTION

This entry describes the cdnode structure and related concepts for the CDFS file system. Refer to other *inode*(4) manual pages for information regarding the inode structure for other file systems.

The CDFS file system does not have the concept of a separate entity called an inode. The information normally found in an HFS inode is kept in a **cdnode** data structure. However, the cdnode data structure does not reside on the physical media, but instead is kept in kernel memory space only. The cdnode information is used to uniquely identify a file.

The information kept in the cdnode structure is obtained from two other data structures in the CDFS file system:

1. Directory record for the file or directory, and
2. Extended attribute record (XAR) for the file or directory, if one exists.

Because few files usually have XARs associated with them, the cdnode information most often consists only of attributes given by the directory record for the file.

Since cdnodes are kept in kernel memory, they cannot be directly accessed by the user. The **stat()** system call attempts to map whatever information is included in the cdnode for a given file into the standard stat structure (see *stat*(2)). However, since a cdnode includes information that does not have corresponding fields in the stat structure, that information cannot be mapped and therefore cannot be accessed. No method is provided to access an entire cdnode structure.

FILES

/usr/include/sys/cdnode.h
/usr/include/sys/cdfsdir.h

SEE ALSO

stat(2), cdrom(4), cdfsdir(4).

NAME

cdrom - CD-ROM background information

DESCRIPTION

This manual entry provides general information on existing CD-ROM standards, terminology, data layout, and levels of support. More detailed information is available in the standard documents listed in SEE ALSO.

Not all topics discussed here are supported in the current HP-UX release. Refer to the DEPENDENCIES section for details about the contents of the current release.

Standard Formats

Currently, two standard formats are defined for CD-ROM.

The High Sierra Group (HSG) standard was produced by the CD-ROM Ad Hoc Advisory Committee, and is documented in a publication entitled *The Working Paper for Information Processing – Volume and File Structure of Compact Read Only Optical Discs for Information Interchange*. This document is available from the National Information Standards Organization (NISO).

The second standard, which evolved from the HSG standard, was produced by the International Organization for Standardization (ISO). This standard is documented in a publication entitled *Information Processing – Volume and File Structure of CD-ROM for Information Interchange*, reference number ISO 9660: 1988 (E).

Data Layout

The data layout on a CD-ROM can be represented as follows:

System Area - 32 kilobytes
Volume Descriptor
. . .
Volume Descriptor Terminator
. . .
Path Table
Path Table
. . .
Directory and File Data . . .

There are typically four sections in the CD-ROM data (indicated by double horizontal lines in the table above): Only the first two sections must occur in the order shown above.

The **System Area** consists of the first sixteen 2048-byte blocks on the media. The contents of this section are not specified by either standard; here, the creator of the CD-ROM can put data that is relevant to the system for which the CD-ROM is intended.

The **Volume Descriptor** typically contains one primary volume descriptor and zero or more supplementary volume descriptors. Each volume descriptor is 2048 bytes in length, and describes the attributes and structure of a directory hierarchy on the CD-ROM. The list of volume descriptors is terminated by one or more **volume descriptor terminators**. A volume descriptor terminator is also 2048 bytes in length, and simply signals the end of the volume descriptor section.

The **Path Table** contains all the path tables for all directory hierarchies on the CD-ROM. Path tables do not have to be constrained to this section of the CD-ROM data, but can be interspersed with **Directory and File Data** (described below) to minimize seek times.

The **Directory and File Data** contains data for all directory hierarchies on the CD-ROM and, as described above, can be made noncontiguous by the occasional inclusion of a path table.

Volumes and Directory Hierarchies

A **volume** is a single physical CD-ROM. A **directory hierarchy** is a hierarchical file system written on a volume. Multiple directory hierarchies can be placed on a single volume, or a single directory hierarchy can span multiple volumes. Each directory hierarchy on a volume is described by a **volume descriptor**.

Directory hierarchies on the same volume can be totally independent of each other with each one defining a totally unique and unrelated file system. They can also be related to each other through the sharing of data between them.

C

A **volume set** is a set of one or more volumes that are to be treated as a unit. Each successive volume in the volume set updates or augments the data on the volumes preceding it. Thus, the last volume in a volume set is always the volume which describes the most up-to-date directory hierarchy for the volume set. A unique and ascending value called the **volume sequence number**, is assigned to each volume in a volume set. Volume sets are useful for updating large multivolume databases without having to rework the entire set.

Volume Descriptors

Each directory hierarchy on a volume is described by a **volume descriptor**. There are several types of volume descriptors, but the two of most interest are the **primary volume descriptor** and the **supplementary volume descriptor**. Their content is almost identical, but they have different intended uses.

The primary volume descriptor describes the primary directory hierarchy on a volume. If there are additional directory hierarchies on the volume, or different ways to view the same directory hierarchy, these are described by supplementary volume descriptors. In the case of a volume set, the primary volume descriptor on each volume describes the primary directory hierarchy for that volume and all preceding volumes in the set thus far.

Volume descriptors contain the following information:

standard ID (identifies the format of the volume);
system ID;
volume ID;
size of the volume;
volume set size;
volume sequence number;
logical block size;
path table size;
pointers to the path tables;
directory record for the root directory;
volume set ID;
publisher ID;
data preparer ID;
application ID;
copyright file name;
abstract file name;
bibliographic file name (ISO only);
volume creation date and time;
volume modification date and time;
volume expiration date and time;
volume effective date and time;
application use area.

Path Tables

A **path table** defines a directory hierarchy structure within a volume. Each path table contains a record for each directory in the hierarchy. In each record are kept the directory's name, the length of any extended attribute record associated with the directory, the logical block number of the block in which the directory begins, and the number of the parent directory for that directory. (All directories in a path table are numbered according to the order in which they appear in the path table.)

There are two types of path tables. One is a **type-L** path table in which all numerical values in each path table record are recorded least-significant-byte-first. The other type, **type-M**, is a path table in which all numerical values are recorded most-significant-byte-first. One of each type of path table is required by both standards. The ISO standard allows for one additional optional copy of each type of path table, while the HSG standard allows for up to three additional optional copies of each type. Additional copies of path tables are useful for redundancy or seek time minimization.

Extended Attribute Records

An **extended attribute record** (abbreviated **XAR**) is a data structure specifying additional information about the file or directory with which the XAR is associated. An XAR contains the following information:

owner id;
group id;
permissions;
creation date and time;
modification date and time;
expiration date and time;
effective date and time;
record information;
application use area.

If an XAR is recorded, the XAR is written beginning at the first block of the file or directory. The actual data for the file or directory is written beginning at the next block after the block in which the XAR ends.

Where possible, XAR information is mapped into the stat structure by the **stat()** system call (see *stat*(2)). However, many items do not map very well due to lack of appropriate fields in the stat structure for information provided by the XAR. To preserve backward compatibility of the stat structure, such information is discarded by **stat()**. The **fsctl()** system call can be used to obtain the XAR for a particular file or directory (see *fsctl*(2)).

Interleaving

For performance reasons, data in a file can be interleaved when recorded on the volume. This is accomplished by dividing the file into pieces called **file units**. The size of each file unit (in logical blocks) is called the **file unit size**. The interleaved file is then recorded onto the volume by writing a file unit, skipping one or more blocks, writing another file unit, skipping more blocks, and so on until the entire file is recorded. The number of blocks to skip between file units is called the **interleave gap size**. Blocks making up the interleave gap are available for assignment to other files.

File unit and interleave gap sizes are kept in the directory record for each file. Thus, the file unit and interleave gap sizes may change from file to file, but cannot change within the same file (unless the file is written in **sections** – see below).

Directories cannot be interleaved.

File Sections

In order to be able to share data between files, a file can be broken up into **file sections**. File sections for a particular file are not necessarily all the same size.

Each file section is treated like a separate file in that each section gets its own directory record. This implies that each file section has its own size, its own XAR, and its own unique file unit and interleave gap sizes. However, all file sections for the same file must all share the same file name. The order of the file sections in the file is determined by the order of the directory records for each section. A bit in each directory record determines whether or not that record is the last record for the file.

A file section can appear more than once in a single file, or appear many times in many different files. A file section in one volume can also be claimed by a file in a subsequent volume in a volume set (this is how updates are accomplished).

Each file section can have its own XAR. However, if the final file section of a file has no associated XAR, the entire file is treated as if it has no XAR. This is done to make updates work sensibly.

Directories must always consist of a single section.

Implementation and Interchange Levels

CD-ROM standards define two levels of implementation and three levels of interchange. **implementation levels** provide a way for receiving systems that support CD-ROM to specify their level of support. The implementation levels are:

Level 1 The system is permitted to ignore supplementary volume descriptors, their associated path tables, and all directory and file data associated with them.

Level 2 No restrictions apply.

In all cases, receiving systems must fulfill the receiving system requirements specified in section 10 of the ISO standard (no equivalent section exists for HSG).

Interchange levels provide a way to specify the data structure and complexity that exists on a CD-ROM. The levels are:

Level 1 Each file consists of a single file section. File names contain no more than eight characters, and file name extensions contain no more than three. Directory names contain no more than eight characters.

Level 2 Each file consists of a single file section.

Level 3 No restrictions apply.

DEPENDENCIES

HP-UX supports only the primary volume descriptor. When a volume is mounted, HP-UX mounts the directory hierarchy described by the first primary volume descriptor it finds. Supplementary volume descriptors are recognized and ignored, as are their associated directory hierarchies.

Directory hierarchies spanning multiple volumes are not supported.

Volume sets consisting of more than one volume are not supported.

Path tables are ignored in HP-UX. The normal path name lookup scheme used in HFS file systems is used instead. This is done to allow other mountable file systems to be mounted on top of a mounted CDFS file system. Also, since HP-UX maintains a cache of cdnodes for CDFS files (see *cdnode*(4)), the additional performance gains provided by path tables are minimal.

HP-UX does not support multiple file sections. Each file must be recorded in a single file section.

HP-UX supports level 1 implementation and level 2 interchange.

NOTES

Additional CD-ROM formats are supported using PFS (Portable File System) utilities. See *pfs*(4) for more details.

SEE ALSO

fsctl(2), stat(2), cdnode(4), pfs(4).

Information Processing - Volume and File Structure of CD-ROM for Information Interchange, Ref. No. ISO 9660: 1988 (E).

The Working Paper for Information Processing – Volume and File Structure of Compact Read Only Optical Discs for Information Interchange, National Information Standards Organization [Z39].

C

NAME

charmap - symbolic translation file for localedef scripts

SYNOPSIS

localedef -f *charmap* **locale_name**

DESCRIPTION

Invoking the **localedef** command with the **-f** option causes symbolic names in the **locale description file** to be translated into the encodings given in the *charmap* file (see *localedef*(1M)). As a recommendation, a locale description file should be written completely with symbolic names.

The *charmap* file has two sections: a declarations section and a character definition section.

Declarations Section

Declarations can precede the character definitions.

Each consists of the symbol (including the surrounding angle brackets), followed by one or more blanks (or tabs or space characters), followed by the value of the symbol.

Certain declarations are required for multibyte character codesets. For single-byte codesets, all are optional.

Following is a list of possible declarations:

<code_set_name> *value*

Used to declare the name of the coded character set for which the charmap file is defined. This keyword is required for multibyte character codesets. For HP15 encoding scheme, **HP15** needs to be part of the name. For EUC encoding scheme, **EUC** needs to be part of the name.

<cswidth> *value*

Used to declare the cswidth parameter of the coded character set for which the charmap file is defined (see *euset*(1)).

<mb_cur_max> *value*

Used to declare the maximum number of bytes in a multibyte character. Defaults to 1 if not given. For multibyte character codesets, this keyword must be specified.

<mb_cur_min> *value*

Used to declare the minimum number of bytes in a character for the encoded character set. The value must be less than or equal to **<mb_cur_max>**. If not given, the default is equal to **<mb_cur_max>**.

<escape_char> *value*

Used to declare the escape character, which is used to escape characters that otherwise would have special meaning. If not given, the default is backslash (****).

<comment_char> *value*

Used to declare the comment character, which is used to begin comments and should be placed in column one of the *charmap* file. If not given, the default is the **#** character.

Character Definition Section

The character-set mapping definitions immediately follow an identifier line containing the string **CHARMAP** and precede a trailer line consisting of the string **END CHARMAP**. (Empty lines and lines beginning with the comment character are ignored.)

The character definitions are of two forms.

The first form defines a single character and its encoding:

<*symbolic_name***>** *encoding*

A *symbolic_name* is one or more visible characters from the portable character set as specified by XPG, enclosed in angle brackets. Metacharacters such as angle brackets, escape characters, or comment characters must be escaped if they are used in the name. Two or more symbolic names can be given for the same encoding.

The *encoding* is a character constant in one of three forms:

- decimal — An escape character followed by the letter **d**, followed by one to three decimal digits.
- octal — An escape character followed by one to three octal digits.
- hexadecimal — An escape character followed by an **x**, followed by two hexadecimal digits.

Multibyte characters are represented by the concatenation of character constants. All constants used in the encoding of a multibyte character must be of the same form.

The second form defines a range of characters consisting of all characters from the first symbolic name to the second, inclusive:

<*symbolic_name***>**... **<***symbolic_name***>** *encoding*

The symbolic name must consist of one or more nonnumeric characters followed by an integer formed of one or more decimal digits. The integer part of the second symbolic name must be larger than that of the first. The range is then interpreted as a list of symbolic names consisting of the same character portion and successive integer values from the first through the last. These names are assigned successive encodings starting with the one given.

For example, the character definition line

```
<C4>...<C6>    \d129
```

is equivalent to:

```
<C4>          \d129
<C5>          \d130
<C6>          \d131
```

EXAMPLES

For examples, see any of the files under **/usr/lib/nls/loc/charmaps** directory.

SEE ALSO

localedef(1M), localedef(4).

STANDARDS COMPLIANCE

localedef POSIX.2, XPG4.

C

NAME

core - format of core image file

DESCRIPTION

The HP-UX system writes out a file containing a core image of a terminated process when certain signals are received (see *signal*(5) for the list of reasons). The most common causes are memory violations, illegal instructions, floating point exceptions, bus errors, and user-generated quit signals. The core image file is called **core** and is written in the process's working directory (provided it is allowed by normal access controls). A process with an effective user ID different from its real user ID does not produce a core image.

The file contains sufficient information to determine what the process was doing at the time of its termination. Core file contents consist of objects that represent different segments of a process. Each object is preceded by a **corehead** data structure, and each **corehead** data structure describes the corresponding object following it. The structure is defined in <**sys/core.h**>, and includes the following members:

```
int       type;
space_t   space;
caddr_t   addr;
size_t    len;
```

The *space* and *addr* members specify the virtual memory address in the process where the described object began. The *len* member is the length of the object in bytes.

The following possible values for *type* are defined in <**sys/core.h**>:

CORE_DATA Process data as it existed at the time the core image was created. This includes initialized data, uninitalized data, and the heap at the time the core image is generated.

CORE_EXEC A compiler-dependent data structure containing the exec data structure, the magic number of the executable file, and the command (see the declaration of the **proc_exec** structure in <**sys/core.h**>).

CORE_FORMAT The version number of the core format produced. This number changes with each HP-UX release where the core format itself has changed. However, it does not necessarily change with every HP-UX release. **CORE_FORMAT** can thus be easily used by core-reading tools to determine whether they are compatible with a given core image. This type is expressed by a four-byte binary integer.

CORE_KERNEL The null-terminated version string associated with the kernel at the time the core image was generated.

CORE_PROC An architecture-dependent data structure containing per-process information such as hardware register contents. See the declaration of the **proc_info** structure in <**sys/core.h**>.

CORE_STACK Process stack contents at the time the core image was created.

Objects dumped in a **core** image file are not arranged in any particular order. Use **corehead** information to determine the type of the object that immediately follows it.

SEE ALSO

adb(1), cdb(1), xdb(1), setuid(2), crt0(3), end(3C), signal(5).

C

NAME

cpio - format of cpio archive

DESCRIPTION

The *header* structure, when the **-c** option of **cpio** is not used (see *cpio*(1)), is:

```
struct {
    short   c_magic,
            c_dev;
    ushort  c_ino,
            c_mode,
            c_uid,
            c_gid;
    short   c_nlink,
            c_rdev,
            c_mtime[2],
            c_namesize,
            c_filesize[2];
    char    c_name[c_namesize rounded to word];
} Hdr;
```

When the **cpio -c** option is used, the *header* information is described by:

```
sscanf(Chdr,"%6ho%6ho%6ho%6ho%6ho%6ho%6ho%6ho%11lo%6ho%11lo",
          &Hdr.c_magic,&Hdr.c_dev,&Hdr.c_ino,&Hdr.c_mode,
          &Hdr.c_uid,&Hdr.c_gid,&Hdr.c_nlink,&Hdr.c_rdev,
          &Longtime,&Hdr.c_namesize,&Longfile);
```

Longtime and *Longfile* are equivalent to **Hdr.c_mtime** and **Hdr.c_filesize**, respectively. The contents of each file are recorded together with other items describing the file. Every instance of **c_magic** contains the constant 070707 (octal). The items **c_dev** through **c_mtime** have meanings explained in *stat*(2). The length of the null-terminated path name **c_name**, including the null byte, is given by **c_namesize**.

The last record of the *archive* always contains the name **TRAILER!!!**. Directories and the trailer are recorded with **c_filesize** equal to zero.

It will not always be the case that **c_dev** and **c_ino** correspond to the results of **stat()**, but the values are always sufficient to tell whether two files in the archive are linked to each other.

When a device special file is archived by HP-UX **cpio** (using the **-x** option), **c_rdev** contains a magic constant which is dependent upon the implementation doing the writing. **H_rdev** flags the device file as an HP-UX 32-bit device specifier, and **c_filesize** contains the 32-bit device specifier (see *stat*(2)). If the **-x** option is not present, special files are not archived or restored. Non-HPUX device special files are never restored.

SEE ALSO

cpio(1), find(1), stat(2).

STANDARDS CONFORMANCE

cpio: XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

default - system default database file for a trusted system

SYNOPSIS

/tcb/files/auth/system/default

DESCRIPTION

The system default database is unique in that it defines system-wide global parameters for a trusted system. It is designed to provide values for users and devices on a global scale rather than requiring an administrator to replicate values in user or device databases when they are all the same. In addition to being easier to specify global values, it is also much easier to make a global system change if necessary.

The system default database is made up of four types of values:

system-wide parameters — These are parameters that do not have corresponding specifications in any other trusted system database. If a system-wide parameter is not specified in the default database, then it is undefined.

user parameters — These parameters are typically specified in a protected password database file.

terminal control parameters — These parameters are typically specified in the terminal control database file.

device assignment parameters — These parameters are typically specified in the device assignment database file.

System default parameters may be specified for fields found in the protected password, terminal control, and device assignment databases. When a specific entry is retrieved from one of these databases, a structure called, *ufld* that contains all of the explicitly specified values, is provided to the caller. A second structure, called *sfld*, is also provided which defines those values supplied from the system default database. Each of these structures has a corresponding flag structure called *uflg* and *sflg*, respectively, that indicates which fields in each structure have been specified and are valid for use. Programs honor the user or device specific value first if one is provided. Otherwise, the program may choose to use the system default value if one has been specified. If neither value is specified, the program may supply a reasonable default value or abort.

For descriptions of the specific fields provided by the protected password, terminal control, and device assignment databases, see the corresponding manual pages listed in the SEE ALSO section for those databases. The following fields are unique to the system default database and can not be specified in any of the other system databases.

d_name — This name is set to the string "default".

d_boot_authenticate — This flag field indicates whether or not boot authentication is required to boot the machine. If authentication is required, it is performed by the system *init*(1M) program prior to completing system boot.

EXAMPLES

The following is an example of a typical system default database. Refer to *authcap*(4) for descriptions of the file and line formats.

```
default:\
     :d_name=default:\
     :d_boot_authenticate@:\
     :u_pwd=*:\
     :u_minchg#0:u_maxlen#10:u_exp#15724800:u_life#31449600:\
     :u_pickpw@:u_genpwd@:u_restrict@:u_nullpw@:\
     :u_genchars@:u_genletters@:\
     :u_maxtries#5:u_lock:\
     :t_logdelay#2:t_maxtries#10:\
     :chkent:
```

This system default database defines the four different types of values which are supported. First, values that can be assigned on a system-wide only basis are defined. Boot authentication at system startup is not enabled. Login programs will provide password expiration warnings if the password expires in less than 604800 seconds from the current system time (this translates into 60*60*24*7 or 7 days).

The system default database also defines numerous protected password database default values. Fields that begin with **u_** correspond to protected password fields. Similarly, fields starting with the **t_** prefix are terminal control database fields. These field types are used to supply system-wide default values if a user or device specific value is not supplied by the corresponding database. See the appropriate manual pages listed in the SEE ALSO section for these databases for a complete description of the applicable fields.

d

FILES

/tcb/files/auth/system/default	system default database file for a trusted system; see *authcap*(4)
/tcb/files/auth/*/*	protected password database files; see *prpwd*(4)
/tcb/files/ttys	terminal control database file; see *ttys*(4)
/tcb/files/devassign	device assignment database file; see *devassign*(4)

AUTHOR

SecureWare Inc.

SEE ALSO

getprdfent(3), authcap(4), devassign(4), prpwd(4), ttys(4).

NAME

devassign - device assignment database file for a trusted system

SYNOPSIS

/tcb/files/devassign

DESCRIPTION

The system supports a single device assignment database that contains entries for local and remote login terminals.

The format of the terminal control database file is identical to other trusted system authentication database files. For more information on the file format, see *authcap*(4). The file consists of keyword field identifiers and values for those fields. The keyword identifiers supported and their use include:

v_devs This field specifies a comma separated list of aliases that refer to the same device defined by the entry. Use of this field avoids the need to replicate device assignment database entries for all device aliases.

v_type This field specifies the device that is described by the entry. Device types supported include:

terminal The device is assigned as a local or remote login terminal device.

v_users This field, if specified, contains a comma separated list of user names that are permitted to use the device for login or the import/export of data. If the list is not present, all users are permitted to use the device. If the list is present, it is searched for a match by the *login* program to determine if the user is permitted to use the device.

EXAMPLES

The following is an example of a device assignment database entry for a terminal device assigned as a login device:

```
tty0:v_devs=/dev/tty0:\
     :v_type=terminal:\
     :chkent:
```

AUTHOR

SecureWare Inc.

SEE ALSO

cpio(1), login(1), tar(1), getdvagent(3), authcap(4), default(4).

d

d

NAME

devices - file of driver information for insf, mksf, lssf

DESCRIPTION

The **devices** file contains a description of I/O drivers, pseudo-drivers, hardware addresses and block/character major numbers. It is created by **uxgen** (see *uxgen*(1M)). This file normally resides in the directory **/etc**.

This is an ASCII file consisting of zero or more lines where each line is terminated by a new-line character. Each line begins with a name which normally represents an I/O driver or pseudo-driver. Tokens are separated by white space.

Each parameter in the line is preceded by a keyword. All parameters are optional. The keywords are: **lu**, **address**, **b_major**, and **c_major**. representing logical unit number, hardware address, block major number, character major number, respectively. Parameters can appear in any order after the name; however, they must be directly preceded by their keyword.

The following lines represent typical entries in a **devices** file:

```
cn                                                c_major 0
disc0   lu 0    address 28.0.0   b_major 0    c_major 4
disc0   lu 1    address 28.0.2   b_major 0    c_major 4
```

AUTHOR

devices was developed by HP.

SEE ALSO

insf(1M), mksf(1M), lssf(1M), uxgen(1M).

NAME

dialups, d_passwd - dialup security control

DESCRIPTION

dialups and **d_passwd** are used to control the dialup security feature of **login** (see *login*(1)). If **/etc/dialups** is present, the first word on each line is compared with the name of the line upon which the login is being performed (including the **/dev/**, as returned by **ttyname()** (see *ttyname*(3C)). If the login is occurring on a line found in **dialups**, dialup security is invoked. Anything after a space or tab is ignored.

When dialup security is invoked, **login** requests an additional password, and checks it against that found in **/etc/d_passwd**. The command name found in the "program to use as shell" field of **/etc/passwd** is used to select the password to be used. Each entry in **d_passwd** consists of three fields, separated by colons. The first is the command name, matching an entry in **passwd**. The second is the encrypted password to be used for dialup security for those users logging in to use that program. The third is commentary, but the second colon is required to delimit the end of the password. A null password is designated with two adjacent colons. The entry for **/usr/bin/sh** is used if no other entry matches the command name taken from **passwd**.

FILES

/etc/dialups	dial-in tty lines
/etc/d_passwd	passwords

SEE ALSO

login(1), passwd(4).

d

NAME

dir - format of directories on short-name HFS file systems

SYNOPSIS

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

REMARKS

This entry describes the System V-compatible directory format for the HFS file system. It is provided strictly for backward compatibility and compatibility with applications expecting a System V file system environment. It is not compatible with the similar but more general HFS directory format in <**dirent.h**>, which describes a format identical to that used in an HFS file system supporting long file names up to 255 bytes in length.

The **dirent** structure defined in <**dirent.h**> should be used in conjunction with the *directory*(3C) routines for portability to other industry UNIX implementations.

DESCRIPTION

A directory behaves exactly like an ordinary file, except that no user can write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry (see *fs*(4)). The structure of a directory entry as given in the <**sys/dir.h**> header file is:

```
#define DIRSIZ              14
#define DIRSIZ_CONSTANT     14
#define DIR_PADSIZE         10
#define MAXNAMLEN           255
struct      direct {
    u_long  d_ino;          /* inode number of entry */
    u_short d_reclen;       /* length of this record */
    u_short d_namlen;       /* length of string in d_name */
    char    d_name[DIRSIZ_CONSTANT];
    char    d_pad[DIR_PADSIZE];
};
/*
 * DIRSTRCTSIZ is the number of bytes in the structure
 * representing a System V-compatible (14-character
 * maximum file name length) HFS directory entry.
 */
#define DIRSTRCTSIZ  32      /* sizeof(struct direct) */
```

By convention, the first two entries in each directory are for **.** and **..** (“dot” and “dot dot”). The first is an entry for the directory itself. The second is for the parent directory. The meaning of **..** is modified for the root directory of the master file system; there is no parent, so **..** and **.** have the same meaning.

AUTHOR

dir was developed by AT&T and HP.

SEE ALSO

fs(4), directory(3C).

NAME

disktab - disk description file

SYNOPSIS

#include <disktab.h>

DESCRIPTION

disktab is a simple database that describes disk geometries. Entries in **disktab** consist of a number of colon-separated fields. The first entry for each disk gives the names by which the disk is known, separated by vertical bar (**|**) characters.

This file is provided for backward compatibility with previous HP-UX releases only. Its use is discouraged.

The following list indicates the normal values stored for each disk entry. Sectors are of size **DEV_BSIZE**, defined in <**sys/param.h**>.

Name	Type	Description
ns	**num**	Number of sectors per track
nt	**num**	Number of tracks per cylinder
nc	**num**	Total number of cylinders on the disk
b0	**num**	Block size (bytes)
f0	**num**	Fragment size (bytes)
s0	**num**	Size of disk in sectors
rm	**num**	Revolution per minute

Example:

```
HP_7914:
    :132.1 MB:ns#16:nt#7:nc#1152:\
    :s0#129024:b0#8192:f0#1024:\
    :se#256:rm#3600:
```

AUTHOR

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

FILES

/etc/disktab

SEE ALSO

newfs(1M), getdiskbyname(3C).

d

d

NAME

DOSIF - DOS Interchange Format description

DESCRIPTION

DOSIF (DOS Interchange Format) is the name given to the media format used by the DOS operating system. This format is based upon that used in IBM PC and PC AT and HP Vectra systems.

The DOS utilities described in Section 1 (referred to hereafter as *dos*∗(1)) are provided for reading data from and writing data to DOSIF volumes. Use these utilities to retrieve information from a DOSIF volume.

The *dos*∗(1) utilities are the only HP-UX commands that can interact directly with the contents of a DOSIF volume. The only other way to interact with the contents of a DOSIF volume is to use an HP-UX DOS emulation or coprocessor facility such as SoftPC or the DOS Coprocessor. **mount** cannot be used on a DOSIF volume because the operating system does not recognize it (see *mount*(1M)).

When constructing file names for *dos*∗(1) commands, start with the HP-UX path name of the DOSIF volume, then add a colon (**:**) followed by the file name:

device_file **:** *file*

or

path_name **:** *file*

Note: This file naming convention is suitable for use only in arguments to the *dos*∗(1) utilities. It does not constitute a legal path name for any other use in HP-UX applications.

Metacharacters *****, **?**, and **[** ... **]** can be used when specifying both HP-UX and DOS file names. These must be quoted when specifying a DOS file name, because file name expansion must be performed by the DOS utilities, not by the shell. The *dos*∗(1) utilities expand file names as described in *regexp*(5) under *PATTERN MATCHING NOTATION*.

By convention, if the HP-UX device name and a trailing colon are specified, but no file or directory name is provided (for example, **/dev/rdsk/c1t1d0:**), the root (**/**) of the DOS file system is assumed.

EXAMPLES

Specify DOSIF file **/dos/ivy** accessed through HP-UX special file **/dev/rdsk/c1t1d0**:

/dev/rdsk/c1t1d0:/dos/ivy

Specify DOSIF file **/math** accessed through the DOS volume stored as HP-UX file **/home/mydir/driveC**:

/home/mydir/driveC:/math

SEE ALSO

dos2ux(1), doschmod(1), doscp(1), dosdf(1), dosls(1), dosmkdir(1), dosrm(1).

NAME

dp - dedicated ports file used by DDFA software and Telnet port identification feature

DESCRIPTION

The **dp** file has two uses:

Datacommunications and Terminal Controller Device File Access

The **dp** file is used by the Datacommunications and Terminal Controller Device File Access (DDFA) software to allow terminal server ports to be programmatically accessed from HP-UX applications in the same way as devices connected directly to the HP-UX system. It contains a one-line entry for each configured terminal server port.

The **dp** file contains the information the DDFA software needs to set up and manage an outbound connection to a specified terminal server port. The file is parsed by the Dedicated Port Parser (**dpp**) which spawns an Outbound Connection Daemon (**ocd**) for each outbound connection specified in the file.

Telnet Port Identification

The **dp** file is used by the HP-UX telnet daemon (**telnetd**) to identify the calling port and board of a telnet connection from an HP Datacommunications and Terminal Controller (DTC).

At connection time, the host negotiates the telnet environment option and the DTC replies with the port and board number of the connecting device. **Telnetd** maps the port and board numbers to the well-known name for the device, which has previously been configured in the **dp** file.

d

Datacommunications and Terminal Controller Device File Access

For outbound connections, an entry should have the following format:

*dtc_name board***/***port pseudonym config_file log_level*

The exact details of each field are given below.

Telnet Port Identification

To configure the **dp** file for using the Telnet port identification feature, the default file **/usr/examples/ddfa/dp** should be copied to a new file and the copy configured with the appropriate values for the incoming connections. The recommended procedure is to create a directory to hold the **dp** file and the modified port configuration files.

An entry for this purpose should have the following format:

*dtc_name board***/***port pseudonym*

The exact details of each field are given below.

Configuration Information

There are three ways to specify a terminal server port:

- Explicitly specify its IP address.
- Specify the node name or the IP address of the DTC then specify the board and port.
- Specify the node name or the IP address of the terminal server and the TCP port service address of the port.

Comments in the **dp** file can be appended by starting them with a **#** character. Everything after the **#** is ignored by the parser. Fields in the **dp** file are separated by space characters.

See *ddfa*(7) for more information on how to configure the DDFA software.

The fields of an entry of the **dp** file are as follows:

dtc_name This field is the node name or the IP address of the terminal server being accessed or the IP address of the port on the terminal server. A node name must be defined in a name database.

*board***/***port* This field contains the terminal server port address with the parts separated by the **/** character. It is not necessary to pad the values with leading zeros. The port address is not checked by **dpp**, but is checked by **ocd**. Valid values are 0 through 7 for *board*, and 0 through 31 for *port* (these restrictions do not apply if the TCP port service address is specified instead).

If the *dtc_name* field explicitly defines the node name or the IP address of the terminal server port, the value in the *board*/*port* field must be **xx/xx** (use **X** or **x**).

If the field is of the form **xx/***n* where *n* is a decimal number, *n* is assumed to be the TCP port service address and it is used when the connection is established.

pseudonym — This field is the absolute path of the device file known to the system and the end-user application. The device file name portion of the path name is limited to 14 characters.

pc_file_path — This field is the path to a port configuration file which contains the configuration information for the terminal server port. This field is mandatory for outbound connections as **dpp** uses the presence of this field as its flag to spawn a daemon for the entry.

d

log_level — This field is the logging level for the particular **ocd** and it determines the severity of messages sent to **/var/adm/syslog**. The logging levels (and how they relate to system logging levels) are as follows:

- **0** Log only LOG_CRIT messages.
- **1** Log only LOG_CRIT and LOG_ERR messages.
- **2** Log only LOG_CRIT, LOG_ERR, and LOG_WARNING messages.
- **3** Log all messages.

It is optional and may only be specified for outbound connections. If it is omitted, the logging level is set to 1.

EXAMPLES

The following examples illustrate file entry syntax.

A printer is connected to port 1 of board 3 of a DTC with the IP address 11.234.87.123. The device attached to the port can be accessed with the HP-UX spooler by using the device file **/dev/telnet/lp1_ocd**.

```
11.234.87.123 03/01 /dev/telnet/lp1_ocd /usr/examples/ddfa/pcf
```

A printer is connected to a terminal server port with IP address 11.234.87.124. The *board*/*port* field contains **xx/xx**. The device attached to the port can be accessed with the HP-UX spooler by using the device file **/dev/telnet/lp2_ocd**.

```
11.234.87.124 xx/xx /dev/telnet/lp2_ocd /usr/examples/ddfa/pcf
```

A printer is connected to a port accessed with TCP port service address 5001 of a terminal server with the IP address 11.234.87.215. The device attached to the port can be accessed with the HP-UX spooler by using the device file **/dev/telnet/lp3_ocd**.

```
11.234.87.215 xx/5001 /dev/telnet/lp3_ocd /usr/examples/ddfa/pcf
```

A terminal is connected to port 1 of board 2 of a DTC with the IP address 11.234.87.215 and wishes to use Telnet port identification.

```
11.234.87.215 02/01 /dev/telnet/tm02
```

WARNINGS

In order to ensure that commands (such as *ps*) display the correct device file name (that is, the *pseudonym*), all pseudonyms should be placed into the directory **/dev/telnet**. If pseudonyms are not specified for placement in this directory, the correct display of device file names with many commands is not guaranteed.

In addition, in order to ensure that commands (such as **w**, **passwd**, **finger**, and **wall**) work correctly, each pseudonym must be unique in its first 17 characters (including the directory prefix **/dev/telnet/**). If pseudonyms are not unique in their first 17 characters, the correct functioning of many commands is not guaranteed.

FILES

/usr/sbin/dpp
/usr/sbin/ocd
/usr/sbin/ocdebug
/var/adm/dpp_login.bin
/var/adm/utmp.dfa
/usr/examples/ddfa/dp
/usr/examples/ddfa/pcf

SEE ALSO

dpp(1M), ocd(1M), ocdebug(1M), syslog(3C), pcf(4), ddfa(7).

NAME

exports, xtab - directories to export to NFS clients

SYNOPSIS

/etc/exports

/etc/xtab

DESCRIPTION

File **/etc/exports** describes the directories that can be exported to NFS clients. The system administrator creates it using a text editor. **mountd** processes it each time a mount request is received (see *mountd*(1M)).

/etc/exports is read automatically by the **exportfs** command (see *exportfs*(1M)). If this file is changed, **exportfs** must be run (**exportfs -a**) before the changes can affect the daemon's operation.

If this file is present at boot time the **/sbin/init.d/nfs.server** script will execute an **exportfs** command and export the file systems listed in the file.

/etc/xtab contains entries for directories that are currently exported. This file should only be accessed by programs using **getexportent** (see *exportent*(3N)). (Use **exportfs -u** to remove entries from this file).

An entry for a directory consists of a command line of the following form:

directory **-** *option*[**,** *option*] **...**

where *directory* is the path name of a directory (or file).

options can have any of the following values and forms:

ro
: Export the directory read-only. If not specified, the directory is exported read-write. The **ro** and **rw** options are mutually exclusive.

rw=*hostname*[**:***hostname*] **...**
: Export the directory read-mostly. Read-mostly means read-only to most machines, but read-write to those specified. If neither **ro** nor **rw** is specified, the directory is exported read-write to all. The **ro** and **rw** options are mutually exclusive.

anon=*uid*
: If a request comes from an unknown user, use *uid* as the effective user ID. *Note*: Root users (uid 0) are always considered "unknown" by the NFS server unless they are included in the **root** option below.

  The default value for this option is 65 534. Setting **anon** to 65 535 disables anonymous access.

root=*hostname*[**:***hostname*] **...**
: Give root access only to the root users from a specified *hostname*. The default is for no hosts to be granted root access. For this option **hostname** cannot be a netgroup name.

access=*client*[**:***client*] **...**
: Give mount access to each client listed. A client can either be a hostname or a netgroup (see *netgroup*(4)). Each client in the list is first checked in the **netgroup** database, then in the **hosts** database. A directory name with no accompanying name list allows any machine to mount the given directory.

async
: Specifying **async** increases write performance on the NFS server by causing asynchronous writes on the NFS server. The **async** option can be specified anywhere on the command line after *directory*. Before using this option, refer to WARNINGS below.

#
: A **#** character anywhere in the file indicates a comment that extends to the end of the line.

/etc/exports contains a list of file systems and the **netgroup** or machine names allowed to remotely mount each file system (see *netgroup*(4)). The file system names are left-justified and followed by a list of names separated by white space. The names are searched for in **/etc/netgroup** then in **/etc/hosts**. A file system name with no accompanying name list means the file system is available to everyone.

A **#** anywhere in the file indicates a comment extending to the end of that line.

EXAMPLES

```
/usr/games cocoa fudge                  # export to only these machines
/usr       -access=clients              # export to my clients
/usr/local                              # export to the world
/usr2      -access=bison:deer:pup       # export to only these machines
/var/adm   -root=bison:deer             # give root access only to these
/usr/new   -anon=0                      # give all machines root access
/usr/temp  -rw=ram:alligator            # export read-write only to these
/usr/bin   -ro                          # export read-only to everyone
/usr/stuff -access=bear,anon=65534,ro
                                        # several options on one line
```

e

WARNINGS

If the **async** option is used, an unreported data loss may occur *ONLY* on a write and *ONLY* if the NFS server experiences a failure after the write reply has been sent to the client. Specifically, blocks which have been queued for the server's disk, but have not yet been written to the disk *may* be lost.

You cannot export either a parent directory or a subdirectory of an exported directory that resides *within the same file system*. It is not allowed, for instance, to export both **/usr** and **/usr/local** if both directories reside on the same disk partition.

AUTHOR

exports was developed by Sun Microsystems, Inc.

FILES

/etc/exports
/etc/xtab
/etc/hosts
/etc/netgroup
/sbin/init.d/nfs.server

SEE ALSO

exportfs(1M), mountd(1M), nfsd(1M), exportent(3N), hosts(4), netgroup(4).

f

NAME

fs - format of file system volume

SYNOPSIS

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/fs.h>
#include <sys/inode.h>
#include <sys/ino.h>
#include <sys/sysmacros.h>
```

DESCRIPTION

Every file system storage volume has a common format for certain vital information. The first 8 kbytes on a volume contain a volume header which identifies that volume as a Logical Interchange Format (LIF) volume. Such volume may be divided into a number of sections.

Each section can contain a file system. The first 8 kbytes in each section is ignored, except where it coincides with the volume header discussed above. The actual file system begins next with the "super block." The layout of the super block as defined by the include file <**sys/fs.h**> is:

```
#define FS_MAGIC        0x011954
#define FS_MAGIC_LFN    0x095014
#define FS_CLEAN        0x17
#define FS_OK           0x53
#define FS_NOTOK        0x31
struct fs {
    struct fs  *fs_link;       /* linked list of file systems */
    struct fs  *fs_rlink;      /* used for incore super blocks */
    daddr_t    fs_sblkno;      /* addr of super-block in filesys */
    daddr_t    fs_cblkno;      /* offset of cyl-block in filesys */
    daddr_t    fs_iblkno;      /* offset of inode-blocks in filesys*/
    daddr_t    fs_dblkno;      /* offset of first data after cg */
    long       fs_cgoffset;    /* cylinder group offset in cylinder*/
    long       fs_cgmask;      /* used to calc mod fs_ntrak */
    time_t     fs_time;        /* last time written */
    long       fs_size;        /* number of blocks in fs */
    long       fs_dsize;       /* number of data blocks in  fs */
    long       fs_ncg;         /* number of cylinder groups */
    long       fs_bsize;       /* size of basic blocks in fs */
    long       fs_fsize;       /* size of frag blocks in fs */
    long       fs_frag;        /* number of frags in a block in fs*/
/* these are configuration parameters */
    long       fs_minfree;     /* minimum percentage of free blocks*/
    long       fs_rotdelay;    /* num of ms for optimal next block */
    long       fs_rps;         /* disk revolutions per second */
/* these fields can be computed from the others */
    long       fs_bmask;       /* ‘‘blkoff’’ calc of blk offsets */
    long       fs_fmask;       /* ‘‘fragoff’’ calc of frag offsets */
    long       fs_bshift;      /* ‘‘lblkno’’ calc of logical blkno */
    long       fs_fshift;      /* ‘‘numfrags’’ calc number of frags*/
/* these are configuration parameters */
    long       fs_maxcontig;   /* max number of contiguous blks */
    long       fs_maxbpg;      /* max number of blks per cyl group */
/* these fields can be computed from the others */
    long       fs_fragshift;   /* block to frag shift */
    long       fs_fsbtodb;     /* fsbtodb and dbtofsb shift constant*/
    long       fs_sbsize;      /* actual size of super block */
    long       fs_csmask;      /* csum block offset */
    long       fs_csshift;     /* csum block number */
    long       fs_nindir;      /* value of NINDIR */
    long       fs_inopb;       /* value of INOPB */
    long       fs_nspf;        /* value of NSPF */
    long       fs_sparecon[6]; /* reserved for future constants */
```

```
/* sizes determined by number of cylinder groups and their sizes */
    daddr_t    fs_csaddr;      /* blk addr of cyl grp summary area */
    long       fs_cssize;      /* size of cyl grp summary area */
    long       fs_cgsize;      /* cylinder group size */
/* these fields should be derived from the hardware */
    long       fs_ntrak;       /* tracks per cylinder */
    long       fs_nsect;       /* sectors per track */
    long       fs_spc;         /* sectors per cylinder */
/* this comes from the disk driver partitioning */
    long       fs_ncyl;        /* cylinders in file system */
/* these fields can be computed from the others */
    long       fs_cpg;         /* cylinders per group */
    long       fs_ipg;         /* inodes per group */
    long       fs_fpg;         /* blocks per group * fs_frag */
/* this data must be re-computed after crashes */
    struct     csum fs_cstotal; /* cylinder summary information */
/* these fields are cleared at mount time */
    char       fs_fmod;        /* super block modified flag */
    char       fs_clean;       /* file system is clean flag */
    char       fs_ronly;       /* mounted read-only flag */
    char       fs_flags;       /* currently unused flag */
    char       fs_fsmnt[MAXMNTLEN];/* name mounted on */
/* these fields retain the current block allocation info */
    long       fs_cgrotor;     /* last cg searched */
    struct     csum  *fs_csp[MAXCSBUFS]; /* list of fs_cs info buffers */
    long       fs_cpc;         /* cyl per cycle in postbl */
    short      fs_postbl[MAXCPG][NRPOS];/*head of blocks per rotation */
    long       fs_magic;       /* magic number */
    char       fs_fname[6];    /* name of file system */
    char       fs_fpack[6];    /* pack name of file system */
    u_char     fs_rotbl[1];    /* list of blocks for each rotation */
/* actually longer */
};
```

A file system consists of a number of cylinder groups. Each cylinder group has inodes and data.

A file system is described by its super-block, which in turn describes the cylinder groups. The super-block is critical data and is replicated in each cylinder group to protect against catastrophic loss. This is done at file system creation time and the critical super-block data does not change, so the copies need not be referenced further unless disaster strikes.

Addresses stored in inodes are capable of addressing fragments of 'blocks'. File system blocks of at most size **MAXBSIZE** can be optionally broken into smaller pieces, each of which is addressable; these pieces may be **DEV_BSIZE**, or some multiple of a **DEV_BSIZE** unit (**DEV_BSIZE** is defined in <**sys/param.h**>).

Large files consist of exclusively large data blocks. To avoid undue wasted disk space, the last data block of a file is allocated only as many fragments of a large block as are necessary, if that file is small enough to not require indirect data blocks. The file system format retains only a single pointer to such a fragment, which is a piece of a single large block that has been divided. The size of such a fragment is determinable from information in the inode, using the **blksize(fs, ip, lbn)** macro.

The file system records space availability at the fragment level; to determine block availability, aligned fragments are examined.

I-numbers begin at 0. Inodes 0 and 1 are reserved. Inode 2 is used for the root directory of the file system. The **lost+found** directory is given the next available inode when it is initially created by **mkfs**.

fs_minfree gives the minimum acceptable percentage of file system blocks that can be free. If the freelist drops below this level, only the super-user may continue to allocate blocks. This can be set to 0 if no reserve of free blocks is deemed necessary. However, severe performance degradations result if the file system is run at greater than 90% full; thus the default value of **fs_minfree** is 10%.

The best trade-off between block fragmentation and overall disk utilization and performance varies for each intended use of the file system. Suggested values can be found in the system administrator's manual for each implementation.

Cylinder-Group-Related Limits

Each cylinder keeps track of the availability of blocks at different rotational positions, so that sequential blocks can be laid out with minimum rotational latency. NRPOS is the number of rotational positions which are distinguished. For example, with NRPOS 8 the resolution of the summary information is 2ms for a typical 3600 rpm drive.

fs_rotdelay gives the minimum number of milliseconds to initiate another disk transfer on the same cylinder. It is used in determining the rotationally optimal layout for disk blocks within a file; the default value for **fs_rotdelay** is 2ms. Suggested values of **fs_rotdelay** for different disks can be found in the system administrator's manual.

Each file system has a statically allocated number of inodes. An inode is allocated for each **NBPI** bytes of disk space. The inode allocation strategy is extremely conservative.

MAXIPG bounds the number of inodes per cylinder group, and is needed only to keep the structure simpler by having only a single variable size element (the free bit map).

f

Important Note: **MAXIPG** must be a multiple of **INOPB(fs)**.

MINBSIZE is the smallest allowable block size. With a **MINBSIZE** of 4096, it is possible to create files of size 2^{32} with only two levels of indirection. **MINBSIZE** must be big enough to hold a cylinder group block, thus **MINBSIZE** must always be greater than **sizeof(struct cg)**. Note that super blocks are never more than size **SBSIZE**.

The path name on which the file system is mounted is maintained in **fs_fsmnt**. **MAXMNTLEN** defines the amount of space allocated in the super block for this name. The limit on the amount of summary information per file system is defined by **MAXCSBUFS**. It is currently parameterized for a maximum of two million cylinders.

Per cylinder group information is summarized in blocks allocated from the first cylinder group's data blocks. These blocks are read in from **fs_csaddr** (size **fs_cssize**) in addition to the super block.

Important Note: **sizeof (struct csum)** must be a power of two in order for the **fs_cs** macro to work.

The two possible values for **fs_magic** are **FS_MAGIC**, the default magic number for an HFS file system with a fixed-size directory format that limits file name length to **DIRSIZ** (14), and **FS_MAGIC_LFN**, the magic number of a file system using a variable-size directory format that supports file names of up to **MAXNAMLEN** (255) characters in length.

Super Block for a File System:

MAXBPC bounds the size of the rotational layout tables and is limited by the fact that the super block is of size **SBSIZE**. The size of these tables is inversely proportional to the block size of the file system. The size of the tables is increased when sector sizes are not powers of two, as this increases the number of cylinders included before the rotational pattern repeats (**fs_cpc**). The size of the rotational layout tables is derived from the number of bytes remaining in **(struct fs)**.

MAXBPG bounds the number of blocks of data per cylinder group, and is limited by the fact that cylinder groups are, at most, one block. The size of the free block table is derived from the size of blocks and the number of remaining bytes in the cylinder group structure **(struct cg)**.

inode:

The inode is the focus of all file activity in the HP-UX file system. There is a unique inode allocated for each active file, each continuation inode, each current directory, each mounted-on file, text file, and the root. An inode is "named" by its device-and-i-number pair. For the format of an inode and its flags, see *inode*(4).

DEPENDENCIES

Series 700

Series 700 systems support only one section per volume. Thus, there can only be one file system on each volume and the first 8 Kbytes of a file system is the boot area. This area contains the LIF volume header, the directory that defines the contents of the volume, and the bootstrapping program.

AUTHOR

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

SEE ALSO

inode(4), lif(4).

NAME

fs (vxfs) - format of VxFS file system volume

SYNOPSIS

```
#include <sys/param.h>
#include <sys/kern_sem.h>
#include <sys/fs/vx_hpux.h>
#include <sys/fs/vx_port.h>
#include <sys/fs/vx_inode.h>
#include <sys/fs/vx_fs.h>
```

DESCRIPTION

The **vxfs** super-block always begins at byte offset 8192 from the start of the file system. The super-block location is fixed so utilities know where to look for it.

The super-block contains the following fundamental sizes and offsets:

fs_magic
The magic number for the file system (**VX_MAGIC**). This number identifies the file system as being a **vxfs** FSType.

fs_version
The version number of the file system layout (**VX_VERSION**). This is currently 2 for the **vxfs** Version 2 disk layout and 3 for the Version 3 layout.

fs_ctime
The creation date of the file system. The *time*(2) system call supplies the time.

fs_ectime
This field is a placeholder in instances when the creation date for a file system is expanded for more precision. It currently is zero.

fs_logstart
The block address of the first Log Area block. It currently is two.

fs_logend
The block address of the last Log Area block. The Log Area size in blocks may be specified as part of *mkfs*(1M). If not specified, a default of 512 blocks is used. A minimum size of 32 blocks is enforced. For smaller file systems, the default is reduced to avoid wasting space.

fs_bsize
The block size of the file system. The current choices are 1024, 2048, 4096, and 8192 bytes.

fs_size
The number of blocks in the file system, expressed as the number of blocks of size *fs_bsize*. The **fs_size** field is a signed 32 bit number. The maximum number of blocks in a **vxfs** file system is limited to 31 bits.

fs_dsize
The number of data blocks in the file system. A data block is a block which may be allocated to a file in the file system.

fs_ninode
The number of inodes in the file system allocation units. For Version 2 or Version 3 layout file systems, this field is 0.

fs_nau
The number of allocation units in the file system. The number of allocation units may be specified as part of *mkfs*(1M).

fs_defiextsize
The default size for indirect data extents, expressed in blocks. This field is currently set to 64 by default.

fs_oilbsize
The size of an old inode list block, expressed in bytes. For Version 2 and 3 layouts, this is initialized to 0 and not used.

fs_immedlen
The size, in bytes, of the immediate data area in each inode. This value is 96 for the **vxfs** file system.

fs_ndaddr
The number of direct extents supported by the **VX_EXT4** mapping type (see the section describing inode list). This value is 10 for the **vxfs** file system.

The preceding fields define the size and makeup of the file system. To reduce the calculations required in utilities, a number of values are derived from the fundamental values and placed in the super-block.

The super-block contains the following derived offsets:

fs_aufirst
The address, in blocks, of the first allocation unit. There can be a gap between the end of the intent log and the first allocation unit. This gap could be used to align the first allocation unit on a desired boundary.

fs_emap
The offset in blocks of the free extent map (**emap**) from the start of an allocation unit.

fs_imap
The offset in blocks of the free inode map (**imap**) from the start of an allocation unit.

fs_iextop
The offset in blocks of the extended inode operation map from the start of an allocation unit. For Version 2 and 3 layouts, this is initialized to 0 and not used.

fs_istart
The offset in blocks of the inode list (**ilist**) from the start of an allocation unit. For Version 2 and 3 layouts, this is initialized to 0 and not used.

fs_bstart
The offset in blocks of the first data block from the start of an allocation unit. An allocation unit header may contain padding to align the first data block.

fs_femap
The offset in blocks of the first free extent map (**emap**) from the start of the file system.

fs_fimap
The offset in blocks of the first free inode map (**imap**) from the start of the file system.

fs_fiextop
The offset in blocks of the first extended inode operation map from the start of the file system. For Version 2 and 3 layouts, this is initialized to 0 and not used.

fs_fistart
The offset in blocks of the first ilist from the start of the file system. For Version 2 and 3 layouts, this is initialized to 0 and not used.

fs_fbstart
The offset in blocks of the first data block from the start of the file system.

fs_nindir
The number of entries in an indirect address extent. An indirect address extent is currently 8192 bytes in length, making the current value for **fs_nindir** 2048.

fs_aulen
The length of an allocation unit in blocks.

fs_auimlen
The length of a free inode map in blocks. For Version 2 and 3 layouts, this is initialized to 0 and not used.

fs_auemlen
The length of a free extent map in blocks.

fs_auilen
The length, in blocks, of the inode list for this allocation unit. For Version 2 and 3 layouts, this is initialized to 0 and not used.

f

fs_aupad
The length, in blocks, of the allocation unit alignment padding.

fs_aublocks
The number of data blocks in an allocation unit.

fs_maxtier
The log base 2 of *fs_aublocks*.

fs_inopb
The number of inode entries per *fs_bsize* block in the inode list. The **vxfs** inode is currently 256 bytes long.

fs_inopau
The number of inodes in an allocation unit.

fs_inopilb
The number of inode entries per *fs_oilbsize* block in the inode list. For Version 2 and 3 layouts, this field is obsolete.

fs_ndiripau
Expected number of directory inodes per allocation unit. For Version 2 and 3 layouts, this is initialized to 0 and not used.

fs_iaddrlen
The size, in blocks, of an indirect address block. An indirect address block is 8K bytes. This field will be set to (8K / *fs_bsize*).

fs_bshift
The log base 2 of *fs_bsize*. Used to convert a byte offset into a block offset.

fs_inoshift
The log base 2 of *fs_inopb*. Used to convert an inode number into a block offset in the inode list.

fs_bmask
A mask value such that (*byte_offset* & *fs_bmask*) rounds the offset to the nearest smaller block boundary.

fs_boffmask
A mask value such that (*byte_offset* & *fs_boffmask*) yields the offset from the start of the nearest smaller block boundary.

fs_inomask
A mask value such that (*inode_number* & *fs_inomask*) yields the offset from the start of the containing inode list block of the corresponding inode list entry. For Version 2 and 3 layouts, this field is obsolete.

fs_checksum
A simple checksum of the above fields. A macro, **VX_FSCHECKSUM** is provided to verify or calculate the checksum.

The above fields are initialized when the file system is created and do not change unless the file system is resized. These fields are replicated in each allocation unit header.

There are additional fields which are considered to be dynamic:

fs_free
The current number of free data blocks.

fs_ifree
The current number of free inodes. For Version 2 and 3 layouts, a separate free inode count is kept for each fileset; this is initialized to 0 and is not used.

fs_efree
An array of the current number of free extents of each extent size in the file system.

fs_flags
The following flags are recognized:

VX_FULLFSCK
Set when a file system requires a full structural check to recover from an error. If this flag is set, a full check will be performed after the replay recovery is finished.

VX_NOLOG
Set when the file system was mounted with the **VX_MS_NOLOG** option. If this flag is set, then no log replay recovery will be performed.

VX_LOGBAD
Set when an I/O error has invalidated the log. If this flag is set, then no log replay recovery will be performed.

VX_LOGRESET
Set when the log ID runs over **VX_MAXLOGID** (2ˆ30). The log ID will be reset at the next appropriate opportunity (such as a mount or 60-second sync).

VX_RESIZE
Set when a file system resizing is in progress. If an *fsck*(1M) sees this flag, it will have to perform resize recovery. Refer to *fsadm*(1M) for a description of file system expansion.

VX_UPGRADING
Set when a file system upgrade is in progress. If an *fsck*(1M) sees this flag, it will have to perform upgrade recovery.

f

fs_mod
Set whenever a mounted file system is modified. It is used to indicate if the super-block needs to be written when a **sync** operation is performed.

fs_clean
Set to **VX_DIRTY** when a file system is mounted for read/write access. Set to **VX_CLEAN** upon **umount** or successful *fsck*(1M). The file system cannot be mounted for read/write access unless the **fs_clean** field is **VX_CLEAN**.

fs_reserved
Reserved for future use.

fs_firstlogid
Initial log ID to use when the file system is mounted.

fs_time
Last time the super-block was written to disk, indicated as the number of seconds that have elapsed since 00:00 January 1, 1970.

fs_fname
File system name (6 characters).

fs_fpack
File system pack label (6 characters).

fs_logversion
The version number of the log format. This field is set by the kernel on each mount to ensure that an *fsck*(1M) running log replay understands the log format written by the kernel.

The log format may change with each release, so all file systems should be clean before upgrading to a new release.

The read-only area that supports the **vxfs** Version 2 layout has the following fields:

fs_oltext
This is an array of two extent addresses. These extent addresses point to the two replicated copies of the first object location table extent.

fs_oltsize
This is the size, in blocks, of the object location table extents pointed to by **fs_oltext**.

fs_iauimlen
The length, in blocks, of a free inode map in an inode allocation unit.

fs_iausize
The size, in blocks, of an inode allocation unit.

fs_dinosize
The size, in bytes, of a disk inode. This is currently 256 bytes.

fs_checksum2
This is a checksum of the fields specific to the Version 2 layout.

SEE ALSO

fsck(1M), fsdb(1M), inode_vxfs(4), mkfs(1M), mount(2).

f

NAME

fspec - format specification in text files

DESCRIPTION

It is sometimes convenient to maintain text files on the HP-UX system with non-standard tabs, (meaning tabs that are not set at every eighth column). Generally, such files must be converted to a standard format – frequently by replacing all tabs with the appropriate number of spaces – before they can be processed by HP-UX system commands. A format specification occurring in the first line of a text file specifies how tabs are to be expanded in the remainder of the file.

A format specification consists of a sequence of parameters separated by blanks and surrounded by the brackets **<:** and **:>**. Each parameter consists of a keyletter, possibly followed immediately by a value. The following parameters are recognized:

t*tabs* The **t** parameter specifies tab settings for the file. The value of *tabs* must be one of the following:

1. A list of column numbers separated by commas, indicating tabs set at the specified columns;
2. A **-** followed immediately by an integer *n*, indicating tabs at intervals of *n* columns;
3. A **-** followed by the name of a “canned” tab specification.

Standard tabs are specified by **t-8**, or equivalently, **t1**, **9**, **17**, **25**, etc. Recognized canned tabs are defined by the **tabs** command (see *tabs*(1)).

s*size* The **s** parameter specifies a maximum line size. The value of *size* must be an integer. Size checking is performed after tabs have been expanded, but before the margin is inserted at the beginning of the line.

m*margin* The **m** parameter specifies a number of spaces to be inserted at the beginning of each line. The value of *margin* must be an integer.

d The **d** parameter takes no value. Its presence indicates that the line containing the format specification is to be deleted from the converted file.

e The **e** parameter takes no value. Its presence indicates that the current format is to prevail only until another format specification is encountered in the file.

Default values (assumed for parameters not supplied) are **t-8** and **m0**. If the **s** parameter is not specified, no size checking is performed. If the first line of a file does not contain a format specification, the above defaults are assumed for the entire file. The following is an example of a line containing a format specification:

```
* <:t5,10,15 s72:> *
```

If a format specification can be disguised as a comment, it is not necessary to code the **d** parameter.

Several HP-UX system commands correctly interpret the format specification for a file. Among them is **ed**, which can be used to convert files to a standard format acceptable to other HP-UX system commands.

SEE ALSO

ed(1), newform(1), tabs(1).

f

NAME

fstab - static information about the file systems

SYNOPSIS

#include <fstab.h>

DESCRIPTION

fstab is an ASCII file that resides in directory **/etc**. Programs read it, but do not write to or from it. System administrators are responsible for creating and maintaining this file properly.

/etc/fstab contains a list of mountable file-system entries. Each file-system entry appears on a separate line, and consists of fields separated by one or more blanks or tabs.

The order of entries in **/etc/fstab** is important only for entries without a *pass number* field. Entries without a *pass number* are sequentially checked by **fsck** (see *fsck*(1M)) after the entries with a *pass number* have been checked.

f

Each file-system entry must contain a *device special file* and may additionally contain all of the following fields, in the following order:

directory

type

options

backup frequency

pass number (on parallel **fsck**)

comment

If any field after the name of the device special file is present, all fields must be present in the order indicated, to ensure correct place-holding.

Entries from this file are accessed using **getmntent()** (see *getmntent*(3X)).

The fields are separated by white space, and a **#** as the first non-whitespace character in an entry or field indicates a comment.

device special file A block device special file name. This field is used by **fsck**, **mount**, **swapon**, **crashconf**, and other commands to identify the location of the storage device on which the file system resides.

directory Name of the root of the mounted file system that corresponds to the *device special file*. If *type* is **swapfs**, *directory* can be the name of any directory within a file system. Only one directory should be specified per file system. *directory* must already exist and must be given as an absolute path name.

type Can be **swap**, **swapfs**, **dump**, **ignore**, or a file system type (for example, **hfs**, **vxfs**, **cdfs**, **nfs**, or **lofs**).

If *type* is **swap**, the *device special file* is made available as an area of swap space by the **swapon** command (see *swapon*(1M)). The *options* field is valid. The fields *directory*, *pass number*, and *backup frequency* are ignored for **swap** entries.

If *type* is **swapfs**, the file system in which *directory* resides is made available as swap space by **swapon**. The *options* field is valid. The fields *device special file*, *pass number*, and *backup frequency* are ignored for **swapfs** entries.

If *type* is **dump**, the *device special file* is made available as an area into which a system crash dump may occur, by the **crashconf** command (see *crashconf*(1M)). The fields *options*, *directory*, *pass number*, and *backup frequency* are ignored for **dump** entries.

Entries marked by the *type* **ignore** are ignored by all commands and can be used to mark unused sections. If *type* is specified as either **ignore**, **dump**, **swap**, or **swapfs**, the entry is ignored by the **mount** and **fsck** commands (see *mount*(1M) and *fsck*(1M)). *fsck* also ignores entries with *type* specified as **cdfs**, **nfs**, or **lofs**.

options A comma-separated list of option keywords, as found in *mount*(1M) or *swapon*(1M). The keywords used depend on the parameter specified in *type*.

backup frequency Reserved for possible use by future backup utilities.

pass number Used by the **fsck** command to determine the order in which file system checks are done. The root file system should be specified with a *pass number* of 1, to be checked first, and other file systems should have larger numbers. (A file system with a *pass number* of zero is ignored by the **fsck** command.)

File systems within a drive should be assigned different pass numbers, but file systems on different drives can be checked on the same pass, to utilize possible parallelism available in the hardware. If *pass number* is not present, **fsck** checks each such file system sequentially after all eligible file systems with pass numbers have been checked.

comment An optional field that begins with a **#** character and ends with a new-line character. Space from the *pass number* to the *comment* field (if present) or to the new-line is reserved for future use.

There is no limit to the number of *device special file* fields in **/etc/fstab**.

NETWORKING FEATURES

NFS

If the field *type* is **nfs**, a remote NFS file system is implied. For NFS file systems, the *device special file* should be the serving machine name followed by ":" followed by the path on the serving machine of the directory being served. The *pass number* and *backup frequency* fields are ignored for NFS entries.

EXAMPLES

Examples of typical **/etc/fstab** entries:

Add an HFS file system at **/home** using default mount options; (backup frequency 0) fsck pass 2:

```
/dev/dsk/c0t6d0 /home hfs  defaults 0 2 # /home disk
```

Add a **swap** device to a system managed using LVM, with default options (Note, the *directory* field (**/**) cannot be empty, even though it is ignored):

```
/dev/vg01/lv10 / swap defaults 0 0 # swap device
```

Add a swap device on a system implementing whole-disk layout to use the space after the end of the file system (*options*=**end**):

```
/dev/dsk/c0t5d0 / swap end 0 0 # swap at end of device
```

Add file system swap space on the file system containing directory **/swap**. *type* is **swapfs**; set *options* to **min=10**, **lim=4500**, **res=100**, and **pri=0** (see *swapon*(1M)) for explanation of *options*). *device* field is ignored but must not be empty:

```
default /swap swapfs min=10,lim=4500,res=100,pri=0 0 0
```

(Note that both a file system entry and a swap entry are required for devices providing both services.)

Use a device for dump space if the system crashes. *directory* field is ignored but must not be empty:

```
/dev/dsk/c0t5d0 / dump defaults 0 0
```

(Note that both a swap entry and a dump entry are required for devices providing both services.)

DEPENDENCIES

NFS

Here is an example for mounting an NFS file system for systems that support NFS file systems:

```
server:/mnt /mnt nfs rw,hard 0 0 #mount from server.
```

AUTHOR

fstab was developed by HP, AT&T, Sun Microsystems, Inc., and the University of California, Berkeley.

FILES

/etc/fstab
/usr/include/fstab.h

SEE ALSO

fsck(1M), mount(1M), swapon(1M), crashconf(1M), getfsent(3X), getmntent(3X), mnttab(4).

f

NAME

ftpusers - security file for *ftpd*(1M)

DESCRIPTION

ftpd rejects remote logins to local user accounts that are named in **/etc/ftpusers**. Each restricted account name must appear alone on a line in the file. The line cannot contain any white space. User accounts that specify a restricted login shell in **/etc/passwd** should be listed in **/etc/ftpusers** because **ftpd** accesses local accounts without using their login shells. UUCP accounts should be listed in **/etc/ftpusers**. If **/etc/ftpusers** does not exist, **ftpd** skips the security check.

EXAMPLES

Given an **/etc/ftpusers** file containing the following:

```
Only lines that exactly match user account names are
significant.  Blank lines are harmless because they
do not match any account names.  However you must be
careful.

uucp
guest
```

ftpd would reject login attempts using the local accounts **careful.**, **uucp**, or **guest**.

AUTHOR

ftpusers was developed by the University of California, Berkeley.

SEE ALSO

ftp(1), ftpd(1M).

f

g

NAME

gated.config - GateDaemon Configuration Guide

SYNOPSIS

/etc/gated.conf

DESCRIPTION

Configuration Overview

- Introduction
- Statement Summary
- Preferences and Route Selection
- Trace Statements and Global Options
- Directive Statements
- Options Statements
- Interface Statements and Configuration
- Definition Statements

Protocol Statements

- Protocol Overview
- Interior gateway protocols (igps)
  - RIP, HELLO, OSPF
- Exterior gateway protocols (egps)
  - EGP, BGP
- ICMP Statement
- Redirect Statement
- Router Discovery Statement
- Kernel Interface
- Static Routes

Control Statements

- Route filtering
- Matching AS paths
- Route Importation
- Route Exportation
- Route Aggregation

Appendices

- Glossary of Terms
- References

Introduction to Configuring GateD

Syntax

The gated configuration file consists of a sequence of statements terminated by a semi-colon ('**;**'). Statements are composed of tokens separated by white space, which can be any combination of blanks, tabs and newlines. This structure simplifies identification of the parts of the configuration associated with each other and with specific protocols. Comments may be specified in either of two forms. One form begins with a pound sign ('**#**') and runs to the end of the line. The other form, *C* style, starts with a '**/***' and continues until it reaches '***/**'.

Syntax description conventions

Keywords and special characters that the parser expects exactly are displayed using **bold** type. Parameters are displayed in italic *variable definition* style. Parameters shown in square brackets ('**[**' and '**]**') are used to show optional keywords and parameters. The vertical bar ('**|**') is used to indicate between a choice of optional parameters. Parentheses ('**(**' and '**)**') are used to group keywords and parameters when necessary.

For example, in the syntax description:

[**backbone** | (**area** *area*)]

The square brackets say that either parameter is optional. The keywords are **backbone** and **area**. The vertical bar indicates that either "**backbone**" or "**area** *area*" may be specified. Since *area* is in the *variable definition* style, it is a parameter that needs to be provided.

Statement Grouping

The configuration statements and the order in which these statements appear divide **gated.conf** into options statements, interface statements, definition statements, protocol statements, static statements, control statements, and aggregate statements. Entering a statement out of order causes an error when parsing the configuration file.

Two other types of statements do not fit in these categories: %directive statements and %trace statements. These statements provide instructions to the parser and control tracing from the configuration file. They do not relate to the configuration of any protocol and may occur anywhere in the gated.conf file.

Statement Summary

A summary table of the configuration statements (in the configuration statement summary) lists each GateD configuration statement by name, identifies the statement type, and provides a short synopsis of the command function. More detailed definitions and descriptions of each of the eight classes of GateD statements follow in separate sections.

GateD Configuration Statement Summary

The GateD configuration commands are summarized below. The table lists each command by name, identifies the statement type, and gives a synopsis of the statement function:

Summary of GateD Configuration Statements

%directory (directive)	sets the directory for include files.
%include (directive)	includes a file into gated.conf.
traceoptions (trace)	specifies which events are traced.
options (definition)	defines GateD options.
interfaces (definition)	defines GateD interfaces.
autonomoussystem (definition)	defines the AS number.
routerid (definition)	defines the originating router (BGP, OSPF).
martians (definition)	defines invalid destination addresses.
rip (protocol)	enables RIP protocol.
hello (protocol)	enables HELLO protocol.
isis (protocol)	enables ISIS protocol.
kernel (protocol)	configures kernel interface options.
ospf (protocol)	enables OSPF protocol.
egp (protocol)	enables EGP protocol.
bgp (protocol)	enables BGP protocol.
redirect (protocol)	configures the processing of ICMP redirects.
icmp (protocol)	configures the processing of general ICMP packets.
static (static)	defines static routes.
import (control)	defines which routes to import.
export (control)	defines which routes to export.
aggregate (control)	defines which routes to aggregate.

g

generate (control) defines which routes to generate.

Preference

Preference is the value GateD uses to order preference of routes from one protocol or peer over another. Preference can be set in the GateD configuration files in several different configuration statements. Preference can be set based on network interface over another, from one protocol over another, or from one remote gateway over another. Preference may not be used to control the selection of routes within an **igp**, this is accomplished automatically by the protocol based on metric. Preference may be used to select routes from the same **egp** learned from different peers or autonomous systems. Each route has only one preference value associated with it, even though preference can be set at many places in the configuration file. Simply, the last or most specific preference value set for a route is the value used. (See **Glossary of Terms: Preference**.) The preference value is an arbitrarily assigned value used to determine the order of routes to the same destination in a single routing database. The active route is chosen by the lowest preference value. Some protocols implement a second preference (preference2), sometimes referred to as a tie-breaker.

Selecting a route

- The route with the best (numerically smallest) preference is preferred.
- If the two routes have the same preference, the route with the best (numerically smallest) preference2 (also known as a tie-breaker) is preferred.
- A route learned from a **igp** is preferred to a route learned from an **egp**. Least preferred is a route learned indirectly by an **igp** from an **egp**.
- If AS path information is available, it is used to help determine the most preferred route.
  - A route with an AS path is preferred over one without an AS path.
  - If the AS paths and origins are identical, the route with the lower metric is preferred.
  - A route with an AS path origin of **igp** is preferred over a route with an AS path origin of **egp**. Least preferred is an AS path with an **unknown** origin.
  - A route with a shorter AS path is preferred.
- If both routes are from the same protocol and AS, the one with the lowest metric is preferred.
- The route with the lowest numeric next-hop address is used.

Assigning preferences

A default preference is assigned to each source from which GateD receives routes. Preference values range from 0 to 255 with the lowest number indicating the most preferred route.

The following table summarizes the default preference values for routes learned in various ways. The table lists the statements (some of these are clauses within statements) that set preference, and shows the types of routes to which each statement applies. The default preference for each type of route is listed, and the table notes preference precedence between protocols. The narrower the scope of the statement, the higher precedence its preference value is given, but the smaller the set of routes it affects.

Preference Of	Defined by Statement	Default
direct connected networks	interface	0
OSPF routes	ospf	10
IS-IS level 1 routes	isis level 1	15
IS-IS level 2 routes	isis level 2	18
internally generated default	gendefault	20
redirects	redirect	30
routes learned via route socket	kernel	40
static routes from config	static	60
ANS SPF (SLSP) routes	slsp	70
HELLO routes	hello	90
RIP routes	rip	100
point-to-point interface		110
routes to interfaces that are down	interfaces	120
aggregate/generate routes	aggregate/generate	130
OSPF AS external routes	ospf	150
BGP routes	bgp	170
EGP	egp	200

Sample Preference Specifications

```
interfaces {
    interface 138.66.12.2 preference 10 ;
} ;
rip yes {
    preference 90 ;
} ;
import proto rip gateway 138.66.12.1 preference 75 ;
```

In these statements the preference applicable to routes learned via RIP from gateway 138.66.12.1 is 75. The last preference applicable to routes learned via RIP from gateway 128.66.12.1 is defined in the accept statement. The preference applicable to other RIP routes is found in the **rip** statement. The preference set on the interface statement applies only to the route to that interface.

Trace Statements

Trace statements control tracing options. The GateD tracing options may be configured at many levels. Tracing options include the file specifications, control options, and global and protocol specific tracing options. Unless overridden, tracing options from the next higher level are inherited by lower levels. For example, BGP peer tracing options are inherited from BGP group tracing options, which are inherited from global BGP tracing options, which are inherited from global GateD tracing options. At each level tracing specifications override the inherited options.

g

Global tracing options

There are two types of global options, those which only affect global operations and those which have potential significance to protocols.

Global significance only

The trace flags that only have global significance are:

parse Trace the lexical analyzer and parser. Mostly used by GateD developers for debugging.

adv Trace the allocation of and freeing of policy blocks. Mostly used by the GateD developers for debugging.

symbols Used to trace symbols read from the kernel at startup. The only useful way to specify this level of tracing is via the **-t** option on the command line since the symbols are read from the kernel before parsing the configuration file.

iflist Used to trace the reading of the kernel interface list. It is useful to specify this with the **-t** option on the command line since the first interface scan is done before reading the configuration file.

Protocol significance

The options flags that have potential significance to protocols are:

all Turn on all of the following.

general A shorthand notation for specifying both **normal** and **route**.

state Trace state machine transitions in the protocols.

normal Trace normal protocols occurrences. Abnormal protocol occurrences are always traced.

policy Trace application of protocol and user-specified policy to routes being imported and exported.

task Trace system interface and processing associated with this protocol or peer.

timer Trace timer usage by this protocol or peer.

route Trace routing table changes for routes installed by this protocol or peer.

Not all of the above options apply to all of the protocols. In some cases their use does not make sense (for instance, RIP does not have a state machine) and in some instances the requested tracing has not been implemented (such as RIP support of the **policy** option).

It is not currently possible to specify packet tracing from the command line. This is because a global option for packet tracing would potentially create too much output.

When protocols inherit their tracing options from the global tracing options, tracing levels that do not make sense (such as **parse**, **adv** and packet tracing options) are masked out.

Global tracing statements have an immediate effect, especially parsing options that effect the parsing of the configuration file. Tracing values inherited by protocols specified in the configuration file are initially inherited from the global options in effect as they are parsed, unless they are overridden by more specific options. After the configuration file is read, tracing options that were not explicitly specified are inherited from the global options in effect at the end of the configuration file.

Packet tracing

Tracing of packets is very flexible. For any given protocol there are one or more options for tracing packets. all protocols allow use of the **packets** keyword allows for tracing *all* packets sent and received by the protocol. most protocols have other options for limiting tracing to a useful subset of packet types. These tracing options can be further controlled with the following modifiers:

detail **detail** must be specified before **send** or **recv**. Normally packets are traced in a terse form of one or two lines. When **detail** is specified, a more verbose format is used to provide further detail on the contents of the packet.

send
recv These options limit the tracing to packets sent or received. Without these options both sent and received packets will be traced.

Detail, if specified, must be before **send** or **recv**. If a protocol allows for several different types of packet tracing, modifiers may be applied to each individual type. But be aware that within one tracing specification the trace flags are summed up, so specifying **detail packets** will turn on full tracing for all packets.

Traceoptions syntax

traceoptions ["*trace_file*" [**replace**] [**size** *size*[**k** | **m**] **files** *files*]]
[*control_options*] *trace_options* [**except** *trace_options*] ;

traceoptions none ;

trace_file Specifies the file to receive tracing information. If this file name does not begin with a slash **(/)**, the directory where gated was started in prepended to the name.

replace Tracing should start by replacing an existing file. The default is to append to an existing file.

size *size*[**k** | **m**] **files** *files*
Limit the maximum size of the trace file to the specified size (minimum 10k). When the trace file reaches the specified size, it is renamed to **file.0**, then **file.1**, **file.2** up to the maximum number of files (minimum specification is 2).

control_options
Specifies options that control the appearance of tracing. Valid values are:

nostamp
Specifies that a timestamp should not be prepended to all trace lines.

except *trace_options*
Used to enable a broad class of tracing and then disable more specific options.

none Specifies that all tracing should be turned off for this protocol or peer.

Directive Statements

Directive statements provide direction to the GateD configuration language parser about included files and the directories in which these files reside. Directive statements are immediately acted upon by the parser. Other statements terminate with a semi-colon (;), but directive statements terminate with a newline. The two directive statements are:

%directory "*directory*"
Defines the directory where the include files are stored. When it is used, GateD looks in the directory identified by pathname for any included files that do not have a fully qualified filename, such as files that do not begin with "/". This statement does not actually change the current the directory, it just specifies the prefix applied to included file names.

g

%include "*filename*"

Identifies an include file. The contents of the file is *included* in the gated.conf file at the point in the gated.conf file where the **%include** directive is encountered. If the filename is not fully qualified (does not begin with "/"), it is considered to be relative to the directory defined in the **%directory** directive. The **%include** directive statement causes the specified file to be parsed completely before resuming with this file. Nesting up to ten levels is supported. The maximum nesting level may be increased by changing the definition of **FI_MAX** in **parse.h**.

In a complex environment, segmenting a large configuration into smaller more easily understood segments might be helpful, but one of the great advantages of GateD is that it combines the configuration of several different routing protocols into a single file. Segmenting a small file unnecessarily complicates routing configurations.

Options Statements

Options statements allow specification of some global options. If used, **options** must appear before any other type of configuration statement in the gated.conf file.

The options statement syntax is:

options
[**nosend**]
[**noresolv**]
[**gendefault** [**preference** *preference*] [**gateway** *gateway*]]
[**syslog** [**upto**] *log_level*]
[**mark** *time*]
;

The options list can contain one or more of the following options:

gendefault [**preference** *preference*] [**gateway** *gateway*]

When **gendefault** is enabled, when a BGP or EGP neighbor is up it causes the creation of a default route with the special protocol **default**. This can be disabled per BGP/EGP group with the **nogendefault** option. By default, this route has a preference of 20. This route is normally not installed in the kernel forwarding table, it is only present so it can be announced to other protocols. If a gateway is specified, the default route will be installed in the kernel forwarding table with a next hop of the listed gateway.

Note that the use of the more general option is preferred to the use of this **gendefault** option. The **gendefault** option may go away in future releases. The the section on Route Aggregation for more information on the **generate** statement.

nosend

Do not send any packets. This option makes it possible to run GateD on a live network to test protocol interactions without actually participating in the routing protocols. The packet traces in the GateD log can be examined to verify that GateD is functioning properly. This is most useful for RIP and HELLO and possibly the SMUX SNMP interface. This option does not yet apply to BGP and is less than useful with EGP and OSPF.

noresolv

By default GateD will try to resolve symbolic names into IP addresses by using the gethostbyname() and getnetbyname() library calls. These calls usually use the Domain Name System (DNS) instead of the hosts local host and network tables. If there is insufficient routing information to send DNS queries, GateD will deadlock during startup. This option can be used to prevent these calls, symbolic names will result in configuration file errors.

syslog [**upto**] *log_level*

Controls the amount of data GateD logs via syslog on systems where setlogmask() is supported. The available logging levels and other terminology are as defined in the **setlogmask(3)** man page. The default is equivalent to **syslog upto info**.

mark *time*

Specifying this option causes gated to output a message to the trace log at the specified interval. This can be used as one method of determining if gated is still running.

Interfaces Statement

Interface Syntax

interfaces {
options

g

[**strictinterfaces**]
[**scaninterval** *time*]
;
interface *interface_list*
[**preference** *preference*]
[**down preference** *preference*]
[**passive**]
[**simplex**]
[**reject**]
[**blackhole**]
;
define *address*
[**broadcast** *address*] | [**pointtopoint** *address*]
[**netmask** *mask*]
[**multicast**]
;
} ;

g

An interface is the connection between a router and one of its attached networks. A physical interface may be specified by interface name, by IP address, or by domain name, (unless the network is an unnumbered point-to-point network.) Multiple levels of reference in the configuration language allow identification of interfaces using wildcard, interface type name, or delete word address. Be careful with the use of interface names as future Unix operating systems may allow more than one address per interface. The interface_list is a list of one or more interface names including wildcard names (names without a number) and names which may specify more than one interface or address, or the token **all** for all interfaces.

options
Allows configuration of some global options related to interfaces. These are:

strictinterfaces
Indicates that it is a fatal error to reference an interface in the configuration file that is not present when GateD is started and not listed in a **define** statement. Without this option a warning message will be issued but GateD will continue.

scaninterval *time*
Specifies how often GateD scans the kernel interface list for changes. The default is every 15 seconds on most systems, and 60 seconds on systems that pass interface status changes through the routing socket (BSD 4.4). Note that GateD will also scan the interface list on receipt of a **SIGUSR2**.

interface *interface_list*
Sets interface options on the specified interfaces. An interface list is **all** or a list of interface names (see warning about interface names), domain names, or numeric addresses. Options available on this statement are:

preference *preference*
Sets the preference for routes to this interface when it is up and appears to be functioning properly. The default preference is **0**.

down preference *preference*
Sets the preference for routes to this interface when GateD does not believe it to be functioning properly, but the kernel does not indicate it is down. The default value is **120**.

passive
Prevents GateD from changing the preference of the route to this interface if it is not believed to be functioning properly due to lack of received routing information. GateD will only perform this check if the interface is actively participating in a routing protocol.

simplex
Defines an interface as unable to hear its own broadcast packets. Some systems define an interface as simplex with the IFF_SIMPLEX flag, on others it needs to be specified in the configuration file. On simplex interfaces, packets from myself are assumed to have been looped back in software and are not used as an indication that the interface is functioning properly.

reject
Specifies that the address of the interface which matches these criteria will be used as the

local address when installing *reject* routes in the kernel. Should only be used with systems based on BSD 4.3 Tahoe or earlier which have installed a reject/blackhole pseudo interface.

blackhole
Specifies that the address of the interface which matches these criteria will be used as the local address when installing *reject* routes in the kernel. Should only be used with systems based on BSD 4.3 Tahoe or earlier which have installed a reject/blackhole pseudo interface.

define *address*
Defines interfaces that might not be present when GateD is started so they may be referenced in the configuration file when **strictinterfaces** is defined. Possible **define** keywords are:

broadcast *address*
Defines the interface as broadcast capable (Ethernet or Token Ring) and specifies the broadcast address.

pointopoint *address*
Defines the interface as a point-to-point interface (SLIP or PPP) and specifies the address on the local side. The first *address* on the **define** statement references the address of the host on the **remote** end of the interface, the *address* specified after this **pointopoint** keyword defines the address on the **local** side of the interface.

An interface not defined as broadcast or point-to-point is assumed to be non-broadcast multiaccess (NBMA), such as an X.25 network.

netmask *mask*
Specifies the subnetmask to be used on this interface. This is ignored on pointtopoint interfaces.

multicast
Specifies that the interface is multicast capable.

g

Interface lists

An interface list is a list of references to interfaces or groups of interfaces. There are four methods available for referring to interfaces. They are listed here from most general to most specific.

all This refers to all available interfaces.

Interface name wildcard
This refers to all the interfaces of the same type. Unix interfaces consist of the name of the device driver, like **ie**, and a unit number, like **0**, **5** or **22**. Reference to the name contain only alphabetic characters and match any interfaces that have the same alphabetic part.

For example, **ie** on a Sun would refer to all Interlan Ethernet interfaces, **le** would refer to all Lance Ethernet interfaces. But **ie** would not match **iel0**.

Interface name
This refers to a specific interface, usually one physical interface. These are specified as an alphabetic part followed by a numeric part. This will match one specific interface. But be aware that on many systems, there can be more than one protocol (*IP*) address on a given physical interface. For example, **ef1** will match an interface named **ef1**, but not an interface named **ef10**.

Interface address
This matches one specific interface. The reference can be by protocol address (*10.0.0.51*), or by symbolic hostname (*nic.ddn.mil*). Note that a symbolic hostname reference is only valid when it resolves to only one address. Use of symbolic hostnames is not recommended.

If many interface lists are present in the configuration file with more than one parameter, these parameters are collected at run-time to create the specific parameter list for a given interface. If the same parameter is specified on more than one list, the parameters with the most specific interface is used.

For example, consider a system with three interfaces, **le0**, **le1** and **du0**.

```
rip yes {
    interface all noripin noripout ;
    interface le ripin ;
    interface le1 ripout ;
} ;
```

RIP packets would only be accepted from interfaces **le0** and **le1**, but not from **du0**. RIP packets would only be sent on interface **le1**.

IP Interface addresses and routes

The *BSD 4.3* and later networking implementations allow four types of interfaces. Some implementations allow multiple protocol addresses per physical interface, these are mostly based on *BSD 4.3 Reno* or later.

loopback
: This interface must have the address of **127.0.0.1**. Packets sent to this interface are sent back to the originator. This interface is also used as a catch all interface for implementing other features, such as *reject* and *blackhole* routes. Although a netmask is reported on this interface, it is ignored. It is useful to assign an additional address to this interface that is the same as the OSPF or BGP *router id*; this allows routing to a system based on the *router id* which will work if some interfaces are down.

broadcast
: This is a multi-access interface capable of a physical level broadcast, such as *Ethernet*, *Token Ring* and *FDDI*. This interface has an associated subnet mask and broadcast address. The interface route to an *broadcast* network will be a route to the complete subnet.

g

point-to-point
: This is a *tunnel* to another host, usually on some sort of *serial* link. This interface has a *local* address, and a *remote* address. Although it may be possible to specify multiple addresses for a *point-to-point* interface, there does not seem to be a useful reason for doing so.

The *remote* address must be unique among all the interface addresses on a given router. The *local* address may be shared among many *point-to-point* and up to one non-*point-to-point* interface. This is technically a form of the *router id* method for addressless links. This technique conserves subnets as none are required when using this technique.

If a subnet mask is specified on a *point-to-point* interface, it is only used by RIP version 1 and HELLO to determine which subnets may be propagated to the router on the other side of this interface.

non-broadcast multi-access or **nbma**
: This type of interface is multi-access, but not capable of broadcast. And example would be *frame relay* and *X.25*. This type of interface has a local address and a subnet mask.

GateD insures that there is a route available to each IP interface that is configured and up. Normally this this done by the *ifconfig* command that configures the interface; GateD does it to insure consistency.

For *point-to-point* interfaces, gated installs some special routes. If the *local* address on one or more *point-to-point* interfaces is not shared with a non-*point-to-point* interface, gated installs a route to the *local* address pointing at the *loopback* interface with a preference of 110. This insures that packets originating on this host destined for this *local* address are handled locally. OSPF prefers to route packets for the *local* interface across the *point-to-point* link where they will be returned by the router on the remote end. This is used to verify operation of the link. Since OSPF installs routes with a preference of 10, these routes will override the route installed with a preference of 110.

If the *local* address of one or more *point-to-point* interfaces is shared with a non-*point-to-point* interface, gated installs a route to the *local* with a preference of 0 that will not be installed in the forwarding table. This is to prevent protocols like OSPF from routing packets to this address across a serial interface when this system could be functioning as a *host*.

When the status of an interface changes, GateD notifies all the protocols, which take the appropriate action. GateD assumes that interfaces which are not marked *UP* do not exist. While this might not be the most correct action, it is the way things currently work.

GateD ignores any interfaces that have invalid data for the *local*, *remote* or *broadcast* addresses or the *subnet mask*. Invalid data includes zeros in any field. GateD will also ignore any *point-to-point* interface that has the same local and remote addresses, it assumes it is in some sort of *loopback test* mode.

Definition Statements

Definition statements are general configuration statements that relate to all of GateD or at least to more than one protocol. The three definition statements are **autonomoussystem**, **routerid** and **martians**. if used, **autonomoussystem**, **routerid** and **martians** must appear before any other type of configuration statement in gated.conf file.

Autonomous System configuration

autonomoussystem *autonomous_system* [**loops** *number*] **;**

Sets the autonomous system number of this router to be **autonomous system**. This option is required if BGP or EGP are in use. The AS number is assigned by the Network Information Center (NIC).

Loops is only for protocols supporting AS paths, such as BGP. It controls the number of times this autonomous system may appear in an AS path and defaults to 1 (one).

Router ID configuration

routerid *host* **;**

Sets the router identifier for use by the BGP and OSPF protocols. The default is the address of the first interface encountered by GateD. The address of a non-point-to-point interface is preferred over the local address of a point-to-point interface and an address on a loopback interface that is not the loopback address (127.0.0.1) is most preferred.

Martian configuration

martians {
  host *host* [**allow**] **;**
    network [**allow**] **;**
    network **mask** *mask* [**allow**] **;**
    network **masklen** *number* [**allow**] **;**
  default [**allow**] **;**
} ;

Defines a list of *martian* addresses about which all routing information is ignored. Sometimes a misconfigured system sends out obviously invalid destination addresses. These invalid addresses, called martians, are rejected by the routing software. This command allows additions to the list of martian addresses. See the section on Route Filtering for more information on specifying ranges. Also, the **allow** parameter may be specified to explicitly allow a subset of a range that was disallowed.

Sample Definition Statements

```
options gendefault ;
autonomoussystem 249 ;
interface 128.66.12.2 passive ;
martians {
    0.0.0.26
    };
```

The statements in the sample perform the following functions:

- The options statement tells the system to generate a default route when it peers with an EGP or BGP neighbor.
- The autonomoussystem statement tells GateD to use AS number 249 for in EGP and BGP.
- The interface statement tells GateD not to mark interface 128.66.12.2 as down even if it sees no traffic.
- The martians statement prevents routes to 0.0.0.26 from ever being accepted.

Protocol Overview

Routing protocols determine the "best" route to each destination and distribute routing information among the systems on a network. Routing protocols are divided into two general groups: interior and exterior protocols. GateD software combines management of the interior and exterior routing protocols in one software daemon.

Interior Routing Protocols

Interior protocols are used to exchange reachability information within an autonomous system (AS). They are referred to as a class by the acronym **igp**. There are several interior protocols:

RIP The Routing Information Protocol, Version 1 and Version 2, is the most commonly used interior protocol. RIP selects the route with the lowest metric as the best route. The metric is a hop count representing the number of gateways through which data must pass to reach its destination. The longest path that RIP accepts is 15 hops. If the metric is greater than 15, a destination is considered unreachable and GateD discards the route. RIP assumes the best route is the one that uses the fewest gateways which is the shortest path, not taking into account congestion or delay on route.

g

The RIP version 1 protocol is described in RFC 1058 and the RIP version 2 protocol is described in RFC 1388.

HELLO

HELLO , another interior protocol, uses delay as the deciding factor in choosing the best route. Round-trip time is the length of time it takes a datagram to travel from the source and destination. HELLO is historically significant for the Internet as it was the protocol used among the original prototype NSFNET backbone *fuzzball* gateways. Today, like fuzzballs, HELLO is a little-used protocol.

An earlier version of the HELLO protocol is described in RFC 891.

OSPF

Open Shortest Path First is a link-state protocol. OSPF is better suited than RIP for complex networks with many routers. OSPF provides equal cost multipath routing.

OSPF is described in RFC 1583, the MIB is defined in RFC 1253. Other related documents are RFC 1245, RFC 1246 and RFC 1370.

g

Exterior Routing Protocols

Exterior protocols are used to exchange routing information between autonomous systems. Exterior protocols are only required when an autonomous system must exchange routing information with another autonomous system. Routers within an autonomous system run an interior routing protocol like RIP. Only those gateways that connect an autonomous system to another autonomous system need to run an exterior routing protocol. There are two exterior protocols currently supported by GateD:

EGP

Exterior Gateway Protocol: Originally EGP reachability information was passed into ARPANET/MILNET "core" gateways where the best routes were chosen and passed back out to all connected autonomous systems. As the Internet moved toward a less hierarchical architecture, EGP, an exterior routing protocol which assumes a hierarchical structure, became less effective.

The EGP protocol is described in RFC 827 and RFC 904.

BGP

Border Gateway Protocol is replacing EGP as the exterior protocol of choice. BGP exchanges reachability information between autonomous systems, but provides more capabilities than EGP. BGP uses path attributes to provide more information about each route as an aid in selecting the best route. Path attributes may include, for example, administrative preferences based on political, organizational, or security (policy) considerations in the routing decision. BGP supports nonhierarchical topologies and can be used to implement a network structure of equivalent autonomous systems.

BGP version 1 is described in RFC 1105, version 2 in RFC 1163, version 3 in RFC 1267. The version 3 MIB is described in RFC 1269. The two documents, RFC 1164 and RFC 1268 describe the application of version 2 and three in the internet. A protocol analysis of and experience with BGP version 3 are available in RFC 1265 and RFC 1266. RFC 1397 talks about advertising a *default* route in BGP version 2 and 3. And finally, RFC 1403 describes BGP - OSPF interaction.

Other Routing Protocols

Router Discovery

The Router Discovery protocol is used to inform hosts of the availability of hosts it can send packets to and is used to supplement a statically configured default router. This is the preferred protocol for hosts to run, they are discouraged from *wiretapping* routing protocols.

Router Discovery is described in RFC 1256.

Routing Information Protocol (RIP)

One of the most widely used interior gateway protocols is the Routing Information Protocol (RIP). RIP is an implementation of a distance-vector, or Bellman-Ford routing protocol for local networks. It classifies routers as active and passive (silent). Active routers advertise their routes (reachability information) to others; passive routers listen and update their routes based on advertisements, but do not advertise. Typically, routers run RIP in active mode, while hosts use passive mode.

A router running RIP in active mode broadcasts updates at set intervals. Each update contains paired values where each pair consists of an IP network address and an integer distance to that network. RIP uses a hop count metric to measure the distance to a destination. In the RIP metric, a router advertises directly connected networks at a metric of 1. Networks which are reachable through one other gateway are two hops etc. Thus, the number of hops or hop count along a path from a given source to a given destination refers to the number of gateways that a datagram would encounter along that path. Using hop counts to calculate shortest paths does not always produce optimal results. For example, a path with hop count 3 that crosses three Ethernets may be substantially faster that a path with a hop count 2 that crosses two slow-speed serial lines. To compensate for differences in technology many routers advertise artificially high hop counts for slow links.

As delivered with most UNIX systems, RIP is run by the routing daemon, **routed** (pronounced route-"d"). A RIP routing daemon dynamically builds on information received through RIP updates. When started up, it issues a REQUEST for routing information and then listens for responses to the request. If a system configured to supply RIP hears the request, it responds with a RESPONSE packet based on information in its routing database. The RESPONSE packet contains destination network addresses and the routing metric for each destination.

When a RIP RESPONSE packet is received, the routing daemon takes the information and rebuilds the routing database adding new routes and "better" (lower metric) routes to destinations already listed in the database. RIP also deletes routes from the database if the next router to that destination says the route contains more than 15 hops, or if the route is deleted. All routes through a gateway are deleted if no updates are received from that gateway for a specified time period. In general, routing updates are issued every 30 seconds. In many implementations, if a gateway is not heard from for 180 seconds, all routes from that gateway are deleted from the routing database. This 180 second interval also applies to deletion of specific routes.

RIP version 2 (more commonly known as RIP II) add additional capabilities to RIP. Some of these capabilities are compatible with RIP I and some are not. To avoid supplying information to RIP I routes that could be mis-interpreted, RIP II can only use non-compatible features when its packets are multicast. On interfaces that are not capable of IP multicast, RIP I compatible packets are used that do not contain potentially confusing information.

Some of the most notable RIP II enhancements are:

Next hop

The primary ones are the ability to advertise a next hop to use other than the router supplying the routing update. This is quite useful when advertising a static route to a *dumb* rotuer that does not run RIP as it avoids having packets destined through the dumb router from having to cross a network twice.

RIP I routers will ignore next hop information in RIP II packets. This may result in packets crossing a network twice, which is exactly what happens with RIP I. So this information is provided in RIP I compatible RIP II packets.

Network Mask

RIP I assumes that all subnetworks of a given network have the same network mask. It uses this assumption to calculate the network masks for all routes received. This assumption prevents subnets with different netmasks from being included in RIP packets. RIP II adds the ability to explicitly specify the network mask with each network in a packet.

While RIP I routers will ignore the network mask in RIP II packets, their calculation of the network mask will quite possibly be wrong. For this reason, RIP I compatible RIP II packets must not contain networks that would be mis-interpreted. These network must only be provided in native RIP II packets that are multicast.

Authentication

RIP II packets may also contain one of two types of authentication string that may be used to verify the validity of the supplied routing data. Authentication may be used in RIP I compatible RIP II packets, but be aware that RIP I routers will ignore it.

The first method is a simple password in which an authentication key of up to 16 characters is included in the packet. If this does not match what is expected, the packet will be discarded. This method provides very little security as it is possible to learn the authentication key by watching RIP packets.

The second method is still experimental and may change in incompatible ways in future releases. This method uses the MD5 algorithm to create a crypto-checksum of a RIP packet and an authentication key of up to 16 characters. The transmitted packet does not contain the authentication key itself, instead it contains a crypto-checksum, called the *digest*. The receiving router will perform a calculation using the correct authentication key and discard the packet if the digest does not match. In addition, a sequence number is maintained to prevent the replay of older packets. This method provides a much stronger assurance that routing data originated from a router with a valid authentication key.

Two authentication methods can be specified per interface. Packets are always sent using the *primary* method, but received packets are checked with both the *primary* and *secondary* methods before being discarded. In addition, a separate authentication key is used for non-router queries.

RIP-I and network masks

RIP-I derives the network mask of received networks and hosts from the network mask of the interface the packet via which the packet was received. If a received network or host is on the same natural network as the interface over which it was received and that network is subnetted (the specified mask is more specific than the *natural* netmask), the subnet mask is applied to the destination. If bits outside the mask are set, it is assumed to be a host. Otherwise it is assumed to be a subnet.

On point-to-point interfaces, the netmask is applied to the remote address. The netmask on these interfaces is ignored if it matches the natural network of the remote address or is all ones.

Unlike in previous releases, the zero subnet mask (a network that matches the natural network of the interface, but has a more specific, or longer, network mask) is ignored. If this is not desirable, a route filter may be used to reject it.

g

The RIP Statement

rip yes | **no** | **on** | **off** [**{**
  broadcast ;
  nobroadcast ;
  nocheckzero ;
  preference *preference* **;**
  defaultmetric *metric* **;**
  query authentication [**none** | [[**simple**|**md5**] *password*]] **;**
  interface *interface_list*
    [**noripin**] | [**ripin**]
    [**noripout**] | [**ripout**]
    [**metricin** *metric*]
    [**metricout** *metric*]
    [**version 1**]|[**version 2** [**multicast**|**broadcast**]]
    [[**secondary**] **authentication** [**none** | [[**simple**|**md5**] *password*]] **;**
  trustedgateways *gateway_list* **;**
  sourcegateways *gateway_list* **;**
  traceoptions *trace_options* **;**
}] **;**

The `rip` statement enables or disables RIP. If the `rip` statement is not specified, the default is `rip on ;`. If enabled, RIP will assume `nobroadcast` when there is only one interface and `broadcast` when there is more than one.

The options are as follows:

broadcast
: Specifies that RIP packets will be broadcast regardless of the number of interfaces present. This is useful when propagating static routes or routes learned from anther protocol into RIP. In some cases, the use of `broadcast` when only one network interface is present can cause data packets to traverse a single network twice.

nobroadcast
: Specifies that RIP packets will not be broadcast on attached interfaces, even if there are more than one. If a `sourcegateways` clause is present, routes will still be unicast directly to that gateway.

nocheckzero
: Specifies that RIP should not make sure that reserved fields in incoming version 1 RIP packets are zero. Normally RIP will reject packets where the reserved fields are zero.

preference *preference*
: Sets the preference for routes learned from RIP. The default preference is 100. This preference may be overridden by a preference specified in import policy.

defaultmetric *metric*
: Defines the metric used when advertising routes via RIP that were learned from other protocols. If not specified, the default value is 16 (unreachable). This choice of values requires you to explicitly specify a metric in order to export routes from other protocols into RIP. This metric may be

overridden by a metric specified in export policy.

query authentication [**none** | [[**simple** | **md5**] *password*]] **;**

Specifies the authentication required of query packets that do not originate from routers. The default is *none*.

interface *interface_list*

Controls various attributes of sending RIP on specific interfaces. See the section on interface list specification for the description of the *interface_list*.

Note that if there are multiple interfaces configured on the same subnet, RIP updates will only be sent from first one one which RIP output is configured. This limitation is required because of the way the Unix kernel operates. It will hopefully be removed in a future release.

The possible parameters are:

noripin

Specifies that RIP packets received via the specified interface will be ignored. The default is to listen to RIP packets on all non-loopback interfaces.

ripin

This is the default. This argument may be necessary when **noripin** is used on a wildcard interface descriptor.

noripout

Specifies that no RIP packets will be sent on the specified interfaces. The default is to send RIP on all broadcast and non-broadcast interfaces when in **broadcast** mode. The sending of RIP on point-to-point interfaces must be manually configured.

ripout

This is the default. This argument is necessary when it is desired to send RIP on point-to-point interfaces and may be necessary when **noripin** is used on a wildcard interface descriptor.

metricin *metric*

Specifies the RIP metric to add to incoming routes before they are installed in the routing table. The default is the kernel interface metric plus 1 (which is the default RIP hop count). If this value is specified, it will be used as the absolute value. The kernel metric will not be added. This option is used to make this router prefer RIP routes learned via the specified interface(s) less than RIP routes from other interfaces.

metricout *metric*

Specifies the RIP metric to be added to routes that are send via the specified interface(s). The default is zero. This option is used to make other routers prefer other sources of RIP routes over this router.

version 1

Specifies that RIP packets send via the specified interface(s) will be version 1 packets. This is the default.

version 2

Specifies that RIP version 2 packets will be sent on the specified interfaces(s). If IP multicast support is available on this interface, the default is to send full version 2 packets. If it is not available, version 1 compatible version 2 packets will be sent.

multicast

Specifies that RIP version 2 packets should be multicast on this interface. This is the default.

broadcast

Specifies that RIP version 1 compatible version 2 packets should be broadcast on this interface, even if IP multicast is available.

[**secondary**] **authentication** [**none** | [**simple** | **md5**] *password*]

This defines the authentication type to use. It applies only to RIP version 2 and is ignored for RIP-1 packets. The default authentication type is **none**. If a password is specified, the authentication type defaults to **simple**. The password should be a quoted string with between 0 and 16 characters.

g

If **secondary** is specified, this defines the secondary authentication. If omitted, the primary authentication is specified. The default is primary authentication of *none* and no secondary authentication.

trustedgateways *gateway_list*
Defines the list of gateways from which RIP will accept updates. The *gateway_list* is simply a list of host names or IP addresses. By default, all routers on the shared network are trusted to supply routing information. But if the **trustedgateways** clause is specified only updates from the gateways in the list are accepted.

sourcegateways *gateway_list*
Defines a list of routers to which RIP sends packets directly, not through multicast or broadcast. This can be used to send different routing information to specific gateways. Updates to gateways in this list are not affected by **noripout** on the interface.

traceoptions *trace_options*
Specifies the tracing options for RIP. (See Trace Statements and the RIP specific tracing options below.)

Tracing options

The **policy** option logs info whenever a new route is announce, the metric being announced changes or a route goes or leaves holddown.

Packet tracing options (which may be modified with **detail**, **send** or **recv**):

packets All RIP packets.

request RIP information request packets, such as **REQUEST**, **POLL** and **POLLENTRY**

response
RIP **RESPONSE** packets, which is the type of packet that actually contains routing information.

other Any other type of packet. The only valid ones are **TRACE_ON** and **TRACE_OFF** both of which are ignored.

The Hello Protocol

It is really better not to use HELLO unless you have a specific need for it. We plan to drop it some time around GateD 4.0.

The HELLO protocol is an interior protocol that uses a routing metric based on the length of time it takes a packet to make the trip between the source and the destination. HELLO packets carry timestamp information which allows receivers to compute the shortest delay paths to destinations. The "best" route is the route with the shortest time delay. The unit of time used in HELLO is milliseconds. If a HELLO update packet takes less than 100 milliseconds to travel between two routers, a minimum value of 100 is used for that hop. Thus on networks built of high-speed interfaces HELLO essentially defaults to using hop counts. As in any routing algorithm, HELLO cannot change routes too rapidly or it would be unstable. To avoid instabilities, implementations of HELLO build in hysteresis and "hesitate" to change routes until they have confidence that the change will be lasting.

By default HELLO, like RIP, uses the kernel interface metric set by the **ifconfig** command to influence metric added to routes as they are installed in the routing table (**metricin**). Since the kernel interface metric is in hops, it must be translated into HELLOs millisecond metric. In order to do that, the following table is used:

Hops	HELLO metric
0	0
1	100
2	148
3	219
4	325
5	481
6	713
7	1057
8	1567
9	2322
10	3440
11	5097

12	**7552**
13	**11190**
14	**16579**
15	**24564**
16	**30000**

HELLO and network masks

HELLO derives the network mask of received networks and hosts from the network mask of the interface the packet via which the packet was received. If a received network or host is on the same natural network as the interface over which it was received and that network is subnetted (the specified mask is more specific than the *natural* netmask), the subnet mask is applied to the destination. If bits outside the mask are set, it is assumed to be a host. Otherwise it is assumed to be a subnet.

On point-to-point interfaces, the netmask is applied to the remote address. The netmask on these interfaces is ignored if it matches the natural network of the remote address or is all ones.

Unlike in previous releases, the zero subnet mask (a network that matches the natural network of the interface, but has a more specific, or longer, network mask) is ignored. If this is not desirable, a route filter may be used to reject it.

g

The Hello Statement

hello yes | **no** | **on** | **off** [**{**
  broadcast ;
  nobroadcast ;
  preference *preference* **;**
  defaultmetric *metric* **;**
  interface *interface_list*
    [**nohelloin**] | [**helloin**]
    [**nohelloout**] | [**helloout**]
    [**metricin** *metric*]
    [**metricout** *metric*] **;**
  trustedgateways *gateway_list* **;**
  sourcegateways *gateway_list* **;**
  traceoptions *trace_options* **;**
}] **;**

the **hello** statement enables or disables HELLO. If the **hello** statement is not specified, the default is **hello off**. If enabled, HELLO will assume **nobroadcast** when there is only one interface and **broadcast** when there is more than one interface.

broadcast
Specifies that HELLO packets will be broadcast regardless of the number of interfaces present. This is useful when propagating static routes or routes learned from anther protocol into HELLO. In some cases, the use of **broadcast** when only one network interface is present can cause data packets to traverse a single network twice.

nobroadcast
Specifies that HELLO packets will not be broadcast on attached interfaces, even if there are more than one. If a **sourcegateways** clause is present, routes will still be unicast directly to that gateway.

preference *preference*
Sets the preference for routes learned from HELLO. The default preference is 90. This preference may be overridden by a preference specified in import policy.

defaultmetric *metric*
Defines the metric used when advertising routes via HELLO that were learned from other protocols. If not specified, the default value is 30000 (unreachable). This choice of values requires you to explicitly specify a metric in order to export routes from other protocols into HELLO. This metric may be overridden by a metric specified in export policy.

interface *interface_list*
Controls various attributes of sending HELLO on specific interfaces. See the section on interface list specification for the description of the *interface_list*.

Note that if there are multiple interfaces configured on the same subnet, HELLO updates will only be sent from first one one which HELLO output is configured. This limitation is required because of the way the Unix kernel operates. It will hopefully be removed in a future release.

The possible parameters are:

nohelloin
Specifies that HELLO packets received via the specified interface will be ignored. The default is to listen to HELLO on all non-loopback interfaces.

helloin
This is the default. This argument may be necessary when **nohelloin** is used on a wildcard interface descriptor.

nohelloout
Specifies that no HELLO packets will be sent on the specified interfaces. The default is to send HELLO on all broadcast and non-broadcast interfaces when in **broadcast** mode. The sending of HELLO on point-to-point interfaces must be manually configured.

helloout
This is the default. This argument is necessary when it is desired to send HELLO on point-to-point interfaces and may be necessary when **nohelloin** is used on a wildcard interface descriptor.

g

metricin *metric*
Specifies the HELLO metric to add to incoming routes before they are installed in the routing table. The default is the kernel interface metric plus 1 (which is the default HELLO hop count). If this value is specified, it will be used as the absolute value. The kernel metric will not be added. This option is used to make this router prefer HELLO routes learned via the specified interface(s) less than HELLO routes from other interfaces.

metricout *metric*
Specifies the HELLO metric to be added to routes that are send via the specified interface(s). The default is zero. This option is used to make other routers prefer other sources of HELLO routes over this router.

trustedgateways *gateway_list*
Defines the list of gateways from which HELLO will accept updates. The *gateway_list* is simply a list of host names or IP addresses. By default, all routers on the shared network are trusted to supply routing information. But if the **trustedgateways** clause is specified only updates from the gateways in the list are accepted.

sourcegateways *gateway_list*
Defines a list of routers to which HELLO sends packets directly, not through multicast or broadcast. This can be used to send different routing information to specific gateways. Updates to gateways in this list are not affected by **noripout** on the interface.

traceoptions *trace_options*
Specifies the tracing options for HELLO. (See Trace Statements and the HELLO specific tracing options below.)

The default preference is 90. The default metric is 30000.

Tracing options

The **policy** option logs info whenever a new route is announce, the metric being announced changes or a route goes or leaves holddown.

Packet tracing options (which may be modified with **detail**, **send** and/or **recv**):

packets
All HELLO packets

The OSPF Protocol

Open Shortest Path Routing (OSPF) is a *shortest path first* (SPF) or *link-state* protocol. OSPF is an interior gateway protocol that distributes routing information between routers in a single autonomous system. OSPF chooses the least cost path as the best path. Suitable for complex networks with a large number of routers, OSPF provides equal cost multipath routing where packets to a single destination can be sent via more than one interface simultaneously. In a link-state protocol, each router maintains a database describing the entire AS topology, which it builds out of the collected link state advertisements of all routers. Each participating router distributes its local state (the usable interfaces and reachable neighbors of the router) throughout the AS by flooding. Each multiaccess network that has at least two attached routers has a *designated router* and a *backup designated router*. The designated router floods a link state advertisement for the multiaccess network and has other special responsibilities. The designated router concept reduces the number of adjacencies required on a multiaccess network.

OSPF allows networks to be grouped into areas. Routing information passed between areas is abstracted, potentially allowing a significant reduction in routing traffic. OSPF uses four different types of routes, listed in order of preference: intra-area, inter-area, type 1 external and type 2 external. Intra-area paths have destinations within the same area, inter-area paths have destinations in other OSPF areas and Autonomous System External (ASE) routes are routes to destinations external to the AS. Routes imported into OSPF as type 1 routes are supposed to be from igps whose external metrics are directly comparable to OSPF metrics. When a routing decision is being made, OSPF will add the internal cost to the AS Border router to the external metric. Type 2 ASEs are used for egps whose metrics are not comparable to OSPF metrics. In this case, only the internal OSPF cost to the AS Border router is used in the routing decision.

From the topology database, each router constructs a tree of the shortest paths with itself as the root. This shortest-path tree gives the route to each destination in the AS. Externally derived routing information appears on the tree as leaves. The link-state advertisement format distinguishes between information acquired from external sources and information acquired from internal routers, so there is no ambiguity about the source or reliability of routes. Externally derived routing information (for example, routes learned from EGP or BGP) is passed transparently through the autonomous system and is kept separate from the OSPF internally derived data. Each external route can also be tagged by the advertising router, enabling a passing of additional information between routers on the borders of the autonomous system.

g

OSPF optionally includes *type of service* (TOS) routing and allows administrators to install multiple routes to a given destination for each type of service (low delay or high throughput.) A router running OSPF uses the destination address and the type of service to choose the best route to the destination.

OSPF intra- and inter-area routes are always imported into the GateD routing database with a preference of 10. It would be a violation of the protocol if an OSPF router did not participate fully in the OSPF of the area, so it is not possible to override this. Although it is possible to give other routes lower preference values explicitly, it is ill-advised to do so.

Hardware multicast capabilities are also used where possible to deliver link-status messages. OSPF areas are connected by the *backbone* area, the area with identifier 0.0.0.0. All areas must be logically contiguous and the backbone is no exception. To permit maximum flexibility, OSPF allows the configuration of *virtual* links enable the backbone area to appear contiguous despite the physical reality.

All routers in an area must agree on the parameters of that area. A separate copy of the link-state algorithm is run for each area. Because of this, most configuration parameters are defined on a per area basis. All routers belonging to an area must agree on the configuration of that area. Misconfiguration will lead to adjacencies not forming between neighbors, and routing information might not flow, or even loop.

Authentication

All OSPF protocol exchanges are authenticated. Authentication guarantees that routing information is only imported from trusted routers, to protect the Internet and its users. A variety of authentication schemes can be used but a single scheme must be configured for each area. This enables some areas to use much stricter authentication than others. OSPF protocol exchanges may be authenticated. Authentication guarantees that routing information is imported only from trusted routers, to protect the Internet and its users. There are two authentication schemes available. The first uses a simple authentication key of up to 8 characters and is standardized. The second is still experimental and uses the MD5 algorithm and an authentication key of up to 16 characters.

The simple password provides very little protection because in many cases it is possible to easily capture packets from the network and learn the authentication key. The experimental MD5 algorithm provides much more protection as it does not include the authentication key in the packet.

The OSPF specification currently specifies that the authentication type be configured per area with the ability to configure separate passwords per interface. This has been extended to allow the configuration of different authentication types and keys per interface. In addition it is possible to specify both a *primary* and a *secondary* authentication type and key on each interface. Outgoing packets use the primary authentication type, but incoming packets may match either the primary or secondary authentication type and key.

The OSPF Statement

ospf yes | **no** | **on** | **off** [**{**
  defaults {
    preference *preference* **;**

```
        cost cost ;
        tag [ as ] tag ;
        type 1 | 2 ;
    } ;
    exportlimit routes ;
    exportinterval time ;
    traceoptions trace_options ;
    monitorauthkey authkey ;
    monitorauth none | ( [ simple | md5 ] authkey ) ;
    backbone | ( area area ) {
        authtype 0 | 1 | none | simple ;
        stub [ cost cost] ;
        networks {
            network [ restrict ] ;
            network mask mask [ restrict ] ;
            network masklen number [ restrict ] ;
            host host [ restrict ] ;
        } ;
        stubhosts {
            host cost cost ;
        } ;
        interface interface_list; [cost cost ] {
            interface_parameters
        } ;
        interface interface_list nonbroadcast [cost cost ] {
            pollinterval time ;
            routers {
                gateway [ eligible ] ;
            } ;
            interface_parameters
        } ;
        Backbone only:
        virtuallink neighborid router_id transitarea area {
            interface_parameters
        } ;
    } ;
} ] ;
```

The following are the *interface_parameters* referred to above. The may be specified on any class of interface and are described under the **interface** clause.

```
    enable | disable ;
    retransmitinterval time ;
    transitdelay time ;
    priority priority ;
    hellointerval time ;
    routerdeadinterval time ;
    authkey auth_key ;
```

defaults

These parameters specify the defaults used when importing OSPF ASE routes into the gated routing table and exporting routes from the gated routing table into OSPF ASEs.

preference *preference*

The preference is used to determine how OSPF routes compete with routes from other protocols in the gated routing table. The default value is 150.

cost *cost*

The cost is used when exporting a non-OSPF route from the GateD routing table into OSPF as an ASE. The default value is 1. This may be explicitly overridden in export policy.

tag [**as**] *tag*

OSPF ASE routes have a 32 bit tag field that is not used by the OSPF protocol, but may be used by export policy to filter routes. When OSPF is interacting with an egp, the tag field may be used to propagate AS path information, in which case the **as** keyword is specified aht the tag is limited to 12 bits of information. If not specified, the tag is set to zero.

type *1 | 2*
Routes exported from the GateD routing table into OSPF default to becoming type 1 ASEs. This default may be explicitly changed here and overridden in export policy.

ASE export rate
Because of the nature of OSPF, the rate at which ASEs are flooded must be limited. These two parameters can be used to adjust those rate limits.

exportinterval *time*
This specifies how often a batch of ASE link state advertisements will be generated and flooded into OSPF. The default is once per second.

exportlimit *routes*
This parameter specifies how many ASEs will be generated and flooded in each batch. The default is 100.

traceoptions *trace_options*
Specifies the tracing options for OSPF. (See Trace Statements and the OSPF specific tracing options below.)

g

monitorauthkey *authkey*
OSPF state may be queried using the **ospf_monitor** (This should be a hyperlink) utility. This utility sends non-standard OSPF packets which generate a text response from OSPF. By default these requests are not authenticated, if an authentication key is configured, the incoming requests must match the specified authentication key. No OSPF state may be changed by these packets, but the act of querying OSPF can utilize system resources.

backbone
area *area*
Each OSPF router must be configured into at least one OSPF area. If more than one area is configured, at least one must the be **backbone**. The backbone may only be configured using the **backbone** keyword, it may not be specified as **area 0**. The backbone interface may be a **virtual-link**.

authtype *0 | 1 | none | simple*
OSPF specifies an authentication scheme per area. Each interface in the area must use this same authentication scheme although it may use a different **authenticationkey**. The currently valid values are **none** (**0**) for no authentication, or **simple** (**1**) for simple password authentication.

stub [**cost** *cost*]
A **stub** area is one in which there are no ASE routes. If a **cost** is specified, this is used to inject a default route into the area with the specified cost.

networks
The **networks** list describes the scope of an area. Intra-area LSAs that fall within the specified ranges are not advertised into other areas as inter-area routes. Instead, the specified ranges are advertised as *summary network* LSAs. If **restrict** is specified, the summary network LSAs are not advertised. Intra-area LSAs that do not fall into any range are also advertised as summary network LSAs. This option is very useful on well designed networks in reducing the amount of routing information propagated between areas. The entries in this list are either networks, or a subnetwork/mask pair. See the section on Route Filtering for more detail about specifying ranges.

stubhosts
This lists specifies directly attached hosts that should be advertised as reachable from this router and the costs they should be advertised with. Point-to-point interfaces on which it is not desirable to run OSPF should be specified here.

It is also useful to assign a additional address to the loopback interface (one not on the 127 network) and advertise it as a stub hosts. If this address is the same one used as the router-id, it enables routing to OSPF routers by router-id, instead of by interface address. This is more reliable than routing to one of the routers interface addresses which may not always be reachable.

interface *interface_list* [**cost** *cost*]
This form of the interface clause is used to configure a **broadcast** (which requires IP multicast support) or a **point-to-point** interface. See the section on interface list specification for the

description of the *interface_list*.

Each interface has a **cost**. The costs of all interfaces a packet must cross to reach a destination are summed to get the cost to that destination. The default cost is one, but another non-zero value may be specified.

Interface parameters common to all types of interfaces are:

retransmitinterval *time*
: The number of seconds between link state advertisement retransmissions for adjacencies belonging to this interface.

transitdelay *time*
: The estimated number of seconds required to transmit a link state update over this interface. Transitdelay takes into account transmission and propagation delays and must be greater than 0.

priority *priority*
: A number between 0 and 255 specifying the priority for becoming the designated router on this interface. When two routers attached to a network both attempt to become designated router, the one with the highest priority wins. A router whose router priority is set to 0 is ineligible to become designated router.

hellointerval *time*
: The length of time, in seconds, between Hello packets that the router sends on the interface.

routerdeadinterval *time*
: The number of seconds not hearing Hello packets of a router before the neighbors of the router will declare it dovn.

authkey *auth_key*
: Used by OSPF authentication to generate and verify the authentication field in the OSPF header. The authentication key can be configured on a per interface basis. It is specified by one to eight decimal digits separated by periods, a one to eight byte hexadecimal string preceded by **0x**, or a one to eight character string in double quotes.

Point-to-point interfaces also support this additional parameter:

nomulticast
: By default, OSPF packets to neighbors on point-to-point interfaces are sent via the IP multicast mechanism. Although, some implementations of IP multicasting for Unix have a bug that precludes the use of IP multicasting on these interfaces. Gated will detect this condition and fall back to using sending unicast OSPF packets to this point-to-point neighbor.

  If the use of IP multicasting is not desired because the remote neighbor does not support it, the **nomulticast** parameter may be specified to force the use of unicast OSPF packets. This option may also be used to eliminate warnings when Gated detects the bug mentioned above.

interface *interface_list* **nonbroadcast** [**cost** *cost*]
: This form of the interface clause is used to specify a **nonbroadcast** interface on a **non-broadcast multi-access** (NBMA) media. Since an OSPF **broadcast** media must support IP multicasting, a broadcast capable media, such as Ethernet, that does not support IP multicasting must be configured as a non-broadcast interface.

  A non-broadcast interface supports any of the standard **interface** clauses listed above, plus the following two that are specific to non-broadcast interfaces:

  pollinterval *time*
  : Before adjacency is established with a neighbor, OSPF packets are sent periodically at the specified **pollinterval**.

  routers
  : By definition it is not possible to send broadcast packets to discover OSPF neighbors on a non-broadcast, so all neighbors must be configured. The list includes one or more neighbors and an indication of their eligibility to become a designated router.

g

virtuallink neighborid *router_id* **transitarea** *area*

Virtual links are used to establish or increase connectivity of the backbone area. The **neighborid** is the *router_id* of the other end of the virtual link. The transit **area** specified must also configured on this system. All standard interface parameters defined by the **interface** clause above may be specified on a virtual link.

Tracing options

In addition to the following OSPF specific trace flags, OSPF supports the **state** which traces interface and neighbor state machine transitions.

lsabuild
Link State Advertisement creation

spf Shortest Path First (SPF) calculations

Packet tracing options (which may be modified with **detail**, **send** and **recv**):

hello
OSPF **HELLO** packets which are used to determine neighbor reachability.

dd OSPF Database Description packets which are used in synchronizing OSPF databases.

request
OSPF Link State Request packets which are used in synchronizing OSPF databases.

lsu OSPF Link State Update packets which are used in synchronizing OSPF databases.

ack OSPF Link State Ack packets which are used in synchronizing OSPF databases.

The Exterior Gateway Protocol (EGP)

The Exterior Gateway Protocol (EGP) is an exterior routing protocol used for exchanging routing information with gateways in other autonomous systems. Unlike interior protocols, EGP propagates only reachability indications, not true metrics. EGP updates contain metrics, called *distances* which range from 0 to 255. GateD will only compare EGP distances learned from the same AS. them.

Before EGP sends routing information to a remote router, it must establish an adjacency with that router. This is accomplished by an exchange of *Hello* (not to be confused with the HELLO protocol, or OSPF HELLO messages) and *I Heard You* (I-H-U) messages with that router. Computers communicating via EGP are called EGP *neighbors*, and the exchange of HELLO and I-H-U messages is referred to as *acquiring a neighbor*. Once the neighbor is acquired, the system *polls* the neighbor for routing information. The neighbor responds by sending an *update* containing routing information. If the system receives a poll from its neighbor, it responds with its own update packet. When the system receives an update, it includes routes from the update into its routing database. If the neighbor fails to respond to three consecutive polls, GateD assumes that the neighbor is down and removes the routes of that neighbor from its database.

The EGP Statement

egp yes | **no** | **on** | **off**
[**{**
  preference *preference* **;**
  defaultmetric *metric* **;**
  packetsize *number* **;**
  traceoptions *trace_options* **;**
  group
    [**peeras** *autonomous_system*]
    [**localas** *autonomous_system*]
    [**maxup** *number*]
  {
    neighbor *host*
      [**metricout** *metric*]
      [**preference** *preference*]
      [**preference2** *preference*]
      [**ttl** *ttl*]
      [**nogendefault**]
      [**importdefault**]
      [**exportdefault**]
      [**gateway** *gateway*]
      [**lcladdr** *local_address*]

```
        [ sourcenet network ]
        [ minhello | p1 time ]
        [ minpoll | p2 time ]
        [ traceoptions trace_options ]
           ;
  } ;
} ] ;
```

preference *preference*

Sets the preference for routes learned from RIP. The default preference is 200. This preference may be overridden by a preference specified on the **group** or **neighbor** statements or by import policy.

defaultmetric *metric* **;**

Defines the metric used when advertising routes via EGP. If not specified, the default value is 255 which some systems may consider unreachable. This choice of values requires you to explicitly specify a metric when exporting routes to EGP neighbors. This metric may be overridden by a metric specified on the neighbor or group statements or in export policy.

g

packetsize *maxpacketsize*

This defines the expected maximum size of a packet that EGP expects to receive from this neighbor. If a packet larger than this value is received, it will be incomplete and have to be discarded. The length of this packet will be noted and the expected size will be increased to be able to receive a packet of this size. Specifying the parameter here will prevent the first packet from being dropped. If not specified, the default size is 8192 bytes. All packet sizes are rounded up to a multiple of the system page size.

traceoptions *trace_options*

Specifies the tracing options for EGP. By default these are inherited from the global trace options. These values may be overridden on a group or neighbor basis. (See Trace Statements and the EGP specific tracing options below.)

group

EGP neighbors must be specified as members of a **group**. A group is usually used to group all neighbors in one autonomous system. Parameters specified on the group clause apply to all of the subsidiary neighbors unless explicitly overridden on a neighbor clause. Any number of **group** clauses may specify any number of **neighbor** clauses.

Any parameters from the **neighbor** subclause may be specified on the **group** clause to provide defaults for the whole group (which may be overridden for individual neighbors). In addition, the **group** clause is the only place to set the following attributes:

peeras

Identifies the autonomous system number expected from peers in the group. If not specified, it will be learned dynamically.

localas

Identifies the autonomous system which GateD is representing to the group. The default is that which has been set globally in the **autonomoussystem** statement. This option is usually only used when *masquerading* as another autonomous system and its use is discouraged.

maxup

Specifies the number of neighbors GateD should acquire from this group. The default is to acquire all of the neighbors in the group. GateD will attempt to acquire the first **maxup** neighbors in the order listed. If one of the first neighbors is not available, it will acquire one further down the list. If after start-up GateD does manage to acquire the more desirable neighbor, it will drop the less desirable one.

neighbor *neighbor_address*

Each neighbor subclause defines one EGP neighbor within a group. The only part of the subclause that is required is the **neighbor_address** argument which is the symbolic host name or IP address of the neighbor. All other parameters are optional.

preference *preference*

Specifies the preference used for routes learned from these neighbors. This can differ from the default EGP preference set in the **egp** statement, so that GateD can prefer routes from one neighbor, or group of neighbors, over another. This preference may be explicitly overridden by import policy.

preference2 *preference*
In the case of a **preference** tie, the second preference, **preference2** may be used to break the tie. The default value is 0.

metricout *metric*
This defines a metric to be used for all routes sent to this neighbor. The value overrides the default metric set in the **egp** statement and any metrics specified by export policy, but only for this specific neighbor or group of neighbors.

nogendefault
Prevents GateD from generating a default route when EGP receives a valid update from its neighbor. The default route is only generated when the gendefault option is enabled.

importdefault
Enables GateD to accept the default route (0.0.0.0) if it is included in a received EGP update. If not specified, the default route contained in an EGP update is ignored. For efficiency, some networks have external routers announce a default route to avoid sending large EGP update packets.

exportdefault
Enables GateD to include the default route (0.0.0.0) in EGP updates sent to this EGP neighbor. This allows the system to advertise the default route via EGP. Normally a default route is not included in EGP updates.

gateway *gateway*
If a network is not shared with a neighbor, **gateway** specifies a router on an attached network to be used as the next hop router for routes received from this neighbor. This option is only rarely used.

lcladdr *local_address*
Specifies the address to be used on the local end of the connection with the neighbor. The local address must be on an interface which is shared with the neighbor or with the *gateway* of the neighbor when the **gateway** parameter is used. A sessionwill only be opened when an interface with the appropriate local address (through which the neighbor or gateway address is directly reachable) is operating.

sourcenet *network*
Specifies the network queried in the EGP Poll packets. By default this is the network shared with neighbors address specified. If there is no network shared with the neighbor, one of the network the neighbor is attached to should be specified. This parameter can also be used to specify a network shared with the neighbor other than the one on which the EGP packets are sent. This parameter is normally not needed.

p1 *time*
minhello *time*
Sets the minimum acceptable interval between the transmission of EGP *HELLO* packets. The default hello interval is 30 seconds. If the neighbor fails to respond to three hello packets, GateD stops trying to acquire the neighbor. Setting a larger interval gives the neighbor a better chance to respond. Minhello is an alias for the **P1** value defined in the EGP specification.

p2 *time*
minpoll *time*
Sets the time interval between polls to the neighbor. The default is 120 seconds. If three polls are sent without a response, the neighbor is declared "down" and all routes learned from that neighbor are removed from the routing database. A longer polling interval supports a more stable routing database but is not as responsive to routing changes. Minpoll is an alias for the **P2** value defined in the EGP specification.

ttl *ttl*
By default, GateD sets the IP TTL for local neighbors to *one* and the TTL for non-local neighbors to 255. This option is provided when attempting to communicate with improperly functioning routers that ignore packets sent with a TTL of one.

traceoptions *trace_options*
Specifies the tracing options for this EGP neighbor. By default these are inherited from group or EGP global trace options. (See Trace Statements and the EGP specific tracing options below.)

Tracing options

The **state** and **policy** options work with EGP.

Packet tracing options (which may be modified with **detail**, **send** and **recv**):

packets
All EGP packets

hello
EGP **HELLO/I-HEARD-U** packets which are used to determine neighbor reachability.

acquire
EGP **ACQUIRE/CEASE** packets which are used to initiate and terminate EGP sessions.

update
EGP **POLL/UPDATE** packets which are used to request and receive reachability updates.

g

The BGP Protocol

The Border Gateway Protocol (BGP) is an exterior routing protocol used for exchanging routing information between autonomous systems. BGP is used for exchange of routing information between multiple transit autonomous systems as well as between transit and stub autonomous systems. BGP is related to EGP but operates with more capability, greater flexibility, and less required bandwidth. BGP uses *path attributes* to provide more information about each route, and in particular maintain an *AS path*, which includes the AS number of each autonomous system the route has transited, providing information sufficient to prevent routing loops in an arbitrary topology. Path attributes may also be used to distinguish between groups of routes to determine administrative preferences, allowing greater flexibility in determining route preference to achieve a variety of administrative ends.

BGP supports two basic types of sessions between neighbors, internal (sometimes referred to as IBGP) and external. Internal sessions are run between routers in the same autonomous system, while external sessions run between routers in different autonomous systems. When sending routes to an external peer the local AS number is prepended to the AS path, hence routes received from an external peer are guaranteed to have the AS number of that peer at the start of the path. Routes received from an internal neighbor will not in general have the local AS number prepended to the AS path, and hence will in general have the same AS path that the route had when the originating internal neighbor received the route from an external peer. Routes with no AS numbers in the path may be legitimately received from internal neighbors; these indicate that the received route should be considered internal to your own AS.

The BGP implementation supports three versions of the BGP protocol, versions 2, 3 and 4. BGP versions 2 and 3 are quite similar in capability and function. They will only propagate classed network routes, and the AS path is a simple array of AS numbers. BGP 4 will propagate fully general address-and-mask routes, and the AS path has some structure to represent the results of aggregating dissimilar routes.

External BGP sessions may or may not include a single metric, which BGP calls the *Multi-Exit Discriminator*, in the path attributes. For BGP versions 2 and 3 this metric is a 16-bit unsigned integer, for BGP version 4 it is a 32-bit unsigned integer. In either case smaller values of the metric are to be preferred. Currently this metric is only used to break ties between routes with equal preference from the same neighbor AS. Internal BGP sessions carry at least one metric in the path attributes, which BGP calls the *LocalPref*. The size of the metric is identical to the MED. For BGP versions 2 and 3 this metric is considered better when its value is smaller, for version 4 it is better when it is larger. BGP version 4 sessions may optionally carry a second metric on internal sessions, this being an internal version of the *Multi-Exit Discriminator*. The use of these metrics is dependent on the type of internal protocol processing which is specified.

BGP collapses routes with similar path attributes into a single update for advertisement. Routes that are received in a single update will be readvertised in a single update. The churn caused by the loss of a neighbor will be minimized and the initial advertisement sent during peer establishment will be maximally compressed. BGP does not read information from the kernel message-by-message, but fills the input buffer. It processes all complete messages in the buffer before reading again. BGP also does multiple reads to clear all incoming data queued on the socket. This feature may cause other protocols to be blocked for prolonged intervals by a busy peer connection.

All unreachable messages are collected into a single message and sent prior to reachable routes during a flash update. For these unreachable announcements, the next hop is set to the local address on the connection, no metric is sent and the path origin is set to incomplete. On external connections the AS path in unreachable announcements is set to the local AS, on internal connections the AS path is set to zero length.

The BGP implementation expects external peers to be directly attached to a shared subnet, and expects those peers to advertise next hops which are host addresses on that subnet (though this constraint can be relaxed by configuration for testing). For groups of internal peers, however, there are several alternatives which may be selected from by specifying the group type. Type `internal` groups expect all peers to be directly attached to a shared subnet so that, like external peers, the next hops received in BGP advertisements may be used directly for forwarding. Type `routing` groups instead will determine the immediate next hops for routes by using the next hop received with a route from a peer as a forwarding address. This forwarding address is used to look up an immediate next hop in routes of the IGP. Such groups support distant peers, but need to be informed of the IGP whose routes they are using to determine immediate next hops. Finally, type `igp` groups expect routes from the group peers to not be used for forwarding at all. Instead they expect that copies of the BGP routes received will also be received via an IGP, and that the BGP routes will only be used to determine the path attributes associated with the IGP routes. Such groups also support distant peers, and also need to be informed of the IGP they are running with.

g

For internal BGP group types (and for test groups), where possible a single outgoing message is built for all group peers based on the common policy. A copy of the message is sent to every peer in the group, with possible adjustments to the next hop field as appropriate to each peer. This minimizes the computational load of running large numbers of peers in these types of groups. BGP allows unconfigured peers to connect if an appropriate group has been configured with an `allow` clause.

The BGP Statement

```
bgp yes | no | on | off
[ {
   preference preference ;
   defaultmetric metric ;
   traceoptions trace_options ;
   group type ( external peeras autonomous_system )
      | ( internal peeras autonomous_system )
      | ( igp peeras autonomous_system proto proto )
      | ( routing peeras autonomous_system proto proto
           interface interface_list )
      | ( test peeras autonomous_system )
   {
      allow {
         network
         network mask mask
         network masklen number
         all
         host host
      } ;
      peer host
         [ metricout metric ]
         [ localas autonomous_system ]
         [ nogendefault ]
         [ gateway gateway ]
         [ preference preference ]
         [ preference2 preference ]
         [ lcladdr local_address ]
         [ holdtime time ]
         [ version number ]
         [ passive ]
         [ sendbuffer number ]
         [ recvbuffer number ]
         [ indelay time ]
         [ outdelay time ]
         [ keep [ all | none ] ]
         [ showwarnings ]
         [ noauthcheck ]
```

```
            [ noaggregatorid ]
            [ keepalivesalways ]
            [ v3asloopokay ]
            [ nov4asloop ]
            [ logupdown ]
            [ ttl ttl ]
            [ traceoptions trace_options ]
            ;
        } ;
    } ] ;
```

external | **internal** | **igp** | **test**

The **bgp** statement enables or disables BGP. By default BGP is disabled. The default metric for announcing routes via BGP is not to send a metric.

preference *preference*
: Sets the preference for routes learned from RIP. The default preference is 170. This preference may be overridden by a preference specified on the **group** or **peer** statements or by import policy.

defaultmetric *metric*
: Defines the metric used when advertising routes via BGP. If not specified, no metric is propagated. This metric may be overridden by a metric specified on the neighbor or group statements or in export policy.

traceoptions *trace_options*
: Specifies the tracing options for BGP. By default these are inherited from the global trace options. These values may be overridden on a group or neighbor basis. (See Trace Statements and the BGP specific tracing options below.)

g

Groups

BGP peers are grouped by type and the autonomous system of the peers. Any number of groups may be specified, but each must have a unique combination of type and peer autonomous system. There are four possible group types:

group type external peeras *autonomous_system*
: In the classic external BGP group, full policy checking is applied to all incoming and outgoing advertisements. The external neighbors must be directly reachable through one of the local interfaces of the machine . By default no metric is included in external advertisements, and the next hop is computed with respect to the shared interface.

group type internal peeras *autonomous_system*
: An internal group operating where there is no IP-level IGP, for example an SMDS network or MILNET. All neighbors in this group are required to be directly reachable via a single interface. All next hop information is computed with respect to this interface. Import and export policy may be applied to group advertisements. Routes received from external BGP or EGP neighbors are by default readvertised with the received metric.

group type igp peeras *autonomous_system* **proto** *proto*
: An internal group that runs in association with an interior protocol. The IGP group examines routes which the IGP is exporting and sends an advertisement only if the path attributes could not be entirely represented in the IGP tag mechanism. Only the AS path, path origin, and transitive optional attributes are sent with routes. No metric is sent, and the next hop is set to the local address used by the connection. Received internal BGP routes are not used or readvertised. Instead, the AS path information is attached to the corresponding IGP route and the latter is used for readvertisement. Since internal IGP peers are sent only a subset of the routes which the IGP is exporting, the export policy of the IGP is used. There is no need to implement the "don't routes from peers in the same group" constraint since the advertised routes are routes that IGP already exports.

group type routing peeras *autonomous_system* **proto** *proto* **interface** *interface_list*
: An internal group which uses the routes of an interior protocol to resolve forwarding addresses. A type routing group propagates external routes between routers which are not directly connected. A type routing group computes immediate next hops for these routes by using the BGP next hop which arrived with the route as a forwarding address. The forwarding address is to be

resolved via the routing information of an internal protocol. In essence, internal BGP is used to carry AS external routes, while the IGP is expected to only carry AS internal routes, and the latter is used to find immediate next hops for the former.

The *proto* names the interior protocol to be used to resolve BGP route next hops, and may be the name of any IGP in the configuration. By default the next hop in BGP routes advertised to type routing peers will be set to the local address on the BGP connection to those peers, as it is assumed a route to this address will be propagated via the IGP. The *interface_list* can optionally provide a list interfaces whose routes are carried via the IGP for which third party next hops may be used instead.

group type test peeras *autonomous_system*
An extension to external BGP which implements a fixed policy using test peers. Fixed policy and *special case* code make test peers relatively inexpensive to maintain. Test peers do not need to be on a directly attached network. If GateD and the peer are on the same (directly attached) subnet, the advertised next hop is computed with respect to that network. Otherwise the next hop is the current next hop of the local machine. All routing information advertised by and received from a test peer is discarded, and all BGP routes that can be advertised are sent back to the test peer. Metrics from EGP-derived and BGP-derived routes are forwarded in the advertisement. Otherwise no metric is included.

g

Group parameters

The BGP statement has `group` clauses and `peer` subclauses. Any number of peer subclauses may be specified within a group. A group clause usually defines default parameters for a group of peers, these parameters apply to all subsidiary peer subclauses. Any parameters from the peer subclause may be specified on the group clause to provide defaults for the whole group (which may be overridden for individual peers).

Specifying peers

Within a group, BGP peers may be configured in one of two ways. They may be explicitly configured with a `peer` statement, or implicitly configured with the `allow` statement. Both are described here:

allow
The allow clauses allows for `peer` connections from any addresses in the specified range of network and mask pairs. All parameters for these peers must be configured on the group clause. The internal peer structures are created when an incoming open request is received and destroyed when the connection is broken. For more detail on specifying the network/mask pairs, see the section on Route Filtering.

peer *host*
A `peer` clause configures an individual peer. Each peer inherits all parameters specified on a group as defaults. Those default may be overridden by parameters explicitly specified on the peer subclaus.

Within each `group` clause, individual peers can be specified or a group of *potential* peers can be specified using `allow`. `Allow` is used to specify a set of address masks. If GateD receives a BGP connection request from any address in the set specified, it will accept it and set up a peer relationship.

Peer parameters

The BGP `peer` subclause allows the following parameters, which can also be specified on the `group` clause. All are optional.

metricout *metric*
If specified, this metric is used as the primary metric on all routes sent to the specified peer(s). This metric overrides the default metric, a metric specified on the group and any metric specified by export policy.

localas *autonomous_system*
Identifies the autonomous system which GateD is representing to this group of peers.. The default is that which has been set globally in the `autonomoussystem` statement.

nogendefault
Prevents GateD from generating a default route when EGP receives a valid update from its neighbor. The default route is only generated when the gendefault option is enabled.

gateway *gateway*
If a network is not shared with a peer, `gateway` specifies a router on an attached network to be

used as the next hop router for routes received from this neighbor. This parameter is not needed in most cases.

preference *preference*
Specifies the preference used for routes learned from these peers. This can differ from the default BGP preference set in the **bgp** statement, so that GateD can prefer routes from one peer, or group of peer, over others. This preference may be explicitly overridden by import policy.

preference2 *preference*
In the case of a **preference** tie, the second preference, **preference2** may be used to break the tie. The default value is 0.

lcladdr *local_address*
Specifies the address to be used on the local end of the TCP connection with the peer. For *external* peers the local address must be on an interface which is shared with the peer or with the *gateway* of the peer when the **gateway** parameter is used. A session with an external peer will only be opened when an interface with the appropriate local address (through which the peer or gateway address is directly reachable) is operating. For other types of peers, a peer session will be maintained when any interface with the specified local address is operating. In either case incoming connections will only be recognized as matching a configured peer if they are addressed to the configured local address.

holdtime *time*
Specifies the BGP holdtime value to use when negotiating the connection with this peer, in seconds. According to BGP, if GateD does not receive a keepalive, update, or notification message within the period specified in the Hold Time field of the BGP Open message, then the BGP connection will be closed. The value must be either 0 (no keepalives will be sent) or at least 3.

version *version*
Specifies the version of the BGP protocol to use with this peer. If not specified, the highest supported version is used first and version negotiation is attempted. If it is specified, only the specified version will be offered during negotiation. Currently supported version are 2, 3 and 4.

passive
Specifies that active OPENs to this peer should not be attempted. GateD should wait for the peer to issue an open. By default all explicitly configured peers are active, they periodically send OPEN messages until the peer responds.

sendbuffer *buffer_size*
recvbuffer *buffer_size*
Control the amount of send and receive buffering asked of the kernel. The maximum supported is 65535 bytes although many kernels have a lower limit. By default, GateD configures the maximum supported. These parameters are not needed on normally functioning systems.

indelay *time*
outdelay *time*
Used to dampen route fluctuations. **Indelay** is the amount of time a route learned from a BGP peer must be stable before it is accepted into the gated routing database. **Outdelay** is the amount of time a route must be present in the gated routing database before it is exported to BGP. The default value for each is 0, meaning that these features are disabled.

keep all
Used to retain routes learned from a peer even if the AS paths of the routes contain one of our exported AS numbers.

showwarnings
Causes GateD to issue warning messages when receiving questionable BGP updates such as duplicate routes and/or deletions of non-existing routes. Normally these events are silently ignored.

noauthcheck
Normally GateD verifies that incoming packets have an authentication field of all ones. This option may be used to allow communication with an implementation that uses some other form of authentication.

noaggregatorid
Causes GateD to specify the routerid in the aggregator attribute as zero (instead of its routerid)

g

in order to prevent different routers in an AS from creating aggregate routes with different AS paths.

keepalivesalways
Causes gated to always send keepalives, even when an update could have correctly substituted for one. This allows interoperability with routers that do not completely obey the protocol specifications on this point.

v3asloopokay
By default gated will not advertise routes whose AS path is looped (with an AS appearing more than once in the path) to version 3 external peers. Setting this flag removes this constraint. Ignored when set on internal groups or peers.

nov4asloop
Prevents routes with looped AS paths from being advertised to version 4 external peers. This can be useful to avoid advertising such routes to peer which would incorrectly forward the routes on to version 3 neighbors.

logupdown
Causes a message to be logged via the syslog mechanism whenever a BGP peer enters or leaves the **ESTABLISHED** state.

ttl *ttl*
By default, GateD sets the IP TTL for local peers to *one* and the TTL for non-local peers to 255. This option mainly is provided when attempting to communicate with improperly functioning routers that ignore packets sent with a TTL of one. Not all kernels allow the TTL to be specified for TCP connections.

traceoptions *trace_options*
Specifies the tracing options for this BGP neighbor. By default these are inherited from group or BGP global trace options. (See Trace Statements and the BGP specific tracing options below.)

Tracing options

Note that the **state** option works with BGP, but does not provide true state transition information.

Packet tracing options (which may be modified with **detail**, **send** and **recv**):

packets
All BGP packets

open
BGP **OPEN** packets which are used to establish a peer relationship.

update
BGP **UPDATE** packets which are used to pass network reachability information.

keepalive
BGP **KEEPALIVE** packets which are used to verify peer reachability.

The ICMP Statement

On systems without the BSD routing socket, gated listens to ICMP messages received by the system. Currently gated only does processing on ICMP redirect packets, but more functionality may be added in the future, such as support for the router discovery messages. Processing of ICMP redirect messages is handled by the **redirect** statement.

Currently the only reason to specify the **icmp** statement is to be able to trace the ICMP messages that gated receives.

The ICMP statement

icmp {
  traceoptions *trace_options* **;**
}

traceoptions *trace_options* **;**
Specifies the tracing options for ICMP. (See Trace Statements and the ICMP specific tracing options below.)

g

Tracing options

Packet tracing options (which may be modified with **detail** and **recv**):

packets
All ICMP packets received.

redirect
Only ICMP **REDIRECT** packets received.

routerdiscovery
Only ICMP **ROUTER DISCOVERY** packets received.

info Only ICMP informational packets, which include mask request/response, info request/response, echo request/response and time stamp request/response.

error
Only ICMP error packets, which include time exceeded, parameter problem, unreachable and source quench.

g

Redirect Processing

The **redirect** code is passed ICMP or ISO redirects learned by monitoring ICMP messages, or via the routing socket on systems that support it. It processes the redirect request and decides whether to accept the redirect. If the redirect is accepted, a route is installed in the gated routing table with the protocol **redirect**. Redirects are deleted from the routing table after 3 minutes.

If GateD determines that a redirect is not acceptable, it tries to figure out if the kernel forwarding table has been modified. On systems where ICMP messages are monitored this is accomplished by trying to second guess what the kernel would have done with the redirect. On systems with the routing socket, the kernel provides and indication of whether the redirect was accepted; GateD ignores redirects that were not processed.

If GateD has determined that the state of the kernel forwarding table has been changed, the necessary requests to the kernel are made to restore the correct state.

Note that on currently available systems it is not possible to disable the processing of ICMP redirects, even when the system is functioning as a router. To ignore the effects of redirects, GateD must process each one and actively restore any changes it made to the state of the kernel. Because of the mechanisms involved, there will be windows where the effects of redirects are present in the kernel.

By default, GateD removes redirects when actively participating in an interior gateway protocol (RIP, HELLO, OSPF or IS-IS). It is not possible to enable redirects once they have been automatically disabled. Listening to RIP or HELLO in *nobroadcast* mode does not cause redirects to be ignored, nor does the use of EGP and BGP. Redirects must be manually configured off in these cases.

Note that in accordance with the latest IETF Router Requirements document, GateD insures that all ICMP net redirects are processed as host redirects. When an ICMP net redirect is accepted, GateD issues the requests to the kernel to make sure that the kernel forwarding table is updated to reflect a host redirect instead of a net redirect.

The redirect statement does not prevent the system from sending redirects, only from listening to them.

The Redirect Statement

redirect yes | **no** | **on** | **off**
[**{**
  preference *preference* **;**
  interface *interface_list*
    [**noredirects**] | [**redirects**] **;**
  trustedgateways *gateway_list* **;**
  traceoptions *trace_options* **;**
}] **;**

preference
Sets the preference for a route learned from a redirect. The default is 30.

interface *interface_list*
The **interface** statement allows the enabling and disabling of redirects on an interface-by-interface basis. See the section on interface list specification for the description of the *interface_list*. The possible parameters are:

noredirects
Specifies that redirects received via the specified interface will be ignored. The default is to accept redirects on all interfaces.

redirects
This is the default. This argument may be necessary when **noredirects** is used on a wildcard interface descriptor.

trustedgateways *gateway_list*
Defines the list of gateways from which redirects will be accepted. The *gateway_list* is simply a list of host names or addresses. By default, all routers on the shared network(s) are trusted to supply redirects. But if the **trustedgateways** clause is specified only redirects from the gateways in the list are accepted.

traceoptions *trace_options*
There are no redirect-specific tracing options. All non-error messages are traced under the **normal** class.

Tracing options

There are no **Redirect**-specific tracing options. All non-error messages are traced under the **normal** class.

The Router Discovery Protocol

The Router Discovery Protocol is an IETF standard protocol used to inform hosts of the existence of routers. It is intended to be used instead of having hosts *wiretap* routing protocols such as RIP. It is used in place of, or in addition to statically configured default routes in hosts.

The protocol is split into to portions, the *server* portion which runs on routers, and the *client* portion that runs on hosts. GateD treats these much like two separate protocols, only one of which may be enabled at a time.

The Router Discovery Server

The Router Discovery Server runs on routers and announces their existence to hosts. It does this by periodically multicasting or broadcasting a **Router Advertisement** to each interface on which it is enabled. These Router Advertisements contain a list of all the routers addresses on a given interface and their preference for use as a default router.

Initially these Router Advertisements occur every few seconds, then fall back to every few minutes. In addition, a host may send a **Router Solicitation** to which the router will respond with a unicast Router Advertisement (unless a multicast or broadcast advertisement is due momentarily).

Each Router Advertisement contains a *Advertisement Lifetime* field indicating for how long the advertised addresses are valid. This lifetime is configured such that another Router Advertisement will be sent before the lifetime has expired. A lifetime of zero is used to indicate that one or more addresses are no longer valid.

On systems supporting IP multicasting, the Router Advertisements are by default send to the all-hosts multicast address **224.0.0.1**. However, the use of broadcast may be specified. When Router Advertisements are being sent to the all-hosts multicast address, or an interface is configured for the limited-broadcast address **255.255.255.255**, all IP addresses configured on the physical interface are included in the Router Advertisement. When the Router advertisements are being sent to a net or subnet broadcast, only the address associated with that net or subnet is included.

The Router Discovery Server Statement

```
routerdiscovery server yes | no | on | off [ {
   traceoptions trace_options ;
   interface interface_list
      [ minadvinterval time ]
      [ maxadvinterval time ]
      [ lifetime time ]
      ;
   address interface_list
      [ advertise ] | [ ignore ]
      [ broadcast ] | [ multicast ]
      [ ineligible ] | [ preference preference ]
      ;
```

}] ;

traceoptions *trace_options*
Specifies the Router Discovery tracing options. (See Trace Statements and the Router Discovery specific tracing options below.)

interface *interface_list*
Specifies the parameters that apply to physical interfaces. Note a slight difference in convention from the rest of GateD, **interface** specifies just physical interfaces (such as **le0**, **ef0** and **en1**), while **address** specifies protocol (in this case IP) addresses.

Interface parameters are:

maxadvinterval *time*
The maximum time allowed between sending broadcast or multicast Router Advertisements from the interface. Must be no less than *4* and no more than *30:00* (30 minutes or 1800 seconds). The default is **10:00** (10 minutes or 600 seconds).

minadvinterval *time*
The minimum time allowed between sending unsolicited broadcast or multicast Router Advertisements from the interface. Must be no less than *3* seconds and no greater than **maxadvinterval**. The default is **0.75 * maxadvinterval**.

lifetime *time*
The lifetime of addresses in a Router Advertisement. Must be no less than **maxadvinterval** and no greater than *2:30:00* (two hours, thirty minutes or 9000 seconds). The default is **3 * maxadvinterval**.

address *interface_list*
Specifies the parameters that apply to the specified set of addresses on this physical interfaces. Note a slight difference in convention from the rest of GateD, **interface** specifies just physical interfaces (such as **le0**, **ef0** and **en1**), while **address** specifies protocol (in this case IP) addresses.

advertise
Specifies that the specified address(es) should be included in Router Advertisements. This is the default.

ignore
Specifies that the specified address(es) should not be included in Router Advertisements.

broadcast
Specifies that the given address(es) should be included in a broadcast Router Advertisement because this system does not support IP multicasting, or some hosts on attached network do not support IP multicasting. It is possible to mix addresses on a physical interface such that some are included in a broadcast Router Advertisement and some are included in a multicast Router Advertisement. This is the default if the router does not support IP multicasting.

multicast
Specifies that the given address(es) should only be included in a multicast Router Advertisement. If the system does not support IP multicasting, the address(es) will not be included. If the system supports IP multicasting, the default is to include the address(es) in a multicast Router Advertisement if the given interface supports IP multicasting. If the given interface does not support IP multicasting, the address(es) will be included in a broadcast Router Advertisement.

preference *preference*
The preferability of the address(es) as a default router address, relative to other router addresses on the same subnet. A 32-bit, signed, twos-complement integer, with higher values meaning more preferable. Note that **hex 80000000** may only be specified as **ineligible**. The default is **0**.

ineligible
Specifies that the given address(es) will be assigned a preference of (hex 80000000) which means that it is not eligible to be the default route for any hosts.

g

This is useful when the address(es) should not be used as a default route, but are given as the next hop in an ICMP redirect. This allows the hosts to verify that the given addresses are up and available.

The Router Discovery Client

A host listens for Router Advertisements via the all-hosts multicast address (**224.0.0.2**) if IP multicasting is available and enabled, or on the interface broadcast address. When starting up, or when reconfigured, a host may send a few Router Solicitations to the all-routers multicast address, **224.0.0.2**, or the interface broadcast address.

When a Router Advertisement with non-zero lifetime is received, the host installs a default route to each of the advertised addresses. If the preference **ineligible**, or the address is not on an attached interface, the route is marked unusable but retained. If the preference is usable, the metric is set as a function of the preference such that the route with the best preference is used. If more than one address with the same preference is received, the one with the lowest IP address will be used. These default routes are not exportable to other protocols.

When a Router Advertisement with a zero lifetime is received, the host deletes all routes with next-hop addresses learned from that router. In addition, any routers learned from ICMP redirects pointing to these addresses will be deleted. The same will happen when a Router Advertisement is not received to refresh these routes before the lifetime expires.

g

The Router Discovery Client Statement

routerdiscovery client yes | **no** | **on** | **off** [**{**
  traceoptions *trace_options* **;**
  preference *preference* **;**
  interface *interface_list*
    [**enable**] | [**disable**]
    [**broadcast**] | [**multicast**]
    [**quiet**] | [**solicit**]
    ;
}] ;

traceoptions *trace_options*
Specifies the tracing options for OSPF. (See Trace Statements and the OSPF specific tracing options below.)

preference *preference* **;**
Specifies the preference of all Router Discovery default routes. The default is **55**.

interface *interface_list*
Specifies the parameters that apply to physical interfaces. Note a slight difference in convention from the rest of GateD, **interface** specifies just physical interfaces (such as **le0**, **ef0** and **en1**). The Router Discovery Client has no parameters that apply only to interface addresses.

enable Specifies that Router Discovery should be performed on the specified interface(s). This is the default.

disable Specifies that Router Discovery should not be performed on the specified interface(s).

broadcast Specifies that Router Solicitations should be broadcast on the specified interface(s). This is the default if IP multicast support is not available on this host or interface.

multicast Specifies that Router Solicitations should be multicast on the specified interface(s). If IP multicast is not available on this host and interface, no solicitation will be performed. The default is to multicast Router Solicitations if the host and interface support it. Otherwise Router Solicitations are broadcast.

quiet Specifies that no Router Solicitations will be sent on this interface, even though Router Discovery will be performed.

solicit Specifies that initial Router Solicitations will be sent on this interface. This is the default.

Tracing options

The Router Discovery Client and Server support the **state** trace flag which traces various protocol occurrences.

state State transitions

The Router Discovery Client and Server do not directly support any packet tracing options, tracing of router discovery packets is enabled via the ICMP Statement.

The Kernel Statement

While the kernel interface is not technically a routing protocol, it has many characteristics of one, and GateD handles it similarly to one. The routes GateD chooses to install in the kernel forwarding table are those that will actually be used by the kernel to forward packets.

The add, delete and change operations GateD must use to update the typical kernel forwarding table take a non-trivial amount of time. This does not present a problem for older routing protocols (RIP, EGP), which are not particularly time critical and do not easily handle very large numbers of routes anyway. The newer routing protocols (OSPF, BGP) have stricter timing requirements and are often used to process many more routes. The speed of the kernel interface becomes critical when these protocols are used.

g

To prevent GateD from locking up for significant periods of time installing large numbers of routes (up to a minute or more has been observed on real networks), the processing of these routes is now done in batches. The size of these batches may be controlled by the tuning parameters described below, but normally the default parameters will provide the proper functionality.

During normal shutdown processing, GateD normally deletes all the routes it has installed in the kernel forwarding table, except for those marked with **retain**. Optionally, GateD can leave all routes in the kernel forwarding table by not deleting any routes. In this case changes will be made to insure that routes with a **retain** indication are installed in the table. This is useful on systems with large numbers of routes as it prevents the need to re-install the routes when GateD restarts. This can greatly reduce the time it takes to recover from a restart.

Forwarding tables and Routing tables

The table in the kernel that controls the forwarding of packets is a *forwarding table*, also know in ISO speak as a *forwarding information base*, or *FIB*. The table that GateD uses internally to store routing information it learns from routing protocols is a *routing table*, known in ISO speak as a *routing information base*, or *RIB*. The routing table is used to collect and store routes from various protocols. For each unique combination of *network* and *mask* an *active* route is chosen, this route will be the one with the best (numerically smallest) *preference*. All the active routes are installed in the kernel forwarding table. The entries in this table are what the kernel actually uses to forward packets.

Updating the Forwarding Table

There are two main methods of updating the kernel *FIB*, the **ioctl()** interface and the *routing socket* interface. Their various characteristics are described here.

Updating the Forwarding Table with the ioctl interface

The **ioctl** interface to the forwarding table was introduced in *BSD 4.3* and widely distributed in *BSD 4.3*. This is a one-way interface, it only allows GateD to update the kernel forwarding table. It has several other limitations:

Fixed subnet masks
: The BSD 4.3 networking code assumed that all subnets of a given network had the same subnet mask. This limitation is enforced by the kernel. The network mask is not stored in the kernel forwarding table, but determined when a packet is forwarded by searching for interfaces on the same network.

One way interface
: GateD is able to update the kernel forwarding table, but it is not aware of other modifications of the forwarding table. GateD is able to listen to ICMP messages and guess how the kernel has updated the forwarding table with response to ICMP redirects.

Blind updates
: GateD is not able to detect changes to the forwarding table resulting from the use of the the *route* command by the system administrator. Use of the *route* command on systems that use the **ioctl()** interface is strongly discouraged while GateD is running.

Changes not supported
: In all known implementations, there is no change operation supported, to change a route that exists in the kernel, the route must be deleted and a new one added.

Updating the Forwarding Table with the routing socket interface

The *routing socket* interface to the kernel forwarding table was introduced in *BSD 4.3 Reno*, widely distributed in *BSD 4.3 Net/2* and improved in *BSD 4.4*. This interface is simply a socket, similar to a UDP socket, on which the kernel and GateD exchange messages. It has several advantages over the **ioctl()** interface:

Variable subnet masks
: The network mask is passed to the kernel explicitly. This allows different masks to be used on subnets of the same network. It also allows routes with masks that are more general than the natural mask to be used. This is known as classless routing.

Two way interface
: Not only is GateD able to change the kernel forwarding table with this interface, but the kernel can also report changes to the forwarding table to GateD. The most interesting of these is an indication that a redirect has modified the kernel forwarding table; this means that gated no longer needs to monitor ICMP messages to learn about redirects. Plus, there is an indication of whether the kernel processed the redirect, GateD can safely ignore redirect messages that the kernel did not process.

Updates visible
: Changes to the routing table by other processes, including the *route* command are received via the routing socket. This allows GateD to insure that the kernel forwarding table is in sync with the routing table. Plus it allows the system administrator the ability to do some operations with the *route* command while gated is running.

Changes supported
: There is a functioning *change* message that allows routes in the kernel to be atomically changed. Some early versions of the routing socket code had bugs in the change message processing. There are compilation time and configuration time options that cause delete and add sequences to be used in lieu of change messages.

Expandable
: New levels of kernel/GateD communications may be added by adding new message types.

Reading the Forwarding Table

When GateD starts up it reads the kernel forwarding table and installs corresponding routes in the routing table. These routes are called *remnant*s and are timed out after a configured interval (which defaults to 3 minutes), or as soon as a more attractive route is learned. This allows forwarding to occur during the time it takes the routing protocols to start learning routes.

There are three main methods for reading the forwarding table from the kernel.

Reading forwarding table via kmem

On many systems, especially those based on BSD 4.3, GateD must have knowledge of the kernel data structures and can go into the kernel to read the current state of forwarding table. This method is slow and subject to error if the kernel forwarding table is updated while GateD is in the middle of reading it. This can happen if the system administrator uses the *route* command, or an ICMP redirect message is received while GateD is starting up.

Due to an oversight some systems, such as OSF/1, which are based on BSD 4.3 Reno or later, do not have the **getkerninfo()** system call described below which allows GateD to read routes from the kernel without know about kernel internal structures. On these systems it is necessary to read the kernel radix tree from the kernel by poking around in kernel memory. This is even more error prone than reading the hash based forwarding table.

Reading the forwarding table via getkerninfo/sysctl

Besides the routing socket, BSD 4.3 Reno introduced the **getkerninfo()** system call. This call allows a user process (of which GateD is one) to read various information from the kernel without knowledge of the kernel data structures. In the case of the forwarding table, it is returned to gated atomically as a series of routing socket messages. This prevents the problem associated with the forwarding table changing while GateD is in the process of reading it.

g

BSD 4.4 changed the **getkerninfo()** interface into the **sysctl()** interface, which takes different parameters, but otherwise functions identically.

Reading the forwarding table via OS specific methods

Some operating systems, for example SunOS 5, define their own method of reading the kernel forwarding table. The SunOS 5 version is similar in concept to the **getkerninfo()** method.

Reading the interface list

The kernel support subsystem of GateD is responsible for reading the status of the kernel physical and protocol interfaces periodically. GateD detects changes in the interface list and notifies the protocols so they can start or stop instances or peers. The interface list is read one of two ways:

Reading the interface list with SIOCGIFCONF

On systems based on BSD 4.3, 4.3 Reno and 4.3 Net/2 the **SIOCGIFCONF ioctl** interface is used to read the kernel interface list. Using this method a list of interfaces and some basic information about them is return by the SIOCGIFCONF call. Other information must be learned by issuing other ioctls to learn the interface network mask, flags, MTU, metric, destination address (for point-to-point interfaces) and broadcast address (for broadcast capable interfaces).

GateD reads re-reads this list every 15 second looking for changes. When the routing socket is in use, it also re-reads it whenever a messages is received indicating a change in routing configuration. Receipt of a **SIGUSR2** signal also causes GateD to re-read the list. This interval may be explicitly configured in the interface configuration.

Reading the interface list with sysctl

BSD 4.4 added the ability to read the kernel interface list via the *sysctl* system call. The interface status is returned atomically as a list of routing socket messages which GateD parses for the required information.

BSD 4.4 also added routing socket messages to report interface status changes immediately. This allows GateD to react quickly to changes in interface configuration.

When this method is in use, GateD re-reads the interface list only once a minute. It also re-reads it on routing table changes indications and when a **SIGUSR2** is received. This interval may be explicitly configured in the interface configuration.

Reading interface physical addresses

Later version of the **getkerninfo()** and **sysctl()** interfaces return the interface physical addresses as part of the interface information. On most systems where this information is not returned, GateD scans the kernel physical interface list for this information for interfaces with IFF_BROADCAST set, assuming that their drivers are handled the same as Ethernet drivers. On some systems, such as *SunOS 4* and *SunOS 5*, system specific interfaces are used to learn this information

The interface physical addresses are useful for IS-IS, for IP protocols, they are not currently used, but may be in the future.

Reading kernel variables

At startup, GateD reads some special variables out of the kernel. This is usually done with the *nlist* (or *kvm_nlist*) system call, but some systems use different methods.

The variables read include the status of *UDP checksum creation and generation*, *IP forwarding* and *kernel version* (for informational purposes). On systems where the routing table is read directly from kernel memory, the root of the *hash table* or *radix tree routing table* is read. On systems where interface physical addresses are not supplied by other means, the root of the *interface list* is read.

Special route flags

The later *BSD* based kernel support the special route flags described here.

RTF_REJECT

Instead of forwarding a packet like a normal route, routes with **RTF_REJECT** cause packets to be dropped and **unreachable** messages to be sent to the packet originators. This flag is only valid on routes pointing at the *loopback* interface.

RTF_BLACKHOLE

Like the **RTF_REJECT** flag, routes with **RTF_BLACKHOLE** cause packets to be dropped, but **unreachable** messages are not sent. This flag is only valid on routes pointing at the *loopback* interface.

RTF_STATIC

When GateD starts, it reads all the routes currently in the kernel forwarding table. Besides interface routes, it usually marks everything else as a *remnant* from a previous run of GateD and deletes it after a few minutes. This means that routes added with the *route* command will not be retained after GateD has started.

To fix this the **RTF_STATIC** flag was added. When the *route* command is used to install a route that is not an interface route it sets the **RTF_STATIC** flag. This signals to GateD that said route was added by the systems administrator and should be retained.

Kernel Configuration

kernel {
  options
    [**nochange**]
    [**noflushatexit**]
    [**remnantholdtime** *time*]
    ;
  routes *number* **;**
  flash
    [**limit** *number*]
    [**type interface** | **interior** | **all**]
    ;
  background
    [**limit** *number*]
    [**priority flash** | **higher** | **lower**]
    ;
  traceoptions *trace_options* **;**
} ;

options *option_list*

Configure kernel options. The valid options are:

nochange

On systems supporting the routing socket this insures that changes operations will not be performed, only deletes and adds. This is useful on early versions of the routing socket code where the change operation was broken.

noflushatexit

During normal shutdown processing GateD deletes all routes from the kernel forwarding table that do not have a **retain** indication. The **noflushatexit** option prevents route deletions at shutdown. Instead, routes are changed and added to make sure that all the routes marked with **retain** get installed.

This is handy on systems with thousands of routes. Upon startup GateD will notice which routes are in the kernel forwarding table and not have add them back.

remnantholddimte *time*

Normally *remnant* routes read from the kernel forwarding table at startup are timed out in three minutes or as soon as they are overridden. This option allows the interval to be configured to a value between zero and 15 minutes. Setting it to zero causes these routes to be deleted immediately.

routes *number*

On some systems kernel memory is at a premium. With this parameter a limit can be placed on the maximum number of routes GateD will install in the kernel. Normally gated adds/changes/deletes routes in interface/internal/external order. It queues interface routes first, followed by internal routes, followed by external routes, and processes the queue from the beginning. If a this parameter is specified and the limit is hit, GateD does two scans of the list instead. On the first scan it does deletes, and also deletes all changed routes, turning the queued changes into adds. It then rescans the list doing adds in interface/internal/external order until it hits the limit again. This will tend to favor internal routes over external routes. The default is not to limit the number of routes in the kernel forwarding table.

flash

When routes change, the process of notifying the protocols is called a *flash update*. The kernel forwarding table interface is the first to be notified. Normally a maximum of 20 interface routes may be processed during one flash update. The **flash** command allows tuning of these parameters.

limit *number*
Specifies the maximum number of routes which may be processed during one flash update. The default is **20**. A value of **-1** will cause all pending route changes of the specified type to be processed during the flash update.

type interface | **interior** | **all**
Specifies the type of routes that will be processed during a flash update. **Interior** specifies that interior routes (See the definition of interior gateway protocols) will also be installed. **All** specifies the inclusion of exterior routes (See the definition of exterior gateway protocols) as well. The default is **interface** which specifies that only interface routes will be installed during a flash update.

Specifying **flash limit -1 all** causes all routes to be installed during the *flash update*; this mimics the behavior of previous versions of GateD.

background
Since only interface routes are normally installed during a flash update, the remaining routes are processed in batches in the background, that is, when no routing protocol traffic is being received. Normally, 120 routes are installed at a time to allow other tasks to be performed and this background processing is done at lower priority than flash updates the following parameters allow tuning of these parameters:

limit *number*
Specifies the number of route which may be processed at during one batch. The default is 120.

priority flash | **higher** | **lower**
Specifies the priority of the processing of batches of kernel updates in relationship to the *flash update* processing. The default is **lower** which means that flash updates are processed first. To process kernel updates at the same priority as flash updates, specify **flash**; to process them at a lower priority, use **lower**.

g

Tracing options
While the kernel interface is not technically a routing protocol, in many cases it is handled as one. The following two symbols make sense when entered from the command line since the code that uses them is executed before the trace file is parsed.

symbols
Symbols read from the kernel, by **nlist()** or similar interface.

iflist
Interface list scan. This option is useful when entered from the command line as the first interface list scan is performed before the configuration file is parsed.

The following tracing options may only be specified in the configuration file. They are not valid from the command line.

remnants
Routes read from the kernel when GateD starts.

request
Requests by GateD to Add/Delete/Change routes in the kernel forwarding table.

Static Statements
Static statements define the static routes used by GateD. A single **static** statement can specify any number routes. The **static** statements occur after protocol statements and before control statements in the gated.conf file. Any number of **static** statements may be specified, each containing any number of static route definitions. These routes can be overridden by routes with better preference values.

static {
(**host** *host*) | **default** |
(*network* [(**mask** *mask*) | (**masklen** *number*)])
gateway *gateway_list*
[**interface** *interface_list*]
[**preference** *preference*]
[**retain**]

[**reject**]
[**blackhole**]
[**noinstall**] **;**
(*network* [(**mask** *mask*) | (**masklen** *number*)])
interface *interface*
[**preference** *preference*]
[**retain**]
[**reject**]
[**blackhole**]
[**noinstall**] **;**
} ;

host *host* **gateway** *gateway_list*
(*network* [(**mask** *mask*) | (**masklen** *number*)])
default gateway *gateway_list*

This is the most general form of the static statement. It defines a static route through one or more gateways. Static routes are installed when one or more of the **gateways** listed are available on directly attached interfaces. If more than one eligible gateways are available, they are limited by the number of multipath destinations supported (this compile time parameter is currently almost always one on Unix).

Parameters for static routes are:

interface *interface_list*

When this parameter is specified, gateways are only considered valid when they are on one of these interfaces.See the section on interface list specification for the description of the *interface_list*.

preference *preference*

This option selects the preference of this static route. The preference controls how this route competes with routes from other protocols. The default preference is 60.

retain

Normally gated removes all routes except interface routes from the kernel forwarding table during a graceful shutdown. The **retain** option may be used to prevent specific static routes from being removed. This is useful to insure that some routing is available when gated is not running.

reject

Instead of forwarding a packet like a normal route, **reject** routes cause packets to be dropped and **unreachable** messages to be sent to the packet originators. Specifying this option causes this route to be installed as a reject route. Not all kernel forwarding engines support reject routes.

blackhole

A **blackhole** route is the same as a **reject** route except that **unreachable** messages are not supported.

noinstall

Normally the route with the lowest preference is installed in the kernel forwarding table and is the route exported to other protocols. When **noinstall** is specified on a route, it will not be installed in the kernel forwarding table when it is active, but it will still be eligible to be exported to other protocols.

(*network* [(**mask** *mask*) | (**masklen** *number*)]) **interface** *interface*

This form defines a static interface route which is used for primitive support of multiple network addresses on one interface. The **preference**, **retain**, **reject**, **blackhole** and **noinstall** options are the same as described above.

Control Statements Overview

Control statements control routes that are imported from routing peers and routes that are exported to these peers. These are the final statements to be included in the gated.conf file. The control statements are:

- the Import Statement
- the Export Statement

- the Aggregate Statement
- the Generate Statement

Route Filtering

Routes are filtered by specifying configuration language that will match a certain set of routes by destination, or by destination and mask. Among other places, route filters are used on **martians**, **import** and **export** statements.

The action taken when no match is found is dependent on the context, for instance **import** and **export** route filters assume an **all reject ;** at the end a list.

A route will match the most specific filter that applies. Specifying more than one filter with the same destination, mask and modifiers will generate an error.

Filtering syntax

network [**exact** | **refines**]
network **mask** *mask* [**exact** | **refines**]
network **masklen** *number* [**exact** | **refines**]
all
default
host *host*

These are all the possible formats for a route filter. Not all of these formats are available in all places, for instance the **host** and **default** formats are not valid for **martians**.

In most cases it is possible to specify additional parameters relevant to the context of the filter. For example, on a **martian** statement it is possible to specify the **allow** keyword, on an **import** statement you can specify a preference, and on a **export** you can specify a metric.

network [**exact** | **refines**]
network **mask** *mask* [**exact** | **refines**]
network **masklen** *number* [**exact** | **refines**]

Matching usually requires both an address and a mask, although the mask is implied in the shorthand forms listed below. These three forms vary in how the mask is specified. In the first form, the mask is implied to be the natural mask of the network. In the second, the mask is explicitly specified. In the third, the mask is specified by the number of contiguous one bits.

If no additional parameters are specified, any destination that falls in the range given by the network and mask is matched. The mask of the destination is ignored. If a *natural* network is specified, the network, any subnets, and any hosts will be match. The two optional modifiers cause the mask of the destination to be considered also:

exact

This parameter specifies that the mask of the destination must match the supplied mask *exactly*. This is used to match a network, but no subnets or hosts of that network.

refines

Specifies that the mask of the destination must be more specified (longer) than the filter mask. This is used to match subnets and/or hosts of a network, but not the network.

all This entry matches anything. It is equivalent to:

0.0.0.0 mask 0.0.0.0

default

Matches the **default** route. To match, the address must be the default address and the mask must be all zeros. This is equivalent to:

0.0.0.0 mask 0.0.0.0 exact

host *host*

Matches the specific host. To match, the address must exactly match the specified *host* and the network mask must be a host mask (all ones). This is equivalent to:

host **mask 255.255.255 exact**

Matching AS paths

An AS path is a list of autonomous_systems that routing information has passed through to get to this router, and an indicator of the origin of the AS path. This information can be used to prefer one path to a destination network over another. The primary method for doing this with GateD is to specify a list of patterns to be applied to AS paths when *importing* and *exporting* routes.

Each autonomous system that a route passed through prepends its AS number to the beginning of the AS path.

The origin information details the completeness of AS path information. An origin of **igp** indicates the route was learned from an interior routing protocol and is most likely complete. An origin of **egp** indicates the route was learned from an exterior routing protocol that does not support AS paths (EGP for example) and the path is most likely not complete. When the path information is definitely not complete, an origin of **incomplete** is used.

AS path regular expressions are defined in RFC 1164 section 4.2.

AS path matching syntax

An AS path is matched using the following syntax.

aspath *aspath_regexp* **origin any** | ([**igp**] [**egp**] [**incomplete**])

This specifies that an AS matching the *aspath_regexp* with the specified origin is matched.

AS path regular expressions

Technically, an AS path regular expression is a regular expression with the alphabet being the set of AS numbers. An AS path regular expression is composed of one or more AS paths expressions. An AS path expressions is composed of AS path terms and AS path operators.

AS path terms

An AS path term is one of the following three objects:

autonomous_system
.
(*aspath_regexp* **)**

where

autonomous_system	Any valid autonomous system number, from one through 65534 inclusive.
.	Matches any autonomous system number.
(*aspath_regexp* **)**	Parentheses group subexpressions--an operator, such as ***** or **?** works on a single element or on a regular expression enclosed in parentheses

AS path operators

An AS path operator is one of the following:

aspath_term **{m,n}**
aspath_term **{m}**
aspath_term **{m,}**
aspath_term *****
aspath_term **+**
aspath_term **?**
aspath_term **|** *aspath_term*

aspath_term **{m,n}**
a regular expression followed by **{m,n}** (where m and n are both non-negative integers and m <= n) means at least *m* and at most *n* repetitions.

aspath_term **{m}**
a regular expression followed by **{m}** (where m is a positive integer) means exactly *m* repetitions.

aspath_term **{m,}**
a regular expression followed by **{m,}** (where m is a positive integer) means m or more repetitions.

aspath_term *****
an AS path term followed by ***** means *zero or more* repetitions. This is shorthand for **{0,}**.

aspath_term **+**
a regular expression followed by **+** means one or more repetitions. This is shorthand for **{1,}**.

aspath_term **?**
a regular expression followed by **?** means zero or one repetition. This is shorthand for **{0,1}**.

aspath_term **|** *aspath_term*
matches the AS term on the left, or the AS term on the right.

The Import Statement

Importation of routes from routing protocols and installation of the routes in the GateD routing database is controlled by **import** statements. The format of an **import** statement varies depending on the source protocol.

Specifying preferences

In all cases, one of two keywords may be specified to control how routes compete with other protocols:

restrict
preference *preference*

restrict
Specifies that the routes are not desired in the routing table. In some cases this means that the routes are not installed in the routing table. In others it means that they are installed with a negative preference; this prevents them from becoming *active* so they will not be installed in the forwarding table, or exported to other protocols.

preference *preference*
Specifies the preference value used when comparing this route to other routes from other protocols. The route with the lowest preference available at any given route becomes the *active* route, is installed in the forwarding table, and is eligible to be exported to other protocols. The default preferences are configured by the individual protocols.

g

Route Filters

All the formats allow route filters as shown below. See the section on route filters for a detailed explanation of how they work. When no route filtering is specified (when **restrict** is specified on the first line of a statement), all routes from the specified source will match that statement. If any filters are specified, only routes that match the specified filters will be imported. Put differently, if any filters are specified, an **all restrict ;** is assumed at the end of the list.

network [**exact** | **refines**]
network **mask** *mask* [**exact** | **refines**]
network **masklen** *number* [**exact** | **refines**]
default
host *host*

Importing routes from BGP and EGP

import proto bgp | **egp autonomoussystem** *autonomous_system*
  restrict ;
import proto bgp | **egp autonomoussystem** *autonomous_system*
  [**preference** *preference*] **{**
  route_filter [**restrict** | (**preference** *preference*)] **;**
} ;

import proto bgp aspath *aspath_regexp*
  origin any | ([**igp**] [**egp**] [**incomplete**])
  restrict ;
import proto bgp aspath *aspath_regexp*
  origin any | ([**igp**] [**egp**] [**incomplete**])
  [**preference** *preference*] **{**
  route_filter [**restrict** | (**preference** *preference*)] **;**
} ;

EGP importation may be controlled by autonomous system. BGP also supports controlling propagation by the use of an AS path regular expressions, which are documented in the section on Matching AS paths. Note that EGP and BGP versions 2 and 3 only support the propagation of *natural* networks, so the **host** and **default** route filters are meaningless. BGP version 4 supports the propagation of any destination along with a *contiguous* network mask.

EGP and BGP both store any routes that were rejected implicitly by not being mentioned in a route filter, or explicitly with the **restrict** keyword in the routing table with a negative preference. A negative preference prevents a route from becoming active, which prevents it from being installed in the forwarding table, or exported to other protocols. This alleviates the need to break and re-establish a session upon reconfiguration if importation policy is changed.

Importing routes from an RIP, HELLO and Redirects

import proto rip | **hello** | **redirect**
[(**interface** *interface_list*) | (**gateway** *gateway_list*)]
restrict ;
import proto rip | **hello** | **redirect**
[(**interface** *interface_list*) | (**gateway** *gateway_list*)]
[**preference** *preference*] **{**
route_filter [**restrict** | (**preference** *preference*)] **;**
} ;

The importation of RIP, HELLO and Redirect routes may be controlled by any of protocol, source interface and source gateway. If more than one is specified, they are processed from most general (protocol) to most specific (gateway).

RIP and HELLO do not support the use of preference to choose between routes of the same protocol. That is left to the protocol metrics. These protocols do not save routes that were rejected since they have short update intervals.

Importing routes from OSPF

import proto ospfase [**tag** *ospf_tag*] **restrict ;**
import proto ospfase [**tag** *ospf_tag*]
[**preference** *preference*] **{**
route_filter [**restrict** | (**preference** *preference*)] **;**
} ;

Due to the nature of OSPF, only the importation of ASE routes may be controlled. OSPF intra- and inter-area routes are always imported into the gated routing table with a preference of 10. If a tag is specified, the import clause will only apply to routes with the specified tag.

It is only possible to restrict the importation of OSPF ASE routes when functioning as an AS border router. This is accomplished by specifying an **export ospfase** clause. Specification of an empty export clause may be used to restrict importation of ASEs when no ASEs are being exported.

Like the other interior protocols, preference can not be used to choose between OSPF ASE routes, that is done by the OSPF costs. Routes that are rejected by policy are stored in the table with a negative preference.

The Export Statement

The **import** statement controls which routes received from other systems are used by GateD, and the **export** statement controls which routes are advertised by GateD to other systems. Like the **import** statement, the syntax of the **export** statement varies slightly per protocol. The syntax of the **export** statement is similar to the syntax of the **import** statement, and the meanings of many of the parameters are identical. The main difference between the two is that while route importation is just controlled by source information, route exportation is controlled by both destination and source.

The outer portion of a given **export** statement specifies the destination of the routing information you are controlling. The middle portion restricts the sources of importation that you wish to consider. And the innermost portion is a route filter used to select individual routes.

Specifying metrics

The most specific specification of a metric is the one applied to the route being exported. The values that may be specified for a metric depend on the destination protocol that is referenced by this export statement.

restrict
metric *metric*

restrict
Specifies that nothing should be exported. If specified on the destination portion of the **export**

statement, it specifies that nothing at all should be exported to this destination. If specified on the source portion, it specifies that nothing from this source should be exported to this destination. If specified as part of a route filter, it specifies that the routes matching that filter should not be exported.

metric *metric*
Specifies the metric to be used when exporting to the specified destination.

Route Filters

All the formats allow route filters as shown below. See the section on route filters for a detailed explanation of how they work. When no route filtering is specified (when **restrict** is specified on the first line of a statement), all routes from the specified source will match that statement. If any filters are specified, only routes that match the specified filters will be exported. Put differently, if any filters are specified, an **all restrict ;** is assumed at the end of the list.

network [**exact** | **refines**]
network **mask** *mask* [**exact** | **refines**]
network **masklen** *number* [**exact** | **refines**]
default
host *host*

g

Specifying the destination

As mentioned above, the syntax of the **export** statement varies depending on the protocol it is being applied to. One thing that applies in all cases is the specification of a metric. All protocols define a default metric to be used for routes being exported, in most cases this can be overridden at several levels of the export statement.

The specification of the source of the routing information being exported (the export_list) is described below.

Exporting to EGP and BGP

export proto bgp | **egp as** *autonomous system*
  restrict ;
export proto bgp | **egp as** *autonomous system*
  [**metric** *metric*] **{**
  export_list **;**
} ;

Exportation to EGP and BGP is controlled by autonomous system, the same policy is applied to all routers in the AS. EGP metrics range from 0 to 255 inclusive with 0 being the most attractive.

BGP metrics are 16 bit unsigned quantities. They range from 0 to 65535 inclusive with 0 being the most attractive. While BGP version 4 actually supports 32 bit unsigned quantities, GateD does not yet support this.

If no export policy is specified, only routes to attached interfaces will be exported. If any policy is specified, the defaults are overridden; it is necessary to explicitly specify everything that should be exported.

Note that EGP and BGP versions 2 and 3 only support the propagation of *natural* networks, so the **host** and **default** route filters are meaningless. BGP version 4 supports the propagation of any destination along with a *contiguous* network mask.

Exporting to RIP and HELLO

export proto rip | **hello**
  [(**interface** *interface_list*) | (**gateway** *gateway_list*)]
  restrict ;
export proto rip | **hello**
  [(**interface** *interface_list*) | (**gateway** *gateway_list*)]
  [**metric** *metric*] **{**
  export_list **;**
} ;

Exportation to RIP and HELLO is controlled by any of protocol, interface or gateway. If more than one is specified, they are processed from most general (protocol) to most specific (gateway).

It is not possible to set metrics for exporting RIP routes into RIP, or exporting HELLO routes into HELLO. Attempts to do this are silently ignored.

If no export policy is specified, RIP and interface routes are exported into RIP and HELLO and interface routes are exported into HELLO. If any policy is specified, the defaults are overridden. It is necessary to explicitly specify everything that should be exports.

RIP version 1 and HELLO assume that all subnets of the shared network have the same subnet mask so they are only able to propagate subnets of that network. RIP version 2 removes that restriction and is capable of propagating all routes when not sending version 1 compatible updates.

To announce routes which specify a next hop of the loopback interface (static and internally generated default routes) via RIP or HELLO, it is necessary to specify the metric at some level in the **export** clause. Just setting a default metric for RIP or HELLO is not sufficient. This is a safeguard to verify that the announcement is intended.

Exporting to OSPF

export proto osfpase [**type 1** | **2**] [**tag** *ospf_tag*]
  restrict ;
export proto osfpase [**type 1** | **2**] [**tag** *ospf_tag*]
  [**metric** *metric*] **{**
  export_list **;**
} ;

It is not possible to create OSPF intra- or inter-area routes by exporting routes from the GateD routing table into OSPF. It is only possible to export from the GateD routing table into OSPF ASE routes. It is also not possible to control the propagation of OSPF routes within the OSPF protocol.

There are two types of OSPF ASE routes, *type 1* and *type 2*, see the OSPF protocol configuration for a detailed explanation of the two types. The default type is specified by the **defaults** subclause of the **ospf** clause. This may be overridden by a specification on the **export** statement.

OSPF ASE routes also have the provision to carry a *tag*. This is an arbitrary 32 bit number that can be used on OSPF routers to filter routing information. See the OSPF protocol configuration for detailed information on OSPF tags. The default tag specified by the **ospf defaults** clause may be overridden by a tag specified on the **export** statement.

Specifying the source

The export list specifies export based on the origin of a route and the syntax varies depending on the source.

Exporting BGP and EGP routes

proto bgp | **egp autonomoussystem** *autonomous_system*
  restrict ;
proto bgp | **egp autonomoussystem** *autonomous_system*
  [**metric** *metric*] **{**
  route_filter [**restrict** | (**metric** *metric*)] **;**
} ;

BGP and EGP routes may be specified by source autonomous system. All routes may be exported by as path, see below for more information.

Exporting RIP and HELLO routes

proto rip | **hello**
  [(**interface** *interface_list*) | (**gateway** *gateway_list*)]
  restrict ;
proto rip | **hello**
  [(**interface** *interface_list*) | (**gateway** *gateway_list*)]
  [**metric** *metric*] **{**
  route_filter [**restrict** | (**metric** *metric*)] **;**
} ;

RIP and HELLO routes may be exported by protocol, source interface and/or source gateway.

Exporting OSPF routes

proto ospf | **ospfase restrict ;**
proto ospf | **ospfase** [**metric** *metric*] **{**
  route_filter [**restrict** | (**metric** *metric*)] **;**
} ;

g

Both OSPF, and OSPF ASE routes may be exported into other protocols. See below for information on exporting by tag.

Exporting routes from non-routing protocols
Non-routing with interface
proto direct | **static** | **kernel**
[(**interface** *interface_list*)]
restrict ;
proto direct | **static** | **kernel**
[(**interface** *interface_list*)]
[**metric** *metric*] **{**
route_filter [**restrict** | (**metric** *metric*)] **;**
} ;

These protocols may be exported by protocol, or by the interface of the next hop. These protocols are:

direct
Routes to directly attached interfaces.

static
Static routes specified in a **static** clause.

kernel
On systems with the *routing socket*, routes learned from the routing socket are installed in the GateD routing table with a protocol of *kernel*. These routes may be exported by referencing this protocol. This is useful when it is desirable to have a script install routes with the **route** command and propagate them to other routing protocols.

Non-routing by protocol
proto default | **aggregate**
restrict ;
proto default | **aggregate**
[**metric** *metric*] **{**
route_filter [**restrict** | (**metric** *metric*)] **;**
} ;

These protocols may only be referenced by protocol.

default
Refers to routes created by the **gendefault** option. It is recommended that route generation be used instead.

aggregate
Refers to routes synthesized from other routes when the **aggregate** and **generate** statements are used. See the section on Route Aggregation for more information.

Exporting by AS path
proto *proto* | **all aspath** *aspath_regexp*
origin any | ([**igp**] [**egp**] [**incomplete**])
restrict ;
proto *proto* | **all aspath** *aspath_regexp*
origin any | ([**igp**] [**egp**] [**incomplete**])
[**metric** *metric*] **{**
route_filter [**restrict** | (**metric** *metric*)] **;**
} ;

When BGP is configured, all routes are assigned an AS path when they are added to the routing table. For all interior routes this AS path specifies IGP as the origin and no ASes in the AS path (the current AS is added when the route is exported). For EGP routes this AS path specifies EGP as the origin and the source AS as the AS path. For BGP routes, the AS path is stored as learned from BGP.

AS path regular expressions are documented in the section on Matching AS paths.

Exporting by route Tag
proto *proto* | **all tag** *tag* **restrict ;**
proto *proto* | **all tag** *tag*
[**metric** *metric*] **{**

g

route_filter [**restrict** | (**metric** *metric*)] **;**
} ;

Both OSPF and RIP version 2 currently support tags, all other protocols always have a tag of zero. The source of exported routes may be selected based on this tag. This is useful when routes are classified by tag when the are exported into a given routing protocol.

Route Aggregation

Route aggregation is a method of generating a more general route given the presence of a specific route. It is used, for example, at an autonomous system border to generate a route to a network to be advertised via EGP given the presence of one or more subnets of that network learned via RIP. Older versions of GateD automatically performed this function, generating an aggregate route to a *natural* network (using the old Class A, B and C concept) given an interface to a subnet of that *natural* network. However that was not always the correct thing to do, and with the advent of classless interdomain routing it is even more frequently the wrong thing to do, so aggregation must be explicitly configured. No aggregation is performed unless explicitly requested in an **aggregate** statement.

g

Route aggregation is also used by regional and national networks to reduce the amount of routing information passed around. With careful allocation of network addresses to clients, regional networks can just announce one route to regional networks instead of hundreds.

Aggregate routes are not actually used for packet forwarding by the originator of the aggregate route, only by the receiver (if it wishes). A router receiving a packet which does not match one of the component routes which led to the generation of an aggregate route is supposed to respond with an ICMP *network unreachable* message. This is to prevent packets for unknown component routes from following a default route into another network where they would be forwarded back to the border router, and around and around again and again, until their TTL expires. Sending an unreachable message for a missing piece of an aggregate is only possible on systems with support for reject routes.

A slight variation of aggregation is the generation of a route based on the existence of certain conditions. This is sometimes known as the *route of last resort*. This route inherits the next hops and aspath from the contributor specified with the lowest (most favorable) preference. The most common usage for this is to generate a default based on the presence of a route from a peer on a neighboring backbone.

Aggregation and Generation syntax

aggregate default
  | (*network* [(**mask** *mask*) | (**masklen** *number*)])
  [**preference** *preference*] [**brief**] **{**
  proto [**all** | **direct** | **static** | **kernel** | **aggregate** | *proto*]
    [(**as** *autonomous system*) | (**tag** *tag*)
      | (**aspath** *aspath_regexp*)]
    restrict ;
  proto [**all** | **direct** | **static** | **kernel** | **aggregate** | *proto*]
    [(**as** *autonomous system*) | (**tag** *tag*)
      | (**aspath** *aspath_regexp*)]
    [**preference** *preference*] **{**
    route_filter [**restrict** | (**preference** *preference*)] **;**
  } ;
} ;

generate default
  | (*network* [(**mask** *mask*) | (**masklen** *number*)])
  [**preference** *preference*] **{**
    [(**as** *autonomous system*) | (**tag** *tag*)
      | (**aspath** *aspath_regexp*)]
    restrict ;
  proto [**all** | **direct** | **static** | **kernel** | **aggregate** | *proto*]
    [(**as** *autonomous system*) | (**tag** *tag*)
      | (**aspath** *aspath_regexp*)]
    [**preference** *preference*] **{**
    route_filter [**restrict** | (**preference** *preference*)] **;**
  } ;
} ;

Routes that match the route filters are called *contributing* routes. They are ordered according to the aggregation preference that applies to them. If there are more than one contributing routes with the same aggregating preference, the preferences of the route are used to order the routes. The preference of the aggregate route will be that of the contributing route with the lowest aggregate preference.

preference *preference*
Specifies the preference to assign to the resulting aggregate route. The default preference is 130.

brief
Used to specify that the AS path should be truncated to the longest common AS path. The default is to build an AS path consisting of SETs and SEQUENCEs of all contributing AS paths.

proto *proto*
In addition to the special protocols listed, the contributing protocol may be chosen from among any of the ones supported (and currently configured into) GateD.

as *autonomous_system*
Restrict selection of routes to those learned from the specified autonomous system.

tag *tag*
Restrict selection of routes to those with the specified tag.

aspath *aspath_regexp*
Restrict selection of routes to those that match the specified AS path.

restrict
Indicates that these routes are not to be considered as contributors of the specified aggregate. The specified protocol may be any of the protocols supported by GateD.

route_filter
See below.

A route may only contribute to an aggregate route which is more general than itself; it must match the aggregate under its mask. Any given route may only contribute to one aggregate route, which will be the most specific configured, but an aggregate route may contribute to a more general aggregate.

Route Filters

All the formats allow route filters as shown below. See the section on route filters for a detailed explanation of how they work. When no route filtering is specified (when `restrict` is specified on the first line of a statement), all routes from the specified source will match that statement. If any filters are specified, only routes that match the specified filters will be considered as contributors. Put differently, if any filters are specified, an `all restrict ;` is assumed at the end of the list.

network [**exact** | **refines**]
network **mask** *mask* [**exact** | **refines**]
network **masklen** *number* [**exact** | **refines**]
default
host *host*

Glossary of Terms

Terms used in descriptions throughout this document are defined below:

adjacency
A relationship formed between selected neighboring routers for the purpose of exchanging routing information. Not every pair of neighboring routers becomes adjacent.

autonomous system
A set of routers under a single technical administration, using an interior gateway protocol and common metrics to route packets within the AS, and using an exterior gateway protocol to route packets to other ASs. Since this classic definition was developed, it has become common for a single AS to use several interior gateway protocols and sometimes several sets of metrics within an AS. The use of the term Autonomous System stresses that even when multiple igps and metrics are used, the administration of an AS appears to other ASs to have a single coherent interior routing plan and presents a consistent picture of what networks are reachable through it. The AS is represented by a number between 1 and 65534, assigned by the Internet Assigned Numbers Authority.

BGP
Border Gateway Protocol
One of a class of exterior gateway protocols, described in more detail in the BGP section of the

Protocol Overview.

cost An OSPF metric. See metric.

delay
A HELLO metric. Valid values are from zero to 30000 inclusive. The value of 30000 is the maximum metric and means unreachable. See metric.

designated router
OSPF: Each multiaccess network that has at least two attached routers as a designated router. The designated router generates a link state advertisement for the multiaccess network and assists in running the protocol. The designated router is elected by the HELLO protocol.

destination
Any network or any host.

distance
An EGP metric. See metric. Valid values are from zero to 255 inclusive.

egp
exterior gateway protocol
exterior routing protocol
A class of routing protocols used to exchange routing information within an autonomous system. A detailed explanation of **exterior gateway protocols** is available in the Protocol Overview.

EGP
Exterior Gateway Protocol
One of a class of exterior gateway protocols, described in more detail in the EGP section of the Protocol Overview.

gateway
An intermediate destination by which packets are delivered to their ultimate destination. A **host** address of another router that is directly reachable via an attached network. As with any host address it may be specified symbolically.

gateway_list
A list of one or more **gateway**s separated by white space.

HELLO
One of a class of interior gateway protocols, described in more detail in the HELLO section of the Protocol Overview.

host
The IP address of any host. Usually specified as a dotted quad, four values in the range of 0 to 255 inclusive separated by dots (**.**). For example **132.236.199.63** or **10.0.0.51**. It may also be specified as an eight digit hexidecimal string preceded by **0x**. For example **0x????????** or **0x0a000043**. Finally, if **options noresolv** is not specified, a symbolic hostname. For example **gated.cornell.edu** or **nic.ddn.mil**. The numeric forms are much preferred over the symbolic form.

interface
The **host** address of an attached interface. This is the address of a *broadcast*, *nbma* or *loopback* interface and the remote address of a *point-to-point* interface. As with any host address it may be specified symbolically.

interface
The connection between a router and one of its attached networks. A physical interface may be specified by a single IP address, domain name, or interface name. (Unless the network is an unnumbered point-to-point network.) Multiple levels of reference in the configuration language allow identification of interfaces using wildcard, interface type name, or delete word address. Be careful with the use of interface names as future Unix operating systems may allow more than one address per interface. Dynamic interfaces can be added or deleted and indicated as up or down as well as changes to address, netmask and metric parameters.

igp
interior gateway protocol
interior routing protocol
One of a class of routing protocols used to exchange routing information within an autonomous system. A detailed explanation of **interior gateway protocols** is available in the Protocol Overview.

interface_list
A list of one or more interface names including wildcard names (names without a number) and names which may specify more than one interface or address, or the token "all" for all interfaces. See the section on interface lists for more information.

IS-IS
One of a class of interior gateway protocols.

local_address
The **host** address of an attached interface. This is the address of a *broadcast*, *nbma* or *loopback* interface and the local address of a *point-to-point* interface. As with any host address it may be specified symbolically.

mask
A means of subdividing networks using address modification. A mask is a dotted quad specifying which bits of the destination are significant. Except when used in a route filter, GateD only supports contiguous masks.

g

mask length
The number of significant bits in the mask.

metric
One of the units used to help a system determine the best route. Metrics may be based on hop count, routing delay, or an arbitrary value set by the administrator depending on the type of routing protocol. Routing metrics may influence the value of assigned internal preferences. (See preference.)

This sample table shows the range of possible values for each routing protocol metric and the value used by each protocol (See Protocol Overview.) to reach a destination.

```
SAMPLE ROUTING PROTOCOL METRICS
Protocol  Metric Represents     Range    Unreachable
--------  -----------------     -----    -----------
RIP       distance (hop-count)  0-15            16
HELLO     delay (milliseconds)  0-29999      30000
OSPF      cost of path          0-?????     Delete
ISIS      cost of path          0-254       Delete
EGP       distance (unused)     0-65535        255
BGP       unspecified           0-65534      65535
```

multiaccess networks
Those physical networks that support the attachment of multiple (more than two) routers. Each pair of routers on such a network is assumed to be able to communicate directly.

natural mask
The format of an IP address contains the network address and the host address. The natural mask is a default value applied to the 3 class addresses:
0xff000000 for class A (network.host.host.host),
0xffff0000 for class B (network.network.host.host) and
0xffffff00 for class C (network.network.network.host).

neighbor
Another router which with implicit or explicit communication is established by a routing protocol. Neighbors are usually on a shared network, but not always. This term is mostly used in OSPF and EGP. Usually synonymous with *peer*.

neighboring routers
Two routers that have interfaces to a common network. On multiaccess networks, routers are dynamically discovered by the OSPF HELLO protocol.

network
Any packet-switched network. A network may be specified by its IP address or network name. The host bits in a network specification must be zero. *Default* may be used to specify the default network (0.0.0.0).

network
The IP address of a network. Usually specified as a dotted quad, one to four values in the range of 0 to 255 inclusive separated by dots (**.**). For example **132.236.199**, **132.236** or **10**. It may also be specified as a hexidecimal string preceded by **0x** with an even number of digits between two and eight. For example **0x??????**, **0x????** or **0x0a**. Also allowed is the symbolic value **default** which

has the distinguished value **`0.0.0.0`**, the default network. If **`options noresolv`** is not specified, a symbolic network name can be used, for example **`nr-tech-prod`**, **`cornellu-net`** and **`arpanet`**. The numeric forms are much preferred over the symbolic form.

number
A positive integer.

OSPF
Open Shortest Path First
One of a class of interior gateway protocols, described in more detail in the OSPF section of the Protocol Overview.

ospf_area
peer
Another router which with implicit or explicit communication is established by a routing protocol. Peers are usually on a shared network, but not always. This term is mostly used by BGP. Usually synonymous with *neighbor*.

port
A UDP or TCP port number. Valid values are from 1 through 65535 inclusive.

preference
A **`preference`** is a value between 0 (zero) and 255 used to select between many routes to the same destination. The route with the best (numerically lowest) preference is as the *active* route. The active route is the one installed in the kernel forwarding table and *exported* to other protocols. Preference zero is usually reserved for routes to directly attached interfaces. A default preference is assigned to each source from which GateD receives routes. (See Preference.)

prefix
A contiguous mask covering the most significant bits of an address. The prefix length specifies how many bits are covered.

QoS
quality of service
The OSI equivalent of TOS.

RIP
Routing Information Protocol
One of a class of interior gateway protocols, described in more detail in the RIP section of the Protocol Overview.

router id
A 32-bit number assigned to each router running the OSPF protocol. This number uniquely identifies the router within the autonomous system.

router_id
An IP address used as unique identifier assigned to represent a specific router. This is usually the address of an attached interface.

RIB
routing information base
routing database
routing table
The repository of all of the GateD retained routing information, used to make decisions and as a source for routing information which is propagated.

simplex
An interface may be marked as simplex either by the kernel, or by interface configuration. A simplex interface is an interface on a broadcast media that is not capable of receiving packets it broadcasts.

GateD takes advantage of interfaces that are capable of receiving their own broadcast packets to monitor whether an interface appears to be functioning properly.

time
A time value, usually a time interval. It may be specified in any one of the following forms:

number A non-negative decimal number of seconds. For example, **`27`**, **`60`** or **`3600`**.

g

number:*number* — A non-negative decimal number of minutes followed by a seconds value in the range of zero to 59 inclusive. For example, **0:27**, **1:00** or **60:00**.

number:*number*:*number* — A non-negative decimal number of hours followed by a minutes value in the range of zero to 59 inclusive followed by a seconds value in the range of zero to 59 inclusive. For example, **0:00:27**, **0:01:00** or **1:00:00**.

time to live

ttl The *Time To Live (TTL)* of an IP packet. Valid values are from one (1) through 255 inclusive.

TOS

type of service

The *type of service* is for internet service quality selection. The type of service is specified along the abstract parameters precedence, delay, throughput, reliability, and cost. These abstract parameters are to be mapped into the actual service parameters of the particular networks the datagram traverses. The vast majority of IP traffic today uses the default type of service.

g

WARNINGS

gated contains provisions for BGP protocol, but it is not officially supported by HP at the present time. The route aggregation and generation statements which generate a more general route from compressing the specific routes through the explicit configuration are not supported in this release.

AUTHORS

See *gated*(1M).

SEE ALSO

RFC 827: E. Rosen, *Exterior Gateway Protocol EGP*

RFC 891: D. Mills, *DCN local-network protocols*

RFC 904: D. Mills, *Exterior Gateway Protocol formal specification*

RFC 1058: C. Hedrick, *Routing Information Protocol*

RFC 1105: K. Lougheed, Y. Rekhter, *Border Gateway Protocol BGP*

RFC 1163: K. Lougheed, Y. Rekhter, *A Border Gateway Protocol (BGP)*

RFC 1164: J. Honig, D. Katz, M. Mathis, Y. Rekhter, J. Yu, *Application of the Border Gateway Protocol in the Internet*

RFC 1227: M. Rose, *SNMP MUX Protocol and MIB.*

RFC 1245: J. Moy, *OSPF Protocol Analysis*

RFC 1246: J. Moy, *Experience with the OSPF Protocol*

RFC 1253: F. Baker, R. Coltun, *OSPF Version 2 Management Information Base*

RFC 1256: S. Deering, *ICMP Router Discovery Messages*

RFC 1265: Y. Rekhter, *BGP Protocol Analysis*

RFC 1266: Y. Rekhter, *Experience with the BGP Protocol*

RFC 1267: K. Lougheed, Y. Rekhter, *A Border Gateway Protocol 3 (BGP-3)*

RFC 1268: P. Gross, Y. Rekhter, *Application of the Border Gateway Protocol in the Internet*

RFC 1269: J. Burruss, S. Willis, *Definitions of Managed Objects for the Border Gateway Protocol (Version 3)*

RFC 1321: R. Rivest, *The MD5 Message-Digest Algorithm*

RFC 1370: Internet Architecture Board *Applicability Statement for OSPF*

RFC 1388: G. Malkin, *RIP Version 2 Carrying Additional Information*

RFC 1397: D. Haskin, *Default Route Advertisement In BGP2 And BGP3 Versions Of The Border Gateway Protocol*

RFC 1403: K. Varadhan, *BGP OSPF Interaction*

RFC 1583: J. Moy, *OSPF Version 2*

g

g

NAME

gettydefs - speed and terminal settings used by getty

DESCRIPTION

The **/etc/gettydefs** file contains information used by **getty** to set up the speed and terminal settings for a line (see *getty*(1M)). It supplies information on what the **login** prompt should look like. It also supplies the speed to try next if the user indicates the current speed is not correct by typing a **Break** character.

Each entry in **/etc/gettydefs** has the following format:

label# *initial-flags* # *final-flags* # *login-prompt* #*next-label*

Each entry is followed by a blank line. The various fields can contain quoted characters of the form **\b**, **\n**, **\c**, etc., as well as *****nnn*, where *nnn* is the octal value of the desired character. The various fields are:

label — This is the string against which **getty** tries to match its second argument. It is often the speed, such as **1200**, at which the terminal is supposed to run, but it need not be (see below).

initial-flags — These flags are the initial **ioctl()** settings to which the terminal is to be set if a terminal type is not specified to **getty** (see *ioctl*(2)). The flags that **getty** understands are the same as the ones listed in **/usr/include/sys/termio.h** (see *termio*(7)). Normally only the speed flag is required in the *initial-flags*. **getty** automatically sets the terminal to raw input mode and takes care of most of the other flags. The *initial-flag* settings remain in effect until **getty** executes **login**.

final-flags — These flags take the same values as the *initial-flags* and are set just before **getty** executes **login**. The speed flag is again required. The composite flag **SANE** takes care of most of the other flags that need to be set so that the processor and terminal are communicating in a rational fashion. The other two commonly specified *final-flags* are **TAB3**, so that tabs are sent to the terminal as spaces, and **HUPCL**, so that the line is hung up on the final close.

login-prompt — This entire field is printed as the *login-prompt*. Unlike the above fields where white space is ignored (a space, tab or new-line), they are included in the *login-prompt* field.

next-label — If this entry does not specify the desired speed, indicated by the user typing a **Break** character, **getty** searches for the entry with *next-label* as its *label* field and set up the terminal for those settings. Usually, a series of speeds are linked together in this fashion, into a closed set. For example, **2400** linked to **1200**, which in turn is linked to **300**, which finally is linked to **2400**.

If **getty** is called without a second argument, the first entry of **/etc/gettydefs** is used, thus making the first entry of **/etc/gettydefs** the default entry. It is also used if **getty** cannot find the specified *label*. If **/etc/gettydefs** itself is missing, there is one entry built into the command which brings up a terminal at **300** baud.

It is strongly recommended that after making or modifying **/etc/gettydefs**, it be run through **getty** with the check option to ensure that there are no errors.

EXAMPLES

The following two lines show an example of 300/1200 baud toggle, which is useful for dial-up ports:

```
1200# B1200 HUPCL # B1200 SANE IXANY IXANY TAB3 #login: #300
300# B300 HUPCL # B300 SANE IXANY IXANY TAB3 #login: #1200
```

The following line shows a typical 9600 baud entry for a hard-wired connection:

```
9600# B9600 # B9600 SANE IXANY IXANY ECHOE TAB3 #login: #9600
```

FILES

/etc/gettydefs

SEE ALSO

getty(1M), login(1), ioctl(2), termio(7).

NAME

group, logingroup - group file, grp.h

DESCRIPTION

group contains for each group the following information:

- group name
- encrypted password
- numerical group ID
- comma-separated list of all users allowed in the group

This is an ASCII file. Fields are separated by colons, and each group is separated from the next by a newline. No spaces should separate the fields or parts of fields on any line. If the password field is null, no password is associated with the group.

There are two files of this form in the system, **/etc/group** and **/etc/logingroup**. The file **/etc/group** exists to supply names for each group, and to support changing groups by means of the **newgrp** utility (see *newgrp*(1)). **/etc/logingroup** provides a default group access list for each user via **login** and **initgroups()** (see *login*(1) and *initgroups*(3C)).

The real and effective group ID set up by **login** for each user is defined in **/etc/passwd** (see *passwd*(4)). If **/etc/logingroup** is empty or non-existent, the default group access list is empty. If **/etc/logingroup** and **/etc/group** are links to the same file, the default access list includes the entire set of groups associated with the user. The group name and password fields in **/etc/logingroup** are never used; they are included only to give the two files a uniform format, allowing them to be linked together.

All group IDs used in **/etc/logingroup** or **/etc/passwd** should be defined in **/etc/group**. No user should be associated with more than **NGROUPS** (see *setgroups*(2)) groups in **/etc/logingroup**.

These files reside in directory **/etc**. Because of the encrypted passwords, these files can and do have general read permission and can be used, for example, to map numerical group IDs to names.

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 */
```

NETWORKING FEATURES

NIS

The **/etc/group** file can contain a line beginning with a plus (**+**), which means to incorporate entries from Network Information Services (NIS). There are two styles of **+** entries: **+** means to insert the entire contents of NIS group file at that point, and **+***name* means to insert the entry (if any) for *name* from NIS at that point. If a **+** entry has a non-null password or group member field, the contents of that field overide what is contained in NIS. The numerical group ID field cannot be overridden.

A group file can also have a line beginning with a minus (**-**), these entries are used to disallow group entries. There is only one style of **-** entry; an entry that consists of **-***name* means to disallow any subsequent entry (if any) for *name*. These entries are disallowed regardless of whether the subsequent entry comes from the NIS or the local group file.

WARNINGS

Group files must not contain any blank lines. Blank lines can cause unpredictable behavior in system administration software that uses these files.

Group ID (*gid*) 9 is reserved for the Pascal Language operating system and the BASIC Language operating system. These are operating systems for Series 300/400 computers that can co-exist with HP-UX on the same disk. Using this gid for other purposes can inhibit file transfer and sharing.

The length of each line in **/etc/group** is limited to **LINE_MAX**, as defined in <**limits.h**>. Because of this limit, users should not be listed in their primary group - only in their additional groups.

If **/etc/group** is linked to **/etc/logingroup**, group membership for a user is managed by NIS, and no NIS server is able to respond, that user cannot log in until a server does respond.

There is no single tool available to completely ensure that **/etc/passwd**, **/etc/group**, and **/etc/logingroup** are compatible. However, **pwck** and **grpck** can be used to simplify the task (see *pwck*(1M)).

There is no tool for setting group passwords in **/etc/group**.

DEPENDENCIES

NIS

EXAMPLES

Here is a sample **/etc/group** file:

```
other:*:1:root,daemon,uucp,who,date,sync
-oldproj
bin:*:2:root,bin,daemon,lp
+myproject:::bill,steve
+:
```


Group **other** has a gid of 1 and members **root**, **daemon**, **uucp**, **who**, **date**, and **sync**. The group **oldproj** is ignored since it appears after the entry **-oldproj**. Also, the group **myproject** has members **bill** and **steve**, and the password and group ID of the NIS entry for the group **myproject**. All groups listed in the NIS are pulled in and placed after the entry for **myproject**.

WARNINGS

The plus (**+**) and minus (**-**) features are part of NIS. Therefore if NIS is not installed, these features cannot work.

FILES

/etc/group
/etc/logingroup

SEE ALSO

groups(1), newgrp(1), passwd(1), setgroups(2), crypt(3C), getgrent(3C), initgroups(3C), passwd(4).

STANDARDS CONFORMANCE

group: SVID2, SVID3, XPG2

NAME

hosts - host name data base

DESCRIPTION

The file **/etc/hosts** associates Internet (IP) addresses with official host names and aliases. This allows a user to refer to a host by a symbolic name instead of an Internet address.

Note: This file *must* contain all addresses for local interfaces that **ifconfig** needs at boot time (see *ifconfig*(1M)). When using the name server (see *named*(1M)), or Network Information Service (see *ypserv*(1M)), this file often serves as a backup when the server is not running. In such circumstances, it is a common practice for **/etc/hosts** to contain a few addresses of machines on the local network.

/etc/hosts should contain a single line for each host with the following information:

<*internet address*> <*official host name*> <*aliases*>

Aliases are other names by which a host is known. They can substitute for the official host name in most commands. For example:

```
192.45.36.5       hpdxsg   testhost
```

In this example, users can use remote login on hpdxsg by using the command:

```
rlogin testhost
```

instead of

```
rlogin hpdxsg
```

If your system is in a domain naming environment, an official host name consists of the full domain extended host name. For example:

```
192.45.36.5       hpdxsg.xsg.hp.com   hpdxsg   testhost
```

A line cannot start with a blank (space or tab character). Items are separated by any number or combination of space or tab characters (blanks). A **#** character indicates the beginning of a comment. Characters from the **#** to the end of the line are not interpreted by routines that search the file. Trailing blanks are allowed at the end of a line.

For the Internet, this file is normally created from the official host database maintained at the Network Information Control Center (NIC), although local changes may be required to bring it up to date with respect to unofficial aliases and/or unknown hosts.

Network addresses are specified in the conventional Internet dot notation using the **inet_addr()** routine from the Internet address manipulation library (see *inet*(3N)). Host names can contain any printable character other than a white space, newline, or comment character.

EXAMPLES

See **/etc/hosts**.

AUTHOR

hosts was developed by the University of California, Berkeley.

SEE ALSO

gethostent(3N), inet(3N), nsswitch.conf(4).

NAME

hosts.equiv, .rhosts - security files authorizing access by remote hosts and users on local host

DESCRIPTION

The **/etc/hosts.equiv** file and files named **.rhosts** found in users' home directories specify remote hosts and users that are "equivalent" to the local host or user. Users from equivalent remote hosts are permitted to access a local account using **rcp** or **remsh** or to **rlogin** to the local account without supplying a password (see *rcp*(1), *remsh*(1), and *rlogin*(1)). The security provided by **hosts.equiv** is implemented by the **ruserok()** library routine, (see *rcmd*(3N)).

In this description, *hostequiv* means either the system **/etc/hosts.equiv** file or the user **.rhosts** file. Note that **.rhosts** must be owned by the user in whose home directory it is found and it must not be a symbolic link. The **/etc/hosts.equiv** file defines system-wide equivalency, whereas a user's **.rhosts** file defines equivalency between the local user and any remote users to whom the local user chooses to allow or deny access.

An entry in the *hostequiv* file is a single line (no continuations) in the format:

[*hostname* [*username*]] [**#***comment*]

Thus, it can be:

- A blank line.
- A comment line, beginning with a **#**.
- A host name, optionally followed by a comment.
- A host name and user name, optionally followed by a comment.

A host or user name is a string of printable characters, excluding whitespace, newlines, and **#**.

Names are separated by whitespace.

For a user to be granted access, both the remote host name and the user name must "match" an entry in *hostequiv*. When a request is made for access, the **/etc/hosts.equiv** file is searched first. If a match is found, access is permitted. If no match is found, the **.rhosts** file is searched, if one exists in the local user's home directory. If the local user is a superuser, **/etc/hosts.equiv** is ignored.

A host name or user name must match the corresponding field entry in *hostequiv* in one of the following ways:

Literal match
: A host name in *hostequiv* can literally match the official host name (not an alias) of the remote host.

  A user name in *hostequiv* can literally match the remote user name. For a user name to have literal match in the **/etc/hosts.equiv** file, the remote user name must literally match the local user name.

Domain-extended match
: The remote host name to be compared with entries in *hostequiv* is typically the official host name returned by **gethostbyaddr()** (see *gethostent*(3N)). In a domain-naming environment, this is a domain-qualified name. If a host name in *hostequiv* does not literally match the remote host name, the host name in *hostequiv* with the local domain name appended may match the remote host name.

-*name*
: If the host name in *hostequiv* is of this form, and if *name* literally matches the remote host name or if *name* with the local domain name appended matches the remote host name, access is denied regardless of the user name.

  If the user name in *hostequiv* is of this form, and *name* literally matches the remote user name, access is denied.

  Even if access is denied in this way by **/etc/hosts.equiv**, access can still be allowed by **.rhosts**.

+
: Any remote host name matches the host name **+** in *hostequiv*.

  Any remote user matches the user name **+**.

+@*netgroup_name*
: *netgroup_name* is the name of a network group as defined in *netgroup*(4). If the host name in *hostequiv* is of this form, the remote host name (only)

must match the specified network group according to the rules defined in *netgroup*(4) in order for the host name to match.

Similarly, if the user name in *hostequiv* is of this form, the remote user name (only) must match the specified network group in order for the user name to match.

-@*netgroup_name* — *netgroup_name* is the name of a network group as defined in *netgroup*(4). If the host name in *hostequiv* is of this form, and if the remote host name (only) matches the specified network group according to the rules defined in *netgroup*(4), access is denied.

Similarly, if the user name in *hostequiv* is of this form, and if the remote user name (only) matches the specified network group, access is denied.

Even if access is denied in this way by **/etc/hosts.equiv**, access can still be allowed by **.rhosts**.

EXAMPLES

1. **/etc/hosts.equiv** on **hostA** contains the line:

   hostB

   and **/etc/hosts.equiv** on **hostB** is empty. User **chm** on **hostB** can use **remsh** to **hostA**, or **rlogin** to account **chm** on **hostA** without being prompted for a password. **chm** will, however, be prompted for a password with **rlogin**, or denied access with **remsh**, from **hostA** to **hostB**.

   If **.rhosts** in the home directory of user **chm** on **hostB** contains:

   hostA

   or

   hostA chm

   then user **chm** can access **hostB** from **hostA**.

2. **hostA** is in the domain **arg.bob.com**. **hostB** and **hostC** are in the domain **oink.bob.com**. **.rhosts** in the home directory of user **chm** on **hostB** contains:

   hostC
   hostA

   User **chm** can access **hostB** from **hostC**, since **hostC.oink.bob.com** matches **hostC** with **hostB**'s local domain **oink.bob.com** appended. But user **chm** from **hostA** cannot access **hostB**, since **hostA.arg.bob.com** does not match **hostA.oink.bob.com**. In order for user **chm** to be able to access **hostB** from **hostA**, **chm**'s **.rhosts** file on **hostB** must contain:

   hostA.arg.bob.com

   since **hostA** is in a different domain.

3. **.rhosts** in the home directory of user **chm** on **hostA** contains:

   hostB root

   /etc/hosts.equiv on **hostB** contains the line:

   hostA

   However, there is no file **.rhosts** in the home directory of user **chm** on **hostB**. The user **root** on **hostB** can **rlogin** to account **chm** on **hostA** without being prompted for a password, but **root** on **hostA** cannot **rlogin** to account **chm** on **hostB**.

4. **.rhosts** in the home directory of user **chm** on **hostA** contains:

   +
   -hostB
   + root

   User **chm** from any host is allowed to access account **chm** on **hostA**. User **root** from any host except **hostB** can access account **chm** on **hostA**.

5. **/etc/hosts.equiv** on **hostA** contains the lines:

h

```
+ -chm
hostB
```

Any user from **hostB** except **chm** is allowed to access an account on **hostA** with the same user name. However, if **.rhosts** in the home directory of user **chm** on **hostA** contains:

```
hostB
```

then user **chm** from **hostB** can access account **chm** on **hostA**.

6. **/etc/hosts.equiv** on **hostA** contains the line:

```
+@example_group
```

The network group **example_group** consists of:

```
example_group ( , ,EXAMPLE_DOMAIN)
```

If **hostA** is not running Network Information Service (NIS), user **chm** on any host can access account **chm** on **hostA**.

If **hostA** is running Network Information Service (NIS), and **hostA** is in the domain **EXAMPLE_DOMAIN**, user **chm** on any host, whether in **EXAMPLE_DOMAIN** or not, can access account **chm** on **hostA**.

h

However, if **.rhosts** in the home directory of user **chm** on **hostA** contains the line:

```
-@example_group
```

and **hostA** is either not running Network Information Service (NIS) or is in domain **EXAMPLE_DOMAIN**, no user **chm** on any host can access the account **chm** on **hostA**. If **hostA** is running Network Information Service (NIS) but is not in the domain **EXAMPLE_DOMAIN**, this line has no effect.

7. **/etc/hosts.equiv** on **hostA** contains the line:

```
-@example_group
```

The network group **example_group** consists of:

```
example_group (hostB, ,)
```

All users on **hostB** are denied access to **hostA**.

However, if **.rhosts** in the home directory of a user on **hostA** contains any of the following lines:

```
+@example_group chm
hostB chm
+ chm
```

then user **chm** on **hostB** can access that account on **hostA**.

WARNINGS

For security purposes, the files **/etc/hosts.equiv** and **.rhosts** should exist and be readable and writable only by the owner, even if they are empty.

Care must be exercised when creating the **/etc/hosts.equiv**

The **-l** option to **remshd** and **rlogind** prevents any authentication based on **.rhosts** files for users other than a superuser.

AUTHOR

hosts.equiv was developed by the University of California, Berkeley.

The **+**, **-***name*, **+@***netgroup_name*, and **-@***netgroup_name*, extensions were developed by Sun Microsystems, Inc.

FILES

$HOME/.rhosts
/etc/hosts.equiv

SEE ALSO

rcp(1), rdist(1), remsh(1), rlogin(1), remshd(1M), rlogind(1M), gethostent(3N), rcmd(3N), netgroup(4).

NAME

inetd.conf - configuration file for inetd

DESCRIPTION

On invocation, the **inetd** daemon reads its configuration information from the **/etc/inetd.conf** configuration file, and possibly at some later time in response to a **SIGHUP** signal (see *inetd*(1M)).

Each line in the file is treated either as a comment or as configuration information for a given service. Comments are denoted by a **#** at the beginning of a line. Noncomment lines contain seven or nine required fields, depending on the service name specified in the first field. Fields are separated by tabs and/or spaces. A line can be continued if it terminates with a ****. Each configuration line in the file contains the following fields in the order indicated:

- *service name*
- *socket type*
- *protocol*
- **wait**|**nowait**
- *user*
- *server program*
- *program number* (NFS RPC services only)
- *version number* (NFS RPC services only)
- *server program arguments*

Fields are constructed as follows:

service name — **rpc** if the server is RPC-based (NFS); otherwise, the name of a valid service in file **/etc/services**. For example, **shell** for the **remsh** service (see *remsh*(1)), **login** for the **rlogin** service (see *rlogin*(1)), and **telnet** for the **telnet** service (see *telnet*(1)).

socket type — **stream**, **dgram**, or **xti**, depending on whether the server socket is a stream or a datagram socket, or intended for a program built using the XTI API.

protocol — Must be a valid protocol as given in **/etc/protocols**; for example, **tcp** or **udp**. If XTI is specified in the socket type field, a full pathname to a device may be specified here, such as **/dev/tcp**; otherwise, the protocol specified here will be appended to **/dev/**. For example, if **tcp** is specified for an XTI application, the path **/dev/tcp** will be used.

wait|**nowait** — Specifies whether **inetd** should act as a single- or multi-threaded server.

- **wait** — Instructs **inetd** to start one server to handle an incoming request, and cease listening for new requests for the same service until the server started exits.
- **nowait** — Instructs **inetd** to start one server for each incoming request.

Most UDP-based services use **wait** for this field, while TCP-based services use **nowait**.

user — User ID to be used when the server is running.

server program — Absolute path name of the program executed by **inetd** when it finds a request on the server's socket.

server program arguments — Arguments to the server program. The same as in normal use, starting with **argv[0]**, which is the name of the program.

If *service name* is **rpc** (NFS RPC services), two extra fields are required. They must appear between the *server program* field and the *server program arguments* field:

program number — Defines a particular service grouping and is unique.

version number — Version supported by the RPC service. This number can be a single value, or a range, if the program handles multiple versions; for example, **1** or **1-3**. Ranges

i

are separated by a hyphen (**-**). Version numbers allow RPC protocols to be extended and modified, and make it possible for old and new protocols to share the same server process.

Built-in inetd Services

The **inetd** daemon provides several "trivial" services internally by use of built-in routines (see *inetd*(1M) for a list of these services). To configure an internal service, specify **internal** as the *server program* name, and omit the *server program arguments* field.

EXAMPLES

Configure the **shell** service to use TCP protocol, and run the server **remshd** as user **root**.

```
shell stream tcp nowait root /usr/lbin/remshd remshd
```

Configure the FTP server to timeout an inactive session after 75 seconds.

```
ftp stream tcp nowait root /usr/lbin/ftpd ftpd -t75
```

Configure an RPC-based service. Note that the *service name* field contains **rpc** and two more fields are used: the program number (100008) and version number (1).

```
rpc dgram udp wait root /usr/lib/netsvc/rwall/rpc.rwalld 100008 1
rpc.rwalld
```

i

Configure **inetd** to use the built-in **daytime** TCP service.

```
daytime stream tcp nowait root internal
```

AUTHOR

inetd.conf was developed by the University of California, Berkeley.

NFS was developed by Sun Microsystems, Inc.

SEE ALSO

inetd(1M), exec(2), fork(2), inetd.sec(4), protocols(4), services(4).

NAME

inetd.sec - optional security file for inetd

DESCRIPTION

When **inetd** accepts a connection from a remote system, it checks the address of the host requesting the service against the list of hosts to be allowed or denied access to the specific service (see *inetd*(1M)). The file **inetd.sec** allows the system administrator to control which hosts (or networks in general) are allowed to use the system remotely. This file constitutes an extra layer of security in addition to the normal checks done by the services. It precedes the security of the servers; that is, a server is not started by the Internet daemon unless the host requesting the service is a valid host according to **inetd.sec**.

If file **/var/adm/inetd.sec** does not exist, security is limited to that implemented by the servers. **inetd.sec** and the directory **/var/adm** should be writable only by their owners. Changes to **inetd.sec** apply to any subsequent connections.

Lines in **inetd.sec** beginning with **#** are comments. Comments are not allowed at the end of a line of data.

The lines in the file contain a service name, permission field, and the Internet addresses or official names of the hosts and networks allowed to use that service in the local host. The fields in each line are as follows:

<*service name*> <**allow**|**deny**> <*host/net addresses, host/net names*>

service name is the name (not alias) of a valid service in file **/etc/services**. The service name for RPC-based services (NFS) is the name (not alias) of a valid service in file **/etc/rpc**. A service name in **/etc/rpc** corresponds to a unique RPC program number.

allow|**deny** determines whether the list of remote hosts in the next field is allowed or denied access to the specified service. Multiple **allow**|**deny** lines for each service are not unsupported. If there are multiple **allow**|**deny** lines for a particular service, all but the last line are ignored.

Addresses and names are separated by white space. Any mix of addresses and names is allowed. To continue a line, terminate it with ****.

Host names and network names are the official names of the hosts or networks as returned by **gethostbyaddr()** or **getnetbynumber()**, respectively. Wildcard characters (*****) and range characters (**-**) are allowed. The ***** and the **-** can be present in any of the fields of the address. An address field is a string of characters separated by a dot (**.**).

EXAMPLES

Use a wildcard character to permit a whole network to communicate with the local host without having to list all the hosts in that network. For example, to allow all hosts with network addresses starting with a **10**, as well as the single host with address **192.54.24.5** to use *rlogin*:

```
login       allow   10.*  192.54.24.5
```

On a system running NFS, deny host **192.54.24.5** access to *sprayd*, an RPC-based server:

```
sprayd      deny    192.54.24.5
```

A **range** is a field containing a **-** character. To deny hosts in network 10 (arpa) with subnets 3 through 5 access to **remsh**:

```
shell       deny    10.3-5.*
```

The following entry denies **rlogin** access to host **cory.berkeley.edu**, any hosts on the network named **testlan**, and the host with internet address **192.54.24.5**:

```
login       deny    192.54.24.5 cory.berkeley.edu testlan
```

If a remote service is not listed in the security file, or if it is listed but it is not followed by **allow** or **deny**, all remote hosts can attempt to use it. Security is then provided by the service itself. The following lines, if present in **inetd.sec**, allow or deny access to the service indicated:

Allow all hosts to use **ftp**:

```
ftp
```

Deny all access to the **shell** service; i.e., **remsh**:

```
shell   deny
```

Allow access to the **shell** service by any host:

```
shell   allow
```

or

```
shell
```

AUTHOR

inetd.sec was developed by HP.

NFS was developed by Sun Microsystems, Inc.

FILES

/var/adm/inetd.sec

SEE ALSO

inetd(1M), gethostent(3N), getnetent(3N), hosts(4), inetd.conf(4), networks(4), protocols(4), rpc(4), services(4).

i

NAME

inetsvcs.conf - configuration file for secure internet services

DESCRIPTION

The internet services, ftp, rcp, remsh, rlogin and telnet, use the /etc/inetsvcs.conf configuration file to decide their behavior (i.e., whether to allow network authentication using Kerberos V5 or not). The contents of the file decide whether the secure internet services are to be enabled or not. This configuration file is updated by the program **inetsvcs_sec**. The default entry in the file is as follows:

```
kerberos false
```

With this entry, all the specified services show their traditional behavior (i.e., provide authentication by prompting for the user's password).

To enable secure internet services, the inetsvcs_sec program is used to update the configuration file with the following entry:

```
kerberos true
```

WARNING

This file should not be updated manually. The services will not run if an invalid entry exists in this file. The program **inetsvcs_sec** must be used to update the configuration file.

SEE ALSO

sis(5), inetsvcs_sec(1M).

NAME

info - diskless client configuration information file

DESCRIPTION

The **info** file is a POSIX shell sourceable file which contains parameter definitions used at boot time. Typically, it will be an empty file and default values will be used for all parameters. Following is the list of parameters which can be defined in the **info** file:

ROOT_SERVER_IP Specifies the IP address of the client's private root server. If this is not specified, the client's private root server defaults to the boot server.

PRIVATE_ROOT Specifies the pathname to the client's private root on the private root server. If this is not specified, the client's private root path defaults to **/export/private_roots/***client_name*.

MOUNT_ROOT_OPTS Specifies the NFS mount options to mount the client's private root from the private root server. If this is not specified, the mount options default to **boot,hard,nointr,nodevs**.

MOUNT_STAND_OPTS Specifies the NFS mount options to mount the client's **/stand** directory from the boot server. If this is not specified, the mount options default to **boot,hard,nointr,nodevs**.

i

NO_SWAP_TO_NFS Specifies whether NFS should be configured as primary swap. (NOTE: In order to swap to NFS, a diskless kernel must be configured with tunable parameter **remote_nfs_swap** set to 1.) If a diskless machine has a local swap disk and swap to NFS is not desired, the **NO_SWAP_TO_NFS** parameter should be set to the value of 1 and the diskless kernel should be configured without setting **remote_nfs_swap** to 1. If this parameter is not specified in the **info** file and the kernel tunable parameter **remote_nfs_swap** is set to 1, then NFS will be configured as primary swap.

REMOVE_EXTRA_SWAPFILES
If not set, this parameter defaults to a value of 1, and results in the removal of all swapfiles above the configured swap minimum (swap **min** is specified in the client's **/etc/fstab**) when a diskless client boots. This ensures that extraneous swapfiles at boot time are removed, thus freeing disk space. If **REMOVE_EXTRA_SWAPFILES** is set to 0 in the **info** file, removal of extra swapfiles is disabled. This may result in faster boot times due to the time savings in creating additional swap files.

The **info** file resides in the same directory as the client's kernel (**/export/tftpboot/***client***/stand**) on the boot server and is retrieved at boot time using **tftp** command. By default, when a diskless client is created, an empty **info** file is placed in the client's kernel directory. This ensures that all parameters revert to their default values (see above). If the file is not present, this is an error.

EXAMPLES

An example **info** file is shown below:

```
# Sample info(4) file:
# set NO_SWAP_TO_NFS
NO_SWAP_TO_NFS=1
```

FILES

/export/tftpboot/*client***/stand/info**

NAME

inittab - script for the boot **init** process

DESCRIPTION

The **/etc/inittab** file supplies the script to the boot **init** daemon in its role as a general process dispatcher (see *init*(1M)). The process that constitutes the majority of boot **init**'s process dispatching activities is the line process **/usr/sbin/getty** that initiates individual terminal lines. Other processes typically dispatched by boot **init** are daemons and shells.

The **inittab** file is composed of entries that are position-dependent and have the following format:

id **:** *rstate* **:** *action* **:** *process*

Each entry is delimited by a newline; however, a backslash (****) preceding a newline indicates a continuation of the entry. Up to 1024 characters per entry are permitted. Comments can be inserted in the *process* field by starting a "word" with a **#** (see *sh*(1)). Comments for lines that spawn **getty**s are displayed by the **who** command (see *who*(1)). It is expected that they will contain some information about the line such as the location. There are no limits (other than maximum entry size) imposed on the number of entries within the **inittab** file.

The entry fields are:

id A one- to four-character value used to uniquely identify an entry. Duplicate entries cause an error message to be issued, but are otherwise ignored. The use of a four-character value to identify an entry is strongly recommended (see WARNINGS below).

rstate Defines the *run level* in which this entry is to be processed. Run levels correspond to a configuration of processes in the system where each process spawned by boot **init** is assigned one or more run levels in which it is allowed to exist. Run levels are represented by a number in the range **0** through **6**. For example, if the system is in run level **1**, only those entries having a **1** in their *rstate* field are processed.

When boot **init** is requested to change run levels, all processes that do not have an entry in the *rstate* field for the target run level are sent the warning signal (**SIGTERM**) and allowed a 20-second grace period before being forcibly terminated by a kill signal (**SIGKILL**). You can specify multiple run levels for a process by entering more than one run level value in any combination. If no run level is specified, the process is assumed to be valid for all run levels, **0** through **6**.

Three other values, **a**, **b** and **c**, can also appear in the *rstate* field, even though they are not true run levels. Entries having these characters in the *rstate* field are processed only when a user **init** process requests them to be run (regardless of the current system run level). They differ from run levels in that boot **init** can never enter "run level" **a**, **b**, or **c**. Also, a request for the execution of any of these processes does not change the current numeric run level.

Furthermore, a process started by an **a**, **b**, or **c** option is not killed when boot **init** changes levels. A process is killed only if its line in **inittab** is marked **off** in the *action* field, its line is deleted entirely from **inittab**, or boot **init** goes into the *single-user* state.

action A keyword in this field tells boot **init** how to treat the process specified in the *process* field. The following actions can be specified:

boot Process the entry only at boot **init**'s boot-time read of the **inittab** file. Boot **init** starts the process, does not wait for its termination, and when it dies, does not restart the process. In order for this instruction to be meaningful, the *rstate* should be the default or it must match boot **init**'s run level at boot time. This action is useful for an initialization function following a hardware boot of the system.

bootwait Process the entry only at boot **init**'s boot-time read of the **inittab** file. Boot **init** starts the process, waits for its termination, and, when it dies, does not restart the process.

initdefault An entry with this *action* is only scanned when boot **init** is initially invoked. Boot **init** uses this entry, if it exists, to determine which run level to enter initially. It does this by taking the highest run level specified in the **rstate** field and using that as its initial state. If the

i

rstate field is empty, boot **init** enters run level **6**.

The **initdefault** entry cannot specify that boot **init** start in the single-user state. Additionally, if boot **init** does not find an **initdefault** entry in **inittab**, it requests an initial run level from the user at boot time.

off If the process associated with this entry is currently running, send the warning signal (**SIGTERM**) and wait 20 seconds before forcibly terminating the process via the kill signal (**SIGKILL**). If the process is nonexistent, ignore the entry.

once When boot **init** enters a run level that matches the entry's *rstate*, start the process and do not wait for its termination. When it dies, do not restart the process. If boot **init** enters a new run level but the process is still running from a previous run level change, the process is not restarted.

ondemand This instruction is really a synonym for the **respawn** action. It is functionally identical to **respawn** but is given a different keyword in order to divorce its association with run levels. This is used only with the **a**, **b**, or **c** values described in the *rstate* field.

powerfail Execute the process associated with this entry only when boot **init** receives a power-fail signal (**SIGPWR** see *signal*(5)).

powerwait Execute the process associated with this entry only when boot **init** receives a power-fail signal (**SIGPWR**) and wait until it terminates before continuing any processing of **inittab**.

respawn If the process does not exist, start the process; do not wait for its termination (continue scanning the **inittab** file). When it dies, restart the process. If the process currently exists, do nothing and continue scanning the **inittab** file.

sysinit Entries of this type are executed before boot **init** tries to access the console. It is expected that this entry will be only used to initialize devices on which boot **init** might attempt to obtain run level information. These entries are executed and waited for before continuing.

wait When boot **init** enters the run level that matches the entry's *rstate*, start the process and wait for its termination. Any subsequent reads of the **inittab** file while boot **init** is in the same run level cause boot **init** to ignore this entry.

process This is a **sh** command to be executed. The entire **process** field is prefixed with **exec** and passed to a forked **sh** as "**sh -c 'exec** *command***'**". For this reason, any **sh** syntax that can legally follow **exec** can appear in the *process* field. Comments can be inserted by using the **; #***comment* syntax.

WARNINGS

The use of a four-character *id* is strongly recommended. Many pty servers use the last two characters of the pty name as an *id*. If an *id* chosen by a pty server collides with one used in the **inittab** file, the **/etc/utmp** file can become corrupted. A corrupt **/etc/utmp** file can cause commands such as **who** to report inaccurate information.

FILES

/etc/inittab File of processes dispatched by boot **init**.

SEE ALSO

sh(1), getty(1M), exec(2), open(2), signal(5).

i

NAME

inode - format of an inode

SYNOPSIS

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

DESCRIPTION

An inode for a plain file or directory in a file system has the following structure as it appears on a disk block, defined in **<sys/ino.h>**.

```
struct      dinode {
    u_short di_mode;          /* mode and type of file */
    short   di_nlink;         /* number of links to file */
    short   di_uid_lsb;       /* owner's user id */
    short   di_gid_lsb;       /* owner's group id */
    quad    di_size;          /* number of bytes in file */
    time_t  di_atime;         /* time last accessed */
    long    di_atspare;
    time_t  di_mtime;         /* time last modified */
    long    di_mtspare;
    time_t  di_ctime;         /* time of last file status change */
    long    di_ctspare;
    daddr_t di_db[NDADDR];    /* disk block addresses */
    daddr_t di_ib[NIADDR];    /* indirect blocks */
    long    di_flags;         /* status */
    long    di_blocks;        /* blocks actually held */
    long    di_gen;           /* file generation number */
    short   di_uid_msb;       /* owner's user id (top 16 bits) */
    short   di_gid_msb;       /* owner's group id (top 16 bits) */
    long    di_spare[2];      /* reserved, currently unused */
    ino_t   di_contin;        /* continuation inode number */
};
```

A continuation inode contains a file's optional access control list (ACL) entries, and has the following structure as it appears on a disk block:

```
struct       cinode {
    u_short ci_mode;          /* mode and type of file */
    short   ci_nlink;         /* number of links to file */
                              /* optional ACL entries */
    struct  acl_entry_internal_lsb ci_acl_lsb[NOPTENTRIES];
                              /* least sig bytes of ACL uids/gids */
    struct  acl_entry_internal_msb ci_acl_msb[NOPTENTRIES];
                              /* most sig bytes of ACL uids/gids */
    char    ci_spare[3];      /* reserved, currently unused */
    long    ci_ciflags;       /* status */
};
```

For the meaning of the defined types **u_short**, **quad**, **daddr_t**, and **time_t**, see *types*(5).

Continuation inodes are distinguished from other inodes by their file type. See **<sys/inode.h>** for the definition of these values.

See **<sys/inode.h>** for the definition of inode structures for special files, pipes, or FIFOs.

WARNINGS

Kernel internal structures may change from release to release without warning. Applications directly relying on these structures are not supported.

AUTHOR

AT&T, the University of California, Berkeley, and HP.

SEE ALSO

stat(2), fs(4), types(5).

i

NAME

inode (vxfs) - format of a VxFS inode

SYNOPSIS

```
#include <sys/param.h>
#include <sys/kern_sem.h>
#include <sys/fs/vx_hpux.h>
#include <sys/fs/vx_port.h>
#include <sys/fs/vx_inode.h>
```

DESCRIPTION

The inode list consists of *fs_inopau* inode entries in each allocation unit.

Although a **vxfs** inode is typically 256 bytes in length, an inode size of 512 bytes can be used instead.

An inode entry has the following format:

i_mode
The mode and type of file.

i_nlink
The number of links to the file.

i_uid
The inode owner.

i_gid
The inode group.

i_size
The size in bytes of the file. Eight bytes have been allocated.

i_atime
Time of last access, in **struct timeval** format.

i_mtime
Time of last modification, in **struct timeval** format.

i_ctime
Time of last inode change, in **struct timeval** format.

i_aflags
These flags are used to control the allocation and extension of files.

VX_AF_IFBAD
If this flag is set, the inode is invalid in some way. It should be cleared when *fsck*(1M) is run.

VX_AF_NOEXTEND
If this flag is set, the file may not be extended once the current reservation is exceeded. The reservation may be increased by the **VX_SETEXT** ioctl, but the file will not be automatically extended.

VX_AF_NOGROW
If this flag is set, the file may not be extended once the current reservation is exceeded. It should be cleared on truncation or when *setext*(1M) is run. This flag is usually set because an I/O error occurs while extending a file.

VX_AF_ALIGN
If this flag is set, the file must be allocated in extents of a fixed size and alignment. If an extent of *i_fixextsize* blocks aligned on an *i_fixextsize* boundary can't be found, then the allocation will fail. The alignment is relative to the beginning of the allocation unit.

i_orgtype
Mapping type. Indicates how the inode mapping area is to be interpreted. Currently there are three mapping types supported:

IORG_EXT4
Mapping area consists of an array of 32-bit extent block addresses and sizes.

i

IORG_IMMED
Mapping area itself is a data block. This mapping is referred to as Immediate Inode Data.

IORG_TYPED
Mapping area consists of type-extent structures.

i_eopflags
Extended inode operation flag area.

i_eopdata
Extended inode operation data area.

i_ftarea
This is a union. The contents are determined by file type.

For devices, the following fields are supported:

i_rdev
The device number of a block or character special device.

For directories, the following fields are supported:

i_dotdot
The parent directory inode inumber if the inode is a directory. This replaces the standard ".." entry in the first directory block. The **vxfs** file system does not have explicit "." and ".." entries.

For regular files, the following fields are supported:

i_reserve
The number of data blocks reserved for exclusive use by the file (preallocation). A preallocation may be requested using *ioctl*(2). See *vxfsio*(7).

i_fixextsize
Set when the inode has a fixed extent size. The default is to have a variable extent size allocation policy. A fixed extent size may be specified using *ioctl*(2). See *vxfsio*(7).

For structural files, the following fields are supported:

i_matchino
(Version 2 and 3 layouts only.) The inode number of the "matching" inode. For replicated files, this is the inode of the replica. For extent map reorganization files, this is the inode of the file being reorganized.

i_fsetindex
(Version 2 and 3 layouts only.) The index of the fileset associated with this inode.

i_blocks
The number of blocks currently allocated to the file, including any blocks allocated for indirect address extents.

i_gen
The generation number. A serial number which is incremented whenever the inode is freed and reallocated. It is designed to provide a "handle" for stateless servers such as NFS.

i_vversion
A count of the number of times the inode metadata has been modified. This field is a 64-bit number.

ic_org
The mapping area. This field is a union based on the value of **i_orgtype** and the file system type.

For the **vxfs IORG_IMMED** organization type, the following structure is used:

i_immed
The Immediate Inode data area, **NIMMED_N** (currently 96) bytes in length (see **fs_immedlen**). Any directory or symbolic link which is <= 96 bytes in length will be stored directly in the inode.

i

For the **vxfs IORG_EXT4** organization type, the following structure is used:

i_ies
Indirect extent size. This is the size in blocks of the indirect data extents in the file.

i_ie
Array of indirect address extents. There are **NIADDR** (currently 2) indirect address extents. The indirect address extents are 8192 bytes long. Each indirect address extent may contain up to 2048 extent addresses. The first indirect address extent is used for single indirection. With single indirection, each entry in the indirect address extent indicates the starting block number of a data extent. The second indirect address extent is a double indirect address extent. With double indirection, each entry in the indirect address extent indicates the starting block number of a single indirect address extent.

i_dext
An array of structures containing the direct extent addresses and sizes. Up to **NDADDR_N** (currently ten) direct extents are supported. Since a variable length extent allocation policy is used, each direct extent may have a different size. Each structure contains the following elements:

i_de Direct extent address.

i_des Direct extent size.

i

i_iattrino
(Version 2 and 3 layouts only.) Indirect attribute inode. This identifies the inode in the attribute fileset that contains indirect attribute references.

The remaining bytes of the inode are reserved for extended attribute records, which are formatted as follows:

length
The exact length of the attribute record. If this is not a multiple of 4 bytes, the start of the next attribute record is found by rounding the length up to a 4 byte boundary.

format
The format of the data layout of the remainder of the attribute record. Each attribute consists of a class identifying the attribute's administrative domain; a subclass identifying the attribute within the administrative domain; and data. The valid record formats are:

ATTR_EXTIMMED
This record extends the immediate data area so that files larger than 96 bytes can be stored directly in the inode.

ATTR_IMMED
The attribute is stored directly in the inode. The fields in the rest of this record are:

class
The class of the attribute.

subclass
The subclass of the attribute.

data
The attribute data.

ATTR_DIRECT
When attributes are too large to store directly in the inode, each attribute is stored in its own file. This record lists each attribute along with the inode number corresponding to the file in which the attribute is stored. The number of entries in the list is determined by the length of the record. The fields in each entry are:

class
The class of the attribute.

subclass
The subclass of the attribute.

length
The length of the attribute data. This allows attribute operations to check

the length of an attribute without reading the attribute inode.

inumber
The inode number of the file containing the attribute data. The inode is part of the attribute fileset.

The attribute records in the inode are terminated by a record with a format of zero (for compatibility with file systems that have the last 80 bytes of all inodes set to NULL).

SEE ALSO

fs_vxfs(4), stat(2), types(5).

NAME

ioconfig - ioconfig entry format

SYNOPSIS

#include <sys/ioconfig.h>

DESCRIPTION

The **ioconfig** file provides the mapping between information stored in device file **dev_t** (major number and logical unit) and the information the I/O system uses to communicate with devices (hardware paths and manager paths).

At boot time **ioinit** reads the file and stores the information in the **io_tree** kernel data structure (see *ioinit*(1M)). The **ioconfig** file is created by **insf** at install time and is modified by **insf** and **rmsf** when devices are added or removed (see *insf*(1M) and *rmsf*(1M)). The only purpose of the **ioconfig** file to maintain configuration information when the system is not running. While the system is running, all accesses are made directly to the kernel **io_tree** structure, although any tools that change the kernel structures must also keep **ioconfig** consistent.

The **ioconfig** file begins with the ioconfig magic number.

```
#define   IOCONFIG_MAGIC     0x2122494f     /* magic number */
```

Following the magic number is an array of **ioconfig_entry** structures, which logically form a tree structure defining the connectivity of the various levels of software modules and managers, the device class and hardware address of each element, and the logical unit associated with each leaf node. The root of the tree is array element 0.

Each **ioconfig_entry** contains the following fields as defined in <**sys/ioconfig.h**>:

```
#define IOCONFIG_FILE   "/etc/ioconfig"
#define MAX_ID          16
#define NONE            -1

typedef char         io_name_type[MAX_ID];

typedef              struct {
    int              parent;         /* parent in io_tree */
    int              sibling;        /* sibling in io_tree */
    int              child;          /* child in io_tree */
    io_name_type     manager;        /* manager name */
    io_name_type     module;         /* module name */
    io_name_type     class;          /* device class */
    int              lu;             /* logical unit number */
    int              hdw_address;    /* hardware address */

} ioconfig_entry;
```

The definitions of each element are as follows:

parent, **sibling**, **child**
: Each of the parent, sibling, and child fields is the array index of another **ioconfig** structure within the file. This allows the file to represent the tree structure of the kernel **io_tree** without using pointers. The value **NONE** indicates there is no node of the appropriate type.

hdw_address
: Hardware address of the entity. The value **NONE** indicates the node corresponds to a manager controlling a logical device (one without a hardware address).

manager
: A NULL-terminated character string representing the manager name.

module A NULL-terminated character string representing the module name.

class A NULL-terminated character string representing the device class.

lu Logical unit associated with this particular node by the user. Meaningful only for leaf nodes; has the value **NONE** for all others.

There can be multiple chains of nodes in **ioconfig** with the same manager names, module names, and/or hardware addresses, provided each manager name/lu pair uniquely identifies a single leaf node.

i

AUTHOR

ioconfig was developed by HP.

FILES

/etc/ioconfig

SEE ALSO

ioinit(1M), insf(1M), rmsf(1M), magic(4).

NAME

issue - issue identification file

DESCRIPTION

The file **/etc/issue** contains the *issue* or project identification to be printed as a login prompt. This is an ASCII file which is read by the **getty** program then written to any terminal spawned or respawned from the **inittab** file.

FILES

/etc/issue

SEE ALSO

getty(1M), login(1).

i

NAME

lif - logical interchange format description

DESCRIPTION

LIF (Logical Interchange Format) is a Hewlett-Packard standard mass-storage format that can be used for interchange of files among various HP computer systems. A LIF volume contains a header (identifying it as a LIF volume) and a directory that defines the contents (i.e. files) of the volume. The size of the directory is fixed when the volume is initialized (see *lifinit*(1)) and sets an upper bound on the number of files that can be created on the volume.

HP-UX contains a set of utilities (referred to as *lif*∗(1)) that can be used to:

- Initialize a LIF volume (i.e. create a header and an empty directory),
- Copy files to and from LIF volumes,
- List the contents of LIF volumes,
- Remove LIF files,
- Rename LIF files.

The *lif*∗(1) utilities are the only utilities within HP-UX where the internal structure of a LIF volume is known. To the rest of HP-UX, a LIF volume is simply a file containing some unspecified data. The term **LIF volume** should in no way be confused with the HP-UX notion of a file system volume or mountable volume.

LIF utilities on HP-UX currently support three file types, ASCII (1), BINARY (-2) and BIN (-23951).

Three copying modes are associated with these file types:

ASCII If the copying mode is ASCII and an HP-UX file is being copied to a LIF volume, the utility strips the trailing LF (line-feed) character, and inserts two bytes of record length in front of each record. These records are then written to a LIF-formatted medium. When copying a LIF ASCII file to HP-UX the two-byte record length is stripped and a trailing LF is appended. These records are then written to the destination. In this mode of copying, the length of the file is preserved. The default file type for this mode of copying is ASCII (1).

BINARY If the copying mode is **BINARY**, and an HP-UX file is being copied to a LIF volume, the utility simply inserts two bytes for record length in front of each 1-Kbyte record. A trailing fractional block has a count reflecting the number of bytes in that block. No interpretation is placed on the content of the records. These records are then written to a LIF-format medium. When copying a LIF file to an HP-UX file in **BINARY** copying mode, the record lengths are stripped and the content of records is directly written to the destination. In this mode of copying, the length of the binary file is preserved. The default file type for this mode of copying is **BINARY** (-2).

RAW If the copying mode is RAW, and an HP-UX file is being copied to a LIF volume, the utility simply copies the raw data to the destination. File sizes that are not integer multiples of 256 bytes are padded with nulls to the next higher multiple. Therefore, *file sizes are not preserved*. When copying a LIF file to an HP-UX file in RAW mode, the information is copied directly without any interpretation placed on the content of the source. The default file type for this mode of copying is **BIN** (-23951).

A LIF volume can be created on any HP-UX file (either regular disk file or device special file) that supports random access via **lseek()** (see *lseek*(2)). *Do not mount the special file before using lif*∗(1) *utilities*. See *lifinit*(1) for details. Within a LIF volume, individual files are identified by 1- to 10-character file names. File names can consist of uppercase alphanumeric characters (A through Z, 0 through 9) and the underscore character (_). The first character of a LIF file name must be a letter (A through Z). The *lif*∗(1) utilities accept any file name (including illegal file names generated on other systems), but can only create legal names. This means that files whose names contain lowercase letters can be read but not created.

LIF file names are specified to the *lif*∗(1) utilities by concatenating the HP-UX path name for the LIF volume followed by the LIF file name, separating the two with a colon (**:**). For example:

/dev/fd.0:ABC specifies LIF file **ABC** accessed via HP-UX device special file **/dev/fd.0**.

myfile:ABC specifies LIF file **ABC** within HP-UX disk file **myfile**.

Note that this file-naming convention is applicable only for use as arguments to the *lif*∗(1) utilities, and does not constitute valid path naming for any other use within the HP-UX operating system.

l

Do not mount the special file while using lif(1) utilities.*

SEE ALSO

lifcp(1), lifinit(1), lifls(1), lifrename(1), lifrm(1).

NAME

loadmods - loadable modules to load into the running kernel during boot

DESCRIPTION

The **/etc/loadmods** file contains the names of dynamically loadable kernel modules that are loaded into the running kernel at boot time. If the system administrator wants a dynamically loadable kernel module to be demand loaded on every system reboot, s/he may add the name of the module to this file.

If the **/etc/loadmods** file is present at boot time, the **/sbin/init.d/kminit** script executes the **kmadmin** command and demand loads the modules listed in the file.

An entry for a module consists of a line of the following form:

module_name

The specified module must have been configured as loadable (see *config*(1M)).

Blank lines and lines beginning with '**#**' or '*****' are considered comments and are ignored.

FILES

/sbin/init.d/kminit

SEE ALSO

config(1M), kmsystem(1M), kmadmin(1M), master(4).

l

NAME

localedef - format and semantics of locale definition file

DESCRIPTION

This is a description of the syntax and meaning of the locale definition that is provided as input to the **localedef** command to create a locale (see *localedef*(1M)).

The following is a list of category tags, keywords and subsequent expressions which are recognized by **localedef**. The order of keywords within a category is irrelevant with the exception of the **copy** keyword and other exceptions noted under the **LC_COLLATE** description. (Note that, as a convention, the category tags are composed of uppercase characters, while the keywords are composed of lowercase characters).

Category Tags and Keywords

The following keywords do not belong to any category and should appear in the beginning of the locale definition file:

comment_char
Single character indicating the character to be interpreted as starting a comment line within the locale definition file. This character should be in the first column of a comment line. The default *comment_char* is **#**. All lines with a *comment_char* in the first column are ignored.

escape_char
A single character indicating the character to be interpreted as an escape character within the script. The default *escape_char* is ****. *escape_char* is used to escape localedef metacharacters to remove special meaning and in the character constant decimal, octal, and hexadecimal formats. It is also used to continue a line onto the next, if *escape_char* is the last character on the line (before the new-line character).

The following keywords can be used in any category:

copy
A string naming another valid locale available on the system. This causes the category in the locale being created to be a copy of the same category in the named locale. Since the **copy** keyword defines the entire category, if used, it must be the only keyword in the category.

The following six categories are recognized:

LC_CTYPE:
This category defines character classification, case conversion and other character attributes. The following predefined character classifications are recognized:

upper Character codes classified as uppercase letters. Characters specified in the **cntrl**, **digit**, **punct** or **space** classifications cannot be specified in this category.

lower Character codes classified as lowercase letters. Same restrictions applicable to the **upper** category apply to this classification.

digit Character codes classified as numeric. Only ten characters in contiguous ascending sequence by numerical value can be specified. Alternative digits cannot be specified here.

space Character codes classified as white-space. No character specified for the **upper**, **lower**, **alpha**, **digit**, **graph** or **xdigit** categories can be included in this classification.

punct Character codes classified as punctuation characters. No character included in the **upper**, **lower**, **alpha**, **digit**, **cntrl**, **xdigit** or **space** categories can be specified.

cntrl Character codes classified as control characters. No character included in the **upper**, **lower**, **alpha**, **digit**, **punct**, **graph**, **print** or **xdigit** can be included here.

blank Character codes classified as blank characters. The <space> and <tab> characters are automatically included.

xdigit Character codes classified as hexadecimal digits. Only the characters defined for the **digit** class can be specified, followed by one or more sets of six characters,

1

with each set in ascending order.

alpha Character codes classified as letters. Characters classified as **cntrl**, **digit**, **punct** or **space** cannot be specified. Characters specified as **upper** and **lower** classes are automatically included in this class.

print Character codes classified as printable characters. Characters specified for **upper**, **lower**, **alpha**, **digit**, **xdigit**, and **punct** classes and the <space> character are automatically included. No character from the **cntrl** category can be specified.

graph Character codes classified as printable characters, except the <space> character. In all other respect this classification is similar to the **print** category.

The following two are special classifications, used to designate valid first-of-two and second-of-two **bytes**. Note that these are byte classifications and not character classifications; hence, they cannot be used with the *iswctype* interface (see *wctype*(3C)), in the same manner as the other classifications can be used.

first Valid first bytes of two-byte characters.

second Valid second bytes of two-byte characters.

Character case conversion definitions:

toupper Lowercase to uppercase character relationships.

tolower Uppercase to lowercase character relationships.

Miscellaneous character attribute and classifications:

alt_punct String mapped into the ASCII equivalent string “b!"#$%&’()*+,-./:;<=>?@[\]^_‘{}~”, where b is a blank (a *langinfo*(5) item).

charclass Defines one or more locale-specific character class names as *strings* separated by semicolons. Each named character class can then be defined subsequently in the **LC_CTYPE** definition. The first character of a character class name must be a letter and the class name cannot match any of the predefined classifications (e.g., **space**, **letter**, **cntrl**).

direction String operand indicates text direction (a *langinfo*(5) item). String operand "1" indicates right-to-left text direction.

context String operand indicates character context analysis. String "1" indicates Arabic context analysis is required.

LC_COLLATE:

The **LC_COLLATE** category provides collation sequence definition for relative ordering between collating elements (single- and multi-character collating elements) in the locale. The following keywords belong to this category and should come between the category tag **LC_COLLATE** and **END LC_COLLATE**. The first two keywords can be in any order, but must come before the **order_start** keyword. Any number of the first two keywords can be specified.

collating-element <*symbol***> from** *string*

Defines a multi-character collating element, *symbol*, composed of the characters in *string*. *String* is limited to two characters.

collating-symbol <*symbol***>**

Makes *symbol* a collating symbol which can be used to define a place in the collating sequence. *Symbol* does not represent any actual character.

order_start

Denotes the start of the collation sequence. The directives have an effect on string collation.

The lines following the **order_start** keyword and before the **order_end** keyword contain collating element entries, one per line.

Operands can optionally appear after the **order_start** keyword to defined rules for string comparison using a multiple-weight scheme (if no operands are specified, a single **forward** operand is assumed). The possible operands are:

- **forward** Specifies that comparison operations proceed from start of string towards the end of it.
- **backward** Specifies that comparison operations proceed from end of string towards the beginning of it.

order_end Marks the end of the list of collating element entries.

LC_MONETARY:

The **LC_MONETARY** category defines the rules and symbols used to format monetary numeric information. The following keywords belong to this category and should come between the category tag **LC_MONETARY** and **END LC_MONETARY**:

int_curr_symbol
The operand is a four-character string used to designate the international currency symbol.

currency_symbol
The operand is a string used as the local currency symbol.

mon_decimal_point
The operand is a string containing the symbol used as the decimal delimiter (radix character).

mon_thousands_sep
The operand is a string containing the symbol used as a separator for groups of digits to the left of decimal delimiter.

mon_grouping
The operand is a semicolon-separated list of integers. The initial integer defines the size of the group immediately preceding the decimal delimiter, and the following integers define the preceding groups. If the last integer is not -1, then the size of the previous group (if any) will be repeatedly used for the remainder of the digits. If the last integer is -1, then no further grouping will be performed.

positive_sign
The operand is a srting to indicate a non-negative monetary quantity.

negative_sign
The operand is a srting to indicate a negative monetary quantity.

int_frac_digits
The operand is an integer representing the number of fractional digits used in formatted monetary values using **int_curr_symbol**.

frac_digits
The operand is an integer representing the number of fractional digits used in formatted monetary values using **currency_symbol**.

p_cs_precedes
The operand is an integer which if set to 1 indicates the **currency_symbol** or **int_curr_symbol** precedes a monetary quantity, and if set to 0 the symbol succeeds the value.

p_sep_by_space
The operand is an integer which if set to 1 indicates a space separates the **currency_symbol** or **int_curr_symbol** from the value, and otherwise if set to 0.

n_cs_precedes
The operand is an integer which if set to 1 indicates the **currency_symbol** or **int_curr_symbol** precedes a negative monetary quantity, and if set to 0 the symbol succeeds the negative value.

n_sep_by_space
The operand is an integer which if set to 1 indicates a space separates the **currency_symbol** or **int_curr_symbol** from negative monetary value, and otherwise if set to 0.

p_sign_posn
The operand is an integer which setting indicates the positioning of the

1

positive_sign for a non-negative monetary quantity. The possible values are:

- **0** Parenthesis surround the quantity and the **currency_symbol** or **int_curr_symbol**.
- **1** The sign string precedes the quantity and the **currency_symbol** or **int_curr_symbol**.
- **2** The sign string succeeds the quantity and the **currency_symbol** or **int_curr_symbol**.
- **3** The sign string precedes the **currency_symbol** or **int_curr_symbol**.
- **4** The sign string succeeds the **currency_symbol** or **int_curr_symbol**.

n_sign_posn
The operand is an integer which setting parallels that of **p_sign_posn**, but for negative monetary quantities.

LC_NUMERIC:
The **LC_NUMERIC** category defines rules and symbols used to format non-monetary numeric information. The following keywords belong to this category and should come between the category tag **LC_NUMERIC** and **END LC_NUMERIC**:

decimal_point
The operand is a string containing the symbol used as the decimal delimiter (radix character) in numeric, non-monetary formatted quantities. This keyword cannot be omitted and cannot be set to the empty string.

thousands_sep
The operand is a string containing the symbol used as a separator for groups of digits to the left of the decimal delimiter.

grouping The operand is a semicolon-separated list of integers. The initial integer defines the size of the group immediately preceding the decimal delimiter, and the following integers define the preceding groups. If the last integer is not -1, then the size of the previous group (if any) will be repeatedly used for the remainder of the digits. If the last integer is -1, then no further grouping will be performed.

alt_digit String mapped into the ASCII equivalent string “0123456789b+-.,eE”, where b is a blank (a *langinfo*(5) item). The **alt_digit** keyword is a HP extension to the **localedef** POSIX standards and it has a different meaning than the **alt_digits** defined in POSIX standards.

LC_TIME:
The **LC_TIME** category defines the rules for generating locale-specific formatted date strings. The following mandatory keywords belong to this category and should come between the category tag **LC_TIME** and **END LC_TIME**:

abday Seven semicolon-separated strings giving abbreviated names for the days of the week beginning with Sunday.

day Seven semicolon-separated strings giving full names for the days of the week beginning with Sunday.

abmon Twelve semicolon-separated strings giving abbreviated names for the months, beginning with January.

mon Twelve semicolon-separated strings giving full names for the months, beginning with January.

d_t_fmt The operand is a string defining the appropriate date and time representation.

d_fmt The operand is a string defining the appropriate date representation.

t_fmt The operand is a string defining the appropriate time representation.

am_pm The operand is two semicolon-separated strings giving the representations for **AM** and **PM**.

t_fmt_ampm The operand is a string defining the appropriate time representation in the 12-hour clock format with **am_pm**.

era The operand is a semi-colon-separated list of strings. Each string defines the name and date of an era or emperor for a locale. Each string should conform to the following format:

direction:*offset*:*start_date*:*end_date*:*name*:*format*

where:

direction Either a **+** or **-** character. The **+** character indicates the time axis should be such that the years count in the positive direction when moving from the starting date towards the ending date. The **-** character indicates the time axis should be such that the years count in the negative direction when moving from the starting date towards the ending date.

offset A number in the range **[SHRT_MIN,SHRT_MAX]** indicating the number of the first year of the era.

start_date A date in the form *yyyy***/***mm***/***dd* where *yyyy*, *mm*, and *dd* are the year, month and day numbers, respectively, of the start of the era. Years prior to the year 0 A.D. are represented as negative numbers. For example, an era beginning March 5th in the year 100 B.C. would be represented as **3-100/3/5**. Years in the range **[SHRT_MIN+1,SHRT_MAX-1]** are supported.

end_date The ending date of the era in the same form as the *start_date* above or one of the two special values **-*** or **+***. A value of **-*** indicates the ending date of the era extends to the beginning of time while **+*** indicates it extends to the end of time. The ending date can be chronologically either before or after the starting date of an era. For example, the expressions for the Christian eras A.D. and B.C. would be:

```
+:0:0000/01/01:+*:A.D.:%o %N
+:1:-0001/12/31:-*:B.C.:%o %N
```

name A string representing the name of the era which is substituted for the **%N** directive of **date** and **strftime()** (see *date*(1) and *strftime*(3C)).

format A string for formatting the **%E** directive of *date*(1) and *strftime*(3C). This string is usually a function of the **%o** and **%N** directives. If *format* is not specified, the string specified for the **LC_TIME** category keyword **era_d_fmt** (see below) is used as a default.

era_d_fmt The operand is a string defining the format of date in era notation.

era_t_fmt The operand is a string defining the format of time in era notation.

era_d_t_fmt
The operand is a string defining the format of date and time in era notation.

alt_digits The operand is a semi-colon-separated list of strings. The first string is the alternative symbol corresponding to zero, the second string is the alternative symbol corresponding to one, and so on. Note that if the HP-UX-proprietary **alt_digit** keyword has been specified in the same locale, the first ten symbols should be identical for these two keywords.

In addition to the above, the following HP-UX-proprietary keywords are recognized (these are provided for backward compatibility and their use is otherwise not recommended): **year_unit**, **mon_unit**, **day_unit**, **rour_unit**, **min_unit**, **sec_unit**.

LC_MESSAGES:
The **LC_MESSAGES** category defines the format and values for affirmative and negative responses. The following keywords belong to this category and should come between the category tag

1

LC_MESSAGES and **END LC_MESSAGES**:

yesexpr The string operand is an Extended Regular Expression matching acceptable affirmative responses to yes/no queries.

noexpr The string operand is an Extended Regular Expression matching acceptable negative responses to yes/no queries.

yesstr The string operand identifies the affirmative response for yes/no questions. This keyword is now obsolete and **yesexpr** should be used instead.

nostr The string operand identifies the negative response for yes/no questions This keyword is now obsolete and **noexpr** should be used instead.

Keyword Operands

Keyword operands consist of character-code constants and symbols, strings, and metacharacters. The types of legal expressions are: **character lists**, **string lists**, **integer lists**, **shift**, **collating element entries**, **regular expression**, **character constants** and **string**:

character lists

character list operands consist of single character-code constants or symbolic names separated by semicolons, or a character-code range consisting of a constant or symbolic name followed by an ellipsis followed by another constant or symbolic name. The constant preceding the ellipsis must have a smaller code value than the constant following the ellipsis. A range represents a set of consecutive character codes. If the list is longer than a single line, the escape character must be used at the end of each line as a continuation character. It is an error to use any symbolic name that is not defined in an accompanying charmap file (see *charmap*(4)).

l

string lists

string list operands consist of strings separated by semicolons. If longer than one line, the escape character must be used for continuation.

string **string** operands consist of a sequence of zero or more characters surrounded by double quotes ("). Within a string, the double-quote character must be preceded by an escape character. The following escape sequences also can be used:

\n newline

\t horizontal tab

\b backspace

\r carriage return

\f form feed

**** backslash

\' single quote

*****ddd* bit pattern

The escape *****ddd* consists of the escape character followed by 1, 2, or 3 octal digits specifying the value of the desired character (for other possible bit pattern specification, see *character constants* below). Also, an escape character (\) and an immediately-following newline are ignored.

Although the backslash (\) has been used for illustration, another escape character can be substituted by the **escape_char** keyword.

character constants

Constants represent character codes in the operands. They can be used in the following forms:

decimal constants	An escape character followed by a **'d'** followed by up to three decimal digits.
octal constants	An escape character followed by up to three octal digits.
hexadecimal constants	An escape character followed by a **'x'** followed by two hexadecimal digits.

character constants — A single character (e.g., A) having the numerical value of the character in the machine's character set.

symbolic names — A string enclosed between **<** and **>** is a symbolic name. **localedef** input files are recommended to be written entirely in symbolic names, utilizing a user defined or system-supplied charmap file. This aids portability of **localedef** input files between different encoded character sets (see *charmap*(4)).

Symbolic names can be defined within a locale definition file by the **collating-element** and **collating-symbol** keywords. These are not character constants. It is an error if such an internally defined symbolic name collides with one defined in a charmap file.

integer lists
Integer list operands consists of one or more decimal digits separated by semicolons.

shift **Shift** operands follow keywords **toupper** and **tolower**, and must consist of two character-code constants enclosed by left and right parentheses and separated by a comma. Each such character pair is separated from the next by a semicolon. For **tolower**, the first constant represents an uppercase character and the second the corresponding lowercase character. For **toupper**, the first constant represents an lowercase character and the second the corresponding uppercase character.

collating element entry
The **order_start** keyword is followed by collating element entries, one per line, in ascending order by collating position. The collating element entries have the form:

collation_element[*weight*[*;weight*]]

collation_element can be a character, a collating symbol enclosed in angle brackets representing a character or collating element, the special symbol **UNDEFINED** or an ellipsis (**...**).

A character stands for itself; a collating symbol can be a symbolic name for a character that is interpreted by the charmap file, a multi-character collating element defined by a **collating-element** keyword, or a collating symbol defined by the **collating-symbol keyword**.

The special symbol **UNDEFINED** specifies the collating position of any characters not explicitly defined by collating element entries. For example, if some group of characters is to be omitted from the collation sequence and just collate after all defined characters, a collating symbol might be defined before the **order_start** keyword:

```
collating-symbol  <HIGH>
```

Then somewhere in the list of collating element entries:

```
UNDEFINED  <HIGH>
```

Notice that there is no second weight. This means that on a second pass all characters collate by their encoded value.

An ellipsis is interpreted as a list of characters with an encoded value higher than that of the character on the preceding line and lower than that on the following line. Because it is tied to encoded value of characters, the ellipsis is inherently non-portable. If it is used, a warning is issued and no output generated unless the **-c** option was given.

The *weight* operands provide information about how the collating element is to be collated on first and subsequent passes. *Weight* can be a two-character string, the special symbol **IGNORE**, or a collating element of any of the forms specified for *collating_element* except **UNDEFINED**. If there are no *weights*, the character is collating strictly by its position in the list. If there is only one *weight* given, the character sorts by its relative position in the list on the second collation pass.

An equivalence class is defined by a series of collating element entries all having the same character or symbol in the first *weight* position. For example, in many locales all forms of the character This is represented in the collating element entries as:

```
'A'    'A';'A' # first element of equivalence class
'a'    'A';'a' # next element of class
```

Two-to-one collating elements are specified by *collating-elements* defined before the **order_start** keyword. For example, the two-to-one collating element **CH** in Spanish, would be defined before the **order_start** keyword as

```
collating element <CH> from CH
```

It would then be used in a collating element entry as **<CH>**.

A one-to-two collating element is defined by having a two-character string in one of the *weight* positions. For example, if the character **'X'** collates equal to the pair "AE", the collating element entry would be:

```
'X' AE ;'X'
```

A don't-care character is defined by the special symbol **IGNORE**. For example, the dash character, **'-'** may be a don't care on the first collation pass. The collating element entry is:

```
'-'   IGNORE;'-'
```

Symbols defined by the **collating-symbol** keyword can be used to indicate that a given character collates higher or lower than some position in the sequence. For example if all characters with an encoded value less than that of **'0'** are to collate lower than all other characters on the first pass, and in relative order on the second pass, define a collating symbol before the **order_start** keyword:

```
collating-symbol    <LOW>
```

l

The first two collating element entries are then:

```
...    <LOW>;...
'0'    '0';'0'
```

This also illustrates the use of the ellipsis to indicate a range. The first ellipsis is interpreted as "all characters in the encoded character set with a value lower than '0'"; the second ellipsis means that all characters in the range defined by the first collate in relative order.

regular expression

regular expression operands conform to the Extended Regular Expressions specifications as described in *regexp*(5).

Metacharacters

Metacharacters are characters having a special meaning to *localedef* in operands. To escape the special meaning of these characters, surround them with single quotes or precede them by an escape character. *localedef* meta-characters include:

- **<** Indicates the beginning of a symbolic name.
- **>** Indicates the end of a symbolic name.
- **(** Indicates the beginning of a character shift pair following the **toupper** and **tolower** keywords.
- **)** Indicates the end of a character shift pair.
- **,** Used to separate the characters of a character shift pair.
- **"** Used to quote strings.
- **;** Used as a separator in list operands.

escape character

Used to escape special meaning from other metacharacters and itself. It is backslash (\) by default, but can be redefined by the **escape_char** keyword.

Comments

Comments are lines beginning with a comment character. The comment character is pound sign (#) by default, but can be redefined by the **comment_char** keyword. Comments and blank lines are ignored.

Separators

Separator characters include blanks and tabs. Any number of separators can be used to delimit the keywords, metacharacters, constants and strings that comprise a *localedef* script except that all characters between **<** and **>** are considered to be part of the symbolic name even they are <blank>s.

EXAMPLE

Please see the files under **/usr/lib/nls/loc/src** for examples of locale description files. These files were used to create the various locales which are delivered with HP-UX.

NAME

lvmpvg - LVM physical volume group information file

SYNOPSIS

/etc/lvmpvg

DESCRIPTION

lvmpvg is an ASCII file that stores the volume-group information for all of the physical volume groups in the system. The information is stored in a hierarchical format.

First, it starts with a volume group under which multiple physical volume groups can exist. Under each physical volume group, a list of physical volumes can be specified. There must be at least one physical volume group in each volume group that appears in this file. The physical-volume-group name must be unique within the corresponding volume group, although it is permissible to use a common physical volume group name across different volume groups. There can be as many volume groups in this file as there are in the system.

Instead of using the **vgcreate** and **vgextend** commands, the administrator can edit this file to create and extend physical volume groups. However, care must be taken to ensure that all physical volumes to be included in the file have already been defined in their respective volume groups by previous use of **vgcreate** or **vgextend**.

The **lvmpvg** file format has the following structure. **VG** and **PVG** are keywords that introduce the names of the *volume group* and *physical volume group*, respectively. *No comments are allowed in this file*.

VG *vg_name*
PVG *pvg_name*
pv_path
...
PVG *pvg_name*
pv_path
...
VG *vg_name*
PVG *pvg_name*
pv_path
...

The variables are defined as follows:

pv_path The block device path name of a physical volume within the volume group.

pvg_name The name of the physical volume group. It must be unique within the volume group.

vg_name The path name of the volume group.

EXAMPLES

The following example shows an **lvmpvg** file containing two volume groups: the first containing two physical volume groups, each with two physical volumes defined in it; the second containing three physical volume groups, each with one physical volume defined in it.

```
VG   /dev/vg00
PVG  PVG0
/dev/dsk/c2t0d0
/dev/dsk/c2t1d0
PVG  PVG1
/dev/dsk/c3t0d0
/dev/dsk/c3t1d0
VG   /dev/vg01
PVG  PVG0
/dev/dsk/c4t0d0
PVG  PVG1
/dev/dsk/c5t0d0
PVG  PVG2
/dev/dsk/c6t0d0
```

SEE ALSO

vgcreate(1M), vgextend(1M), vgreduce(1M), vgremove(1M).

NAME

magic - magic numbers for HP-UX implementations

SYNOPSIS

```
#include <magic.h>
```

DESCRIPTION

The **magic.h** file localizes all information about HP-UX "magic numbers" in one file, thus facilitating uniform treatment of magic numbers. This file specifies the location of the magic number in a file (always the start of the file) and the structure of the magic number:

```
struct magic_number {
    unsigned short    system_id;
    unsigned short    file_type;
};
typedef      struct    magic_number MAGIC;
```

magic.h includes definitions for the system IDs of all HP machines running HP-UX, and file types that are common to all implementations. There may be additional implementation-dependent file types. The predefined file types are:

```
/* for object code files */
    #define RELOC_MAGIC     0x106   /* relocatable only */
    #define EXEC_MAGIC      0x107   /* normal executable */
    #define SHARE_MAGIC     0x108   /* shared executable */
    #define DEMAND_MAGIC    0x10B   /* demand-load executable */
    #define LISP_MAGIC      0x10C   /* compiled Lisp */
    #define DL_MAGIC        0x10D   /* dynamic load library */
    #define SHL_MAGIC       0x10E   /* shared library */
    #define HPE_MAGIC       0x150   /* HPE boot image */
```

m

The values for **system_id** are defined in *model*(4).

WARNINGS

Files managed by **cpio** use a different form of magic number that is incompatible with <**magic.h**>.

SEE ALSO

ar(1), ld(1), a.out(4), ar(4), model(4).

NAME

master - master kernel configuration information

DESCRIPTION

A master file contains sections of information in a form suitable for **config**, enabling it to create a kernel configuration file. Master files are found in the directory **/usr/conf/master.d**.

Master files are of two types:

- A kernel master file is of the type which usually carries information on several drivers/subsystems.
- A kernel module master file carries information on an individual module. Such master files are named after the module to which they belong and are installed onto a system via **kminstall**.

Each section of a master file begins with a line containing a **$** in column one followed by a section keyword. The section continues to the end of the file or until a line containing only three **$** characters is encountered. Lines beginning with an asterisk (*) are comments.

Kernel Master File

The following table lists the section keywords for the kernel master file and their purpose. Note that some of the section keywords may also be used for the kernel module master file described later:

Section keyword	Section purpose
$DEVICE	Device driver specification
$CDIO	Context Dependent I/O table
$DRIVER_INSTALL	List of drivers with installation functions
$DYN_MAJOR	Dynamic block and character major numbers
$ALIAS	Driver alias table
$TUNABLE	Tunable parameters
$DRIVER_DEPENDENCY	Driver-to-driver dependency table
$DRIVER_LIBRARY	Library location of driver table
$LIBRARY	Required/optional library table
$SUBSYSTEMS_DEFINE	Subsystems requiring **#defines**
$STREAMS_SYNC_LEVEL	STREAMS synchronization level table
$STREAMS_DVR_SYNC	STREAMS driver and module synchronization table

Each section consists of text fields separated by space and tab characters and is described separately below. Bit mask fields are expressed as hexadecimal values which are constructed by computing the logical OR of the component bit values.

$DEVICE Section

NOTE: This section is provided for compatibility with previous HP-UX releases. New drivers should be added to the **$DRIVER_INSTALL** section.

Software drivers are defined using five fields defined as follows:

Field_1 Device name, used in the user-specified *system_file* (8 characters maximum).

Field_2 Handler name, used by the kernel to prefix routines such as **cs80_read**, **lp_write**, and others (8 characters maximum).

Field_3 Driver characteristics, which are specified by computing the logical OR using the hexadecimal bit mask value of the following seven bits.

0x40	STREAMS module
0x20	STREAMS driver
0x10	I/O card or pseudo driver
0x08	Allow only one specification of driver
0x04	Required device (included in all systems)
0x02	Block device
0x01	Character device

Field_4 Functions for the device, specified by creating a bit mask using the following bits:

0x100000	Turn off map buffer to kernel flag (**C_MAP_BUFFER_TO_KERNEL**)
0x010000	Set driver is multiprocessor capable flag (**C_MGR_IS_MP**)
0x008000	Set STREAMS clone major device flag (**C_CLONESMAJOR**)

m

0x004000	Set STREAMS System V release 3 style open flag (**SVR3_OPEN**)
0x002000	Set STREAMS System V release 4 style open flag (**SVR4_OPEN**)
0x001000	Autochanger **mount** routine exists
0x000800	**option1** handler exists (Series 700 only)
0x000400	**dump** handler exists
0x000200	**size** handler exists
0x000100	**link** routine exists
0x000080	**open** handler exists
0x000040	**close** handler exists
0x000020	**read** handler exists
0x000010	**write** handler exists
0x000008	**ioctl** handler exists
0x000004	**select** handler exists
0x000002	**seltru** handler exists
0x000001	Set device **close** routine called on all closes flag (**C_ALLCLOSES**)

Field_5 Block major device number if a block-type device; otherwise -1.

Field_6 Character major device number if a character-type device; otherwise -1.

$CDIO Section

CDIO (Context Dependent I/O) list. List of I/O modules specific to the bus and/or driver environment, and whether they are required in a minimal system.

Field_1 CDIO name.

Field_2 1 if the CDIO is required for a minimal system; otherwise 0.

$DYN_MAJOR Section

Dynamic major numbers. A range of block and character major numbers reserved for drivers whose major numbers are assigned dynamically.

m

Field_1 **block** or **char**.

Field_2 A major number or a range of major numbers. A range is specified as *lo_major_num* **-** *hi_major_num*.

$DRIVER_INSTALL Section

Driver install section is a list of drivers, shown with their block and character major numbers

Field_1 Driver name.

Field_2 Block major device number if a block-type device; otherwise -1.

Field_3 Character major device number if a character-type device; otherwise -1.

Field_4 1 if the driver is required for a minimal system; otherwise 0.

$ALIAS Section

Aliases for names are defined as follows:

Field_1 Alias name => product number (8 characters maximum)

Field_2 Device name (8 characters maximum)

$TUNABLE Section

Tunable parameters are defined as follows:

Field_1 Parameter name as used in the user-specified system_file (20 characters maximum).

Field_2 Parameter name as used in the **#define** statement in **tune.h** (20 characters maximum). In previous releases, the **#define** statement in which the parameter name was used, was in **conf.c**.

Field_3 Default value for the parameter (60 characters maximum).

Field_4 Minimum value for the parameter (60 characters maximum).

$DRIVER_DEPENDENCY Section

List of drivers and the other drivers they depend on.

Field_1 Dependent driver.

Field_2-N Name of supporting drivers or CDIO's.

$DRIVER_LIBRARY Section

List of drivers and the library or libraries containing the driver object code.

Field_1 Driver name.

Field_2-N Name of libraries containing driver code.

$LIBRARY Section

Library list. List of object code libraries and whether they are required is a minimal system.

Field_1 Library name.

Field_2 1 if the library is required for a minimal system; otherwise 0.

$SUBSYSTEMS_DEFINE Section

List of subsystems and/or drivers that require **#define** *IDENTIFIER* statements in **conf.c**. If needed, the identifier will be converted to upper case.

Field_1 Subsystem/driver name.

Field_2 (Optional) Name of identifier to define. If this field is not present, the identifier will be *Field_1* in upper case.

$STREAMS_SYNC_LEVEL Section

List of possible STREAMS synchronization levels. Please refer to the documentation that accompanied the STREAMS/UX product for a more detailed description of this table and STREAMS synchronization levels.

Field_1 Synchronization level.

m

$STREAMS_DVR_SYNC Section

List of STREAMS modules and drivers and the synchronization levels that they require. Please refer to the documentation that accompanied the STREAMS/UX product for more information about this table.

Field_1 Driver or module name.

Field_2 Synchronization level. (Must be present in a **$STREAM_SYNC_LEVEL** list.)

Field_3 (Optional) Additional STREAMS synchronization information.

Kernel Module Master File

The following section keywords and purposes are used only in the kernel module master files.

Section keyword	Section purpose
$VERSION	File format version
$LOADABLE	Load capability of module
$INTERFACE	Interface used by module
$TYPE	Module type specific information

If required, kernel module master files may use the following section keywords and purposes described earlier.

Section keyword	Section purpose
$DRIVER_DEPENDENCY	Dependency to other kernel modules.
$TUNABLE	Same as **$TUNABLE** section.
$DRIVER_INSTALL	Same as **$DRIVER_INSTALL**.

For kernel modules, **$DRIVER_INSTALL** section information is used to link the kernel module into the kernel statically. The first field of this section indicates the module_name.

Each section consists of text fields separated by space and tab characters and is described separately below.

$VERSION Section

Format version.

Format version starts from one.

Field_1 Version number. (decimal number)

Example

```
$VERSION
1
$$$
```

$LOADABLE Section

Capability of a kernel module.

If the section exists, the module is dynamically loadable. Otherwise it can be only statically linked into the kernel. Boot device related kernel modules should not supply the section. Kernel module without **$LOADABLE** section cannot be configured as dynamically loadable module.

Example

```
$LOADABLE
$$$
```

If the module is using stub, keyword **stub** should be specified within the section.

Example

```
$LOADABLE
stub
$$$
```

m

$INTERFACE Section

List of used interfaces by kernel modules.

NOTE: **base** may be specified in Field 1 alternatively. If **base** is specified, interface enforcement and version control will be exempted and module will need to be maintained by its developer to be in synchronization with kernel or other components.

Field_1 Interface name. (string)

Field_2 Version name. (string)

Example

```
$INTERFACE
wsio2 1
xyz   4
$$$
```

$TYPE Section

Module type and type specific information list.

Field_1 Kernel module name.

Field_2 Module type name.

wsio2_class, **wsio2_intfc**, **wsio_class**, **wsio_intfc**, **streams_mod**, **streams_drv**, **misc** are valid.

Fields 3 - 6 contains module type specific fields for these types; **wsio2_class**, **wsio2_intfc**, **wsio_class**, **wsio_intfc**, **streams_drv**:

Field_3 Class name.

Field_4 Flags.

- **c** character device driver.
- **b** block device driver.
- **p** pseudo driver.

- **s** supports scanning.
- **m** MP capable driver.
- **i** Save information to ioconfig.

Field_5 block device major number.

Field_6 character device major number.

Example

```
$TYPE
wsio2 stape2 tape c -1 203
$$$
```

EXAMPLES

The following entry in the **$DRIVER_INSTALL** section will enable the kernel to dynamically assign block and/or character major number(s) for a custom driver, **mydriver**.

```
mydriver    -1    -1    0
```

FILES

/usr/conf/master.d Master files directory

SEE ALSO

kminstall(1M),config(1M), kmsystem(1M).

m

NAME

mnttab - mounted file system table

SYNOPSIS

#include <mntent.h>

DESCRIPTION

mnttab resides in directory **/etc** and contains a table of devices mounted by the **mount** command (see *mount*(1M)). The file contains a line of information for each mounted filesystem which is structurally identical to the contents of **/etc/fstab** described by *fstab*(4).

There are a number of lines of the form:

special_file_name dir type opts freq passno mount_time

consisting of entries similar to:

```
/dev/dsk/c0d0s0  /  hfs  rw  0  1  537851723
```

/etc/mnttab is accessed by programs that use **getmntent()** (see *getmntent*(3X)), It should never be manually edited, nor should **setmnt** ever be used to create invalid entries in **/etc/mnttab** (see *setmnt*(1M)).

mount_time contains the time the file system was mounted using **mount**. Its value is the number of seconds since the Epoch (00:00:00 Coordinated Universal Time, January 1, 1970 (see *time*(2).

mount and **umount** rewrite the **mnttab** file whenever a file system is mounted or unmounted if **mnttab** is found to be out of date with the mounted file system table maintained internally by the HP-UX kernel. **syncer** also updates **mnttab** if it is out of date (see *syncer*(1M).

WARNINGS

The table is provided only as a means for programs to return information about mounted file systems.

/etc/mnttab should never be manually edited. Any manual changes made to **/etc/mnttab** are overwritten without warning by **syncer**, **mount**, and **umount**.

m

AUTHOR

mnttab was developed by the University of California, Berkeley, Sun Microsystems, Inc., and HP.

FILES

/etc/mnttab

SEE ALSO

mount(1M), getmntent(3X), fstab(4).

NAME

model - HP-UX machine identification

SYNOPSIS

#include <model.h>

DESCRIPTION

There are certain inevitable distinctions between HP-UX implementations due to hardware differences. Where such distinctions exist, conditional compilation or other definitions can be used to isolate the differences. Flags and typedefs to resolve these distinctions are collected in the <**model.h**> header file which contains constants identifying various HP-UX implementations.

For example, header file **model.h** contains the following constants whose values are defined in <**sys/magic.h**>:

```
#define   HP_S_500   HP9000_ID
#define   HP_S_200   HP98x6_ID
#define   HP_S_300   CPU_HP_MC68020
#define   HP_S_800   CPU_PA_RISC1_0
#define   HP_S_700   CPU_PA_RISC1_1
```

Other such constants are added as appropriate when HP-UX extends to other machines in subsequent releases.

In addition, **model.h** has a statement defining the preprocessor constant **MYSYS** to represent the specific implementation for which compilation is desired. **MYSYS** is always equal to one of the constants above.

Conditional compilation can be used to adapt a single file for execution on more than one HP-UX implementation if the file contains implementation- or architecture-dependent features. For example, the code segment:

```
#if MYSYS==HP_S_400
     <statements>
#endif
```

causes statements following the if statement to be compiled *only* if the system processor is an HP 9000 Series 400 machine.

model.h also contains typedefs for several predefined types to enhance portability of certain types of code and files.

int8, **u_int8**	Signed and unsigned 8-bit integers.
int16, **u_int16**	Signed and unsigned 16-bit integers.
int32, **u_int32**	Signed and unsigned 32-bit integers.
machptr, **u_machptr**	Signed and unsigned integers large enough to hold a pointer.

Certain C preprocessor conditional compilation variables are defined to aid in implementation-dependent code. See *cpp*(1).

SEE ALSO

cc(1), cpp(1), magic(4).

m

NAME

netconfig - network configuration database

SYNOPSIS

/etc/netconfig

DESCRIPTION

The network configuration database, **/etc/netconfig**, is a system file used to store information about networks that are connected to the system. The **netconfig** database and the routines that access it (see *getnetconfig*(3N)) are part of the Network Selection component. The Network Selection component also includes **getnetpath()** routines to provide application-specific network search paths. These routines access the **netconfig** database based on the environment variable *NETPATH* (see *environ*(5)).

netconfig contains an entry for each network available on the system. Entries are separated by newlines. Fields are separated by whitespace and occur in the order in which they are described below. Whitespace can be embedded as *blank* or *tab*. Lines in **/etc/netconfig** that begin with a **#** (hash) in column 1 are treated as comments.

Each of the valid lines in the **netconfig** database correspond to an available transport. Each entry is of the form:

network_ID semantics_flag protocol_family protocol_name network_device translation_libraries

network_ID A string used to uniquely identify a network. *network_ID* consists of non-null characters, and has a length of at least 1. No maximum length is specified. This namespace is locally significant and the local system administrator is the naming authority. All *network_ID*'s on a system must be unique.

semantics The *semantics* field is a string identifying the "semantics" of the network, that is, the set of services it supports, by identifying the service interface it provides. The *semantics* field is mandatory. The following semantics are recognized.

tpi_clts Transport Provider Interface, connectionless

tpi_cots_ord Transport Provider Interface, connection oriented, supports orderly release.

n

flag The *flag* field records certain two-valued ("true" and "false") attributes of networks. *flag* is a string composed of a combination of characters, each of which indicates the value of the corresponding attribute. If the character is present, the attribute is "true." If the character is absent, the attribute is "false." "**-**" indicates that none of the attributes are present. Only one character is currently recognized:

v Visible ("default") network. Used when the environment variable *NETPATH* is unset.

protocol_family

The *protocol_family* and *protocol_name* fields are provided for protocol-specific applications.

The *protocol_family* field contains a string that identifies a protocol family. The *protocol_family* identifier follows the same rules as those for *network_ID*s; the string consists of non-null characters, it has a length of at least 1, and there is no maximum length specified. A - in the *protocol_family* field indicates that no protocol family identifier applies (the network is experimental). An example protocol family:

inet Internetwork: UDP, TCP, etc.

protocol_name

The *protocol_name* field contains a string that identifies a protocol. The *protocol_name* identifier follows the same rules as those for *network_ID*s; that is, the string consists of non-NULL characters, it has a length of at least 1, and there is no maximum length specified. A "**-**" indicates that none of the names listed apply. The following protocol names are recognized.

tcp Transmission Control Protocol

udp User Datagram Protocol

network_device

The *network_device* is the full pathname of the device used to connect to the transport provider. Typically, this device will be in the */dev* directory. The *network_device* must be specified.

translation_libraries
The *name-to-address translation libraries* support a "directory service" (a name-to-address mapping service) for the network. A "**-**" in this field indicates the absence of any *translation_libraries*. This has a special meaning for networks of the protocol family *inet*: its name-to-address mapping is provided by the name service switch based on the entries for *hosts* and *services* in **switch()** (see *nsswitch.conf*(4)). For networks of other families, a "**-**" indicates non-functional name-to-address mapping. Otherwise, this field consists of a comma-separated list of pathnames to dynamically linked libraries. The pathname of the library can be either absolute or relative.

Each field corresponds to an element in the **struct netconfig** structure. **struct netconfig** and the identifiers described on this manual page are defined in **<netconfig.h>**. This structure includes the following members:

char *nc_netid Network ID, including NULL terminator.

unsigned long nc_semantics
Semantics.

unsigned long nc_flag Flags.

char *nc_protofmly Protocol family.

char *nc_proto Protocol name.

char *nc_device Full pathname of the network device.

unsigned long nc_nlookups
Number of directory lookup libraries.

char **nc_lookups Names of the name-to-address translation libraries.

unsigned long nc_unused[9]
Reserved for future expansion.

The *nc_semantics* field takes the following values, corresponding to the semantics identified above:

NC_TPI_CLTS
NC_TPI_COTS_ORD

The *nc_flag* field is a bitfield. The following bit, corresponding to the attribute identified above, is currently recognized. **NC_NOFLAG** indicates the absence of any attributes.

NC_VISIBLE

EXAMPLES

Below is a sample **netconfig** file:

```
#
# The 'Network Configuration' File.
#
# Each entry is of the form:
#
#   <network_id> <semantics> <flags> <protofamily> <protoname> <device> \
#         <nametoaddr_libs>
#
# The '-' in <nametoaddr_libs> for inet family transports indicates
# redirection to the name service switch policies for 'hosts' and
# 'services'. The '-' may be replaced by nametoaddr libraries that
# comply with the SVr4 specs, in which case the name service switch
# will not be used for netdir_getbyname, netdir_getbyaddr,
# gethostbyname, gethostbyaddr, getservbyname, and getservbyport.
# There are no nametoaddr_libs for the inet family, and currently
# nametoaddr_libs are not supported.
#
udp  tpi_clts   v    inet udp  /dev/udp   -
tcp  tpi_cots_ord    v    inet tcp       /dev/tcp   -
```

n

AUTHOR

netconfig was developed by Sun Microsystems, Inc.

FILES

<netconfig.h>

/etc/netconfig

SEE ALSO

getnetconfig(3N), getnetpath(3N), nsswitch.conf(4).

n

NAME

netgroup - list of network groups

DESCRIPTION

File **/etc/netgroup** defines network-wide groups, and is used for permission checking when executing remote mounts, remote logins, and remote shells. For remote mounts, the information in **netgroup** classifies machines; for remote logins and remote shells, it classifies users. Each line of the **netgroup** file defines a group and has the format

groupname member1 member2 ...

where member *i* is either another group name, or a triple.

(*hostname, username, domainname***)**

If any of these three fields are left empty, it signifies a wild card. Thus

universal (,,)

defines a group to which everyone belongs. Field names that begin with something other than a letter, digit or underscore (such as **-**) do not match any value. For example, consider the following entries.

```
justmachines      (analytica,-,YOURDOMAIN)
justpeople        (-,root,YOURDOMAIN)
```

Machine **analytica** belongs to the group **justmachines** in the domain **YOURDOMAIN**, but no users belong to it. Similarly, the user **root** belongs to the group **justpeople** in the domain **YOURDOMAIN**, but no machines belong to it.

Note, the domain name field must match the current domain name (as returned by the **domainname** command), or the entry is not matched. Also, the user-name field is ignored for remote mounts. Only the hostname and domainname are used.

The Network Information Service (NIS) can serve network groups. When so used, they are stored in the following NIS maps.

netgroup
netgroup.byuser
netgroup.byhost

Refer to *ypserv*(1M) and *ypfiles*(4) for an overview of Network Information Service.

AUTHOR

netgroup was developed by Sun Microsystems, Inc.

FILES

/etc/netgroup

SEE ALSO

makedbm(1M), mountd(1M), ypmake(1M), ypserv(1M), getnetgrent(3C), hosts.equiv(4), ypfiles(4).

Installing and Administering NFS Services, Chapter 7: *NIS Configuration*.

n

NAME

netrc - login information for **ftp** and **rexec**

DESCRIPTION

The **.netrc** file contains login and initialization information used by the **ftp** autologin process, by the **rexec()** library routine, and by the **rexec** command (see *ftp*(1), *rexec*(3N), and *remsh*(1)), respectively. This file is optional. It exists, if at all, in the user's home directory.

If the **.netrc** file contains password or account information for use other than for anonymous **ftp**, its owner must match the effective user ID of the current process. Its read, write, and execute mode bits for group and other must all be zero, and it must be readable by its owner. Otherwise, the file is ignored.

The file can contain the following tokens, separated by white space (spaces, tabs, or newlines) or commas (**,**). To include a comma as part of a token, enclose that token in quotation marks (**"**).

machine *name* — Identify a remote machine name. The autologin process searches the **.netrc** file for a **machine** token that matches the remote machine specified on the **ftp** command line, as an **ftp open** command argument, or as the *****_ahost_ parameter to **rexec()**. Once a match is made, the subsequent **.netrc** tokens are processed, stopping when the end-of-file is reached or another **machine** token or a **default** token is encountered.

default — Same as **machine** *name* except that **default** matches any name. There can be only one **default** token, and it must be after all **machine** tokens. This is normally used for **ftp** as follows:

```
default login anonymous password user@site
```

This provides automatic anonymous **ftp** login to machines not specified in **.netrc**. This can be overridden in **ftp** by using the **-n** flag to disable autologin.

login *name* — Identify a user on the remote machine. If this token is present, the **ftp** or **rexec()** autologin process initiates a login using the specified name. If this token matches the user name used by the **rexec -l** command option, or, by default, the local user name, **rexec** uses the **password** token, if present.

password *string* — Supply a password. If this token is present, the autologin process supplies the specified string if the remote server requires a password as part of the login process. Note that if this token is present in the **.netrc** file for any user other than **anonymous**, **ftp** aborts the autologin process if the **.netrc** is readable by anyone other than the owner. Also note that the passwords in **.netrc** are not encrypted.

account *string* — Supply an additional account password for **ftp** login. If this token is present, the autologin process supplies the specified string if the remote server requires an additional account password, or the autologin process initiates an **acct** command if it does not.

macdef *name* — Define an **ftp** macro. This token is just like the **ftp macdef** command. A macro is defined with the specified name; its contents begin with the next **.netrc** line and continue until an empty line (consecutive newline characters) is encountered. If a macro named **init** is defined, it is automatically executed as the last step in the **ftp** autologin process.

EXAMPLES

The following is a valid entry for the host **hpxdzg** whose **guest** account has the password **sesame**:

```
machine hpxdzg login guest password sesame
```

WARNINGS

It is a security risk to have unencrypted passwords in a file.

AUTHOR

netrc was developed by the University of California, Berkeley.

n

FILES

$HOME/.netrc

SEE ALSO

ftp(1), remsh(1), rexec(3N).

NAME

nettlgen.conf - network tracing and logging configuration file

SYNOPSIS

`/etc/nettlgen.conf`

DESCRIPTION

/etc/nettlgen.conf,**gpr** the configuration file for Common Network Tracing and Logging commands, contains configuration information used by the **nettl** and **netfmt** commands (see *nettl*(1M) and *netfmt*(1M)). The **nettlconf** command (see *nettlconf*(1M)) maintains log and subsystem data in this file, allowing subsystems to safely add, modify, or delete existing entries in the file. **nettlconf** also allows system administrators to customize logging resource usage parameters and file names in the file. Changes to this file should only be made using the **nettlconf** command.

The file is composed of records containing fields which are separated by colons (**:**). Each line is a unique record containing either global log information or subsystem information. The first field in each record is the tag field which identifies the type of information contained in that record. A **LOG** tag identifies log information; a **SS** tag identifies subsystem information. Blank lines or lines beginning with **#** are ignored.

Log Record

The log record defines static information used to configure logging defaults such as the name of the log file and whether to turn console logging on or off. Note that only the last log record encountered in the file is used; prior log records are ignored. Users can alter the log information to suit their particular needs using the **nettlconf** command. For the log information changes to take effect, the system administrator must stop and restart the tracing and logging facility using the **nettl** command.

Log record fields are as follows:

Field Number	Name	Description
1	tag	Contains **LOG** tag string.
2	Console Logging Flag	Set to 1 if console logging is to be enabled, 0 if not.
3	Log Port Size	Amount of memory to reserve for internal log message buffers. Specified in Kbyte units. Valid range is 1 - 32. The default is 8.
4	Maximum Log File Space	Determines the maximum logging file space to be allowed. Specified in Kbyte units. This value is the combined size of the 2 ping-ponged log files. Valid range is 1 - 10240. The default is 1000.
5	Log File prefix	Path and name of the log file, without the type and age extension (.LOG0x, where x is 0 or 1).
6	Console Filter File	Name of filter configuration file used for console logging.

n

The **Console Logging Flag** determines if console logging is to be enabled when the tracing and logging facility is started. Console logging is used to display log messages on the system console using criteria specified in the file named by **Console Filter File**. If there is no console present or console logging is not desired this feature can be turned off using the **nettlconf** command. During system bootup, the **Console Logging Flag** is always updated to reflect the value of the *NETTL_CONSOLE* variable in the **/etc/rc.config.d/nettl** file.

If more information is desired than the special terse form used for console logging, turn off console logging and start a formatter with an options file specifying the filters to use (see *netfmt*(1M)).

The **Log Port Size** defines the number of outstanding messages possible in the log queue. For logging, 256-byte buffers are used. The number chosen here indicates how much space to allocate in kilobytes. The default size is 8192 bytes (specified by 8), which is split into thirty-two 256-byte blocks. The first block is reserved by the system, leaving 31 blocks for log messages. Each log message starts on a new block, taking

64 bytes of overhead. In addition, each block takes 8 bytes of overhead. The largest message that can be stored using the default size is 7624 bytes ((31 ∗ 256) - (31 ∗ 8) - 64). Most log messages are fairly small, so choosing 8K of buffer is sufficient for the logging facility to keep up with a large volume of messages.

The **Maximum Log File Space** determines the maximum logging file space to be allowed. Log files are split into two parts. When an individual log file reaches one-half of the maximum specified here, the logging system deletes any existing old file, renames the current file to the old file, and starts a new file. The default specification allows for 1 Megabyte of total log file storage (each file does not exceed 500K bytes). Since logging is usually infrequent and log messages are fairly small, this should be more than adequate for all needs. The rate at which the file space fills up depends on what level of logging is turned on for each subsystem, the volume of traffic, frequency of connections, etc; and is very difficult to predict.

The **Console Filter File** specifies the name of the file containing formatter filters used for console logging. This file contains filters that control the logged information displayed on the console. The syntax of this file is the same as the filter configuration files that are used with the **netfmt** command. See *netfmt*(1M) for more details on filter configuration files.

If the console filter file does not exist, the specified file is created with a default set of filters which will display DISASTER messages on the console. If the console filter file does exist and contains a *time_from* filter, the *time_of_day* and *day_of_year* fields in the filter will be updated every time **nettl** is started.

The **Console Filter File** field is optional. If omitted the default file **/var/adm/conslog.opts** will be used.

Subsystem Record

The subsystem record defines the information for that subsystem, and has ten fields including the tag field. The fields are separated by colons (**:**); thus no field can contain a colon. An empty field can be represented by the string **NULL**. NOTE: the information in the subsystem records should only be changed by the subsystem using the **nettlconf** command during product installation. Users should not change this information unless directed by a Hewlett-Packard support representative.

Subsystem record fields are as follows:

Field Number	Name	Description
1	tag	Contains **SS** tag string.
2	Subsystem ID	An integer between 0 and 255. This number is set by the HP factory and must not be changed.
3	Subsystem Mnemonic	A text string consisting of letters, numbers, and the underscore character. The string is set at the factory and must not be changed.
4	Initial Log Class	Logging class for the subsystem when the tracing and logging facility is initialized. This is a numeric value as shown below.
5	Subsystem Type	Set to **s** if the subsystem is streams based and exists in the kernel, **k** if the subsystem exists in the kernel and non-streams based, **u** if not.
6	Subformatter Shared Library	Name of the shared library file containing the subformatter functions listed below.
7	Subformatter Message Catalog	Basename of the message catalog to use when formatting data for this subsystem.
8	Subformatter Function	C function in the subformatter library to call when formatting data for this subsystem.

n

9	Subformatter Options	C function in the subformatter library to call to get filter options for this subsystem.
10	Group Name	A text string to be used in the header banner line in the formatted output.

The recommended setting for the default logging level is set by the products' configuration scripts. It can be changed by the user if another level of logging is desired on initialization. The available classes are Disaster (8), Error (4), Warning (2), and Informative (1). Classes can be combined by adding the numbers; thus Disaster and Error together become 12. The logging level can also be changed at run time using the **nettl -log** command. Disaster class is always turned on, even if not specified in this configuration file; thus, specifying the value 14 or 6 turns on Disaster, Error and Warning.

If the subformatter library file name does not contain an absolute path, it is assumed to be under **/usr/lib**. The subformatter library *must* be a shared library.

EXTERNAL INFLUENCES

Message catalogs are found in the path determined by the environment variable **NLSPATH**. Default message catalogs are found in **/usr/lib/nls/%L/%N.cat** where the contents of the **LANG** environment variable is substituted for the **%L** field, and the name specified in this parameter is substituted for the **%N** field.

EXAMPLES

The following example shows the default logging information. Console logging is enabled; logging uses 8 Kbytes to hold log messages; the log files are limited to 1000 Kbytes total (500 Kbytes per file); the log files are **/var/adm/nettl.LOG00** and **/var/adm/nettl.LOG01**; and the console logging filter file is **/var/adm/conslog.opts**. Most recent data is always in the **.LOG00** file.

```
#
# LOG INFORMATION
#

LOG:1:8:1000:/var/adm/nettl:/var/adm/conslog.opts
```

The following example turns off console logging, and limits the size of the log file space to 100 Kbytes. Other values are the same as the default.

```
#
# LOG INFORMATION
#

LOG:0:8:100:/var/adm/nettl:/var/adm/conslog.opts
```

The following example shows a typical subsystem record. These records should not be changed by the user, but are set by the subsystems using **nettlconf** during product installation.

```
#
# TEST SUBSYSTEMS
#

SS:96:TEST_ID_1:8:u:NULL:netfmt:subsys_GENERIC_format: \
ss_96_go:FORMATTER
SS:97:TEST_ID_2:8:u:NULL:netfmt:subsys_GENERIC_format: \
ss_97_go:FORMATTER
```

Note: The continuation marks in this example (**** at end-of-line) and the following one are placed for readability purposes only. **nettl** and **netfmt** do not understand continuation marks.

The following entry *must always* be included in the configuration file. This defines the subsystem for the formatter itself; if it is not in the file, the formatter will not operate properly.

```
#
# FORMATTER SUBSYSTEMS
#

SS:127:FORMATTER:12:u:NULL:netfmt:subsys_GENERIC_format: \
subsys_127_get_options:FORMATTER
```

FILES

/etc/nettlgen.conf

SEE ALSO

netfmt(1M), nettl(1M), nettlconf(1M).

n

NAME

networks - network name data base

DESCRIPTION

The **/etc/networks** file associates Internet (IP) addresses with official network names and aliases. This allows the user to refer to a network by a symbolic name instead of using an Internet address. For each network, a single line should be present with the following information:

<official network name> *<network number>* *<aliases>*

Aliases are other names under which a network is known. For example:

```
loop   192.46.4   testlan
```

where the network named **loop** is also called **testlan**.

A line cannot start with a blank (tab or space character). Items are separated by any number or combination of blanks. A **#** character indicates the beginning of a comment. Characters from the **#** up to the end of the line are not interpreted by routines which search the file. Trailing blanks are allowed at the end of a line. For the Internet, this file is normally created from the official network database maintained at the Network Information Control Center (NIC), though local changes may be required to bring it up-to-date regarding unofficial aliases and/or unknown networks.

Network numbers can be specified in conventional Internet dot notation using the **inet_network()** routine from the internet address manipulation library (see *inet*(3N). Network names can contain any printable character other than a white space, new-line, or comment character.

EXAMPLES

See **/etc/networks**.

AUTHOR

networks was developed by the University of California, Berkeley.

FILES

/etc/networks

SEE ALSO

getnetent(3N).

NAME

nisfiles - NIS+ database files and directory structure

SYNOPSIS

/var/nis

DESCRIPTION

The Network Information Service Plus (NIS+) uses a memory-based, replicated database. This database uses a set of files in the **/var/nis** directory for checkpointing to stable storage and for maintaining a transaction log. Additionally, the NIS+ server and client use files in this directory to store binding and state information.

The NIS+ service implements an authentication and authorization system that is built upon Secure RPC. In this implementation, the service uses a table named **cred.org_dir.***domain-name* to store the public and private keys of principals that are authorized to access the NIS+ namespace. It stores group access information in the subdomain **groups_dir.** *domain-name* as *group* objects. These two tables appear as files in the **/var/nis/** *hostname* directory on the NIS+ server.

Unlike the previous versions of the network information service in NIS+, the information in the tables is initially loaded into the service from the ASCII files on the server and then updated using NIS+ utilities (**nistbladm -D**). Some sites may wish to periodically regenerate the ASCII files for archival purposes. To do this, a script should be added in the *crontab*(1) of the server that lists these tables and creates the ASCII file from the result.

Note: Except for the **NIS_COLDSTART** and **NIS_SHARED_DIRCACHE** file, no other files should be manipulated by commands such as *cp*(1), *mv*(1) or *rm*(1). The transaction log file keeps logs of all changes made, and hence the files cannot be manipulated independently.

The files described below are stored in the **/var/nis** directory:

NIS_COLDSTART
This file contains NIS+ directory objects that are to be preloaded into the NIS+ cache at startup time. This file is usually created at NIS+ installation time. See *nisinit*(1M) or *nisclient*(1M).

NIS_SHARED_DIRCACHE
This file contains the current cache of NIS+ bindings being maintained by the cache manager. The contents can be viewed with *nisshowcache*(1M).

*hostname***.log**
This file contains a transaction log that is maintained by the NIS+ service. It can be viewed using the *nislog*(1M) command. This file contains holes. Its apparent size may be a lot higher than its actual size. There is only one transaction log per server.

*hostname***.dict**
This file is a dictionary that is used by the NIS+ database to locate its files. It is created by the default NIS+ database package.

*hostname***.dict.log**
This is the log file for the database dictionary. When the server is checkpointed (**nisping -C**), this file will be deleted.

hostname This directory contains databases that the server uses.

*hostname***/root.object**
On root servers, this file contains a directory object that describes the root of the name space.

*hostname***/parent.object**
On root servers, this file contains a directory object that describes the parent namespace. This file is created by the *nisinit*(1M) command.

*hostname***/***table_name*
For each table in the directory there will be a file with the same name that stores the information about that table. If there are subdirectories within this directory, the database for the table is stored in the file *table_name***.***subdirectory*.

*hostname***/***table_name.log*
This file contains the database log for the table *table_name*. The log file maintains the state of individual transactions to each database. When a database has been checkpointed (that is, all changes have been made to the *hostname*/*table_name* stable storage), this log file will be deleted.

n

Currently, NIS+ does not automatically do checkpointing. The system administrator may want to do **nisping** -C (see *nisping*(1M)) operations periodically (such as, once a day) to checkpoint the log file. This can be done either through a *cron*(1M) job, or manually.

hostname/**root_dir**
: On root servers, this file stores the database associated with the root directory. It is similar to other table databases. The corresponding log file is called **root_dir.log**.

hostname/**cred.org_dir**
: This table contains the credentials of principals in this NIS+ domain.

hostname/**groups_dir**
: This table contains the group authorization objects needed by NIS+ to authorize group access.

hostname/**serving_list**
: This file contains a list of all NIS+ directories that are being served by the NIS+ server on this server. When this server is added or deleted from any NIS+ directory object, this file is updated by the server.

AUTHOR

nisfiles was developed by Sun Microsystems, Inc.

SEE ALSO

cp(1), crontab(1), mv(1), rm(1), nis+(1), niscat(1), nismatch(1), nistbladm(1), nisclient(1M), nisinit(1M), nislog(1M), nisping(1M), nis_db(3N), nis_objects(3N).

NAME

nlist, nlist64 - nlist and nlist64 structure formats, respectively

SYNOPSIS

#include <nlist.h>

Remarks

The exact content of the structures defined below can be best found by examining **/usr/include/nlist.h**. It varies somewhat between various HP-UX implementations.

DESCRIPTION

nlist() and **nlist64()** can be used to extract information from the symbol table in an object file (see *nlist*(3C)). They are basically the same tool except **nlist()** can only process SOM files on a PA32 system while **nlist64()** can process SOM and Elf files on either a PA32 or PA64 system. Since symbol tables are machine dependent (as defined in each implementation's copy of <**a.out.h**>), a header file, **nlist.h** is defined to encapsulate the differences.

The nlist function, either **nlist()** or **nlist64()**, when used with the corresponding nlist structure, can be used to extract certain information about selected symbols in the symbol table. The data associated with each symbol is machine specific, thus only the name and position of the **n_name** field in the function is standardized by HP-UX. The rest of the structure includes at least the value and type of the symbol. The names and meanings of all fields not standardized will change no more than necessary.

```
struct nlist {
    char    *n_name;
    /* other fields as needed;
       the following are suggested if they apply */
    char           *n_qual;
    unsigned short  n_type;
    unsigned short  n_scope;
    unsigned int    n_info;
    unsigned long   n_value;
};
struct nlist64 {
    char    *n_name;
    /* other fields as needed;
       the following are suggested if they apply */
    char                *n_qual;
    unsigned short      n_type;
    unsigned short      n_scope;
    unsigned long       n_info;
    unsigned long long  n_value;
    unsigned int        is_elf:1;
    unsigned int        is_32:1;
    unsigned int        reserved1:30;
    unsigned long long  reserved2;
    unsigned long long  reserved3;
};
```

SEE ALSO

nlist(3C), a.out(4).

n

NAME

nsswitch.conf - configuration file for the name-service switch

SYNOPSIS

/etc/nsswitch.conf

DESCRIPTION

The operating system uses a number of "databases" of information about hosts, users (**passwd**), groups and so forth. Data for these can come from a variety of sources: host-names and -addresses, for example, may be found in **/etc/hosts**, NIS, NIS+ or DNS. One or more sources may be used for each database; the sources and their lookup order are specified in the **/etc/nsswitch.conf** file.

The following databases use the switch:

Database	*Used by*
aliases	**sendmail**
automount	**automount**
group	**getgrnam()**
hosts	**gethostbyname()**
netgroup	**innetgr()**
networks	**getnetbyname()**
passwd	**getpwnam()**, **getspnam()**
protocols	**getprotobyname()**
publickey	**getpublickey()**, **secure_rpc()**
rpc	**getrpcbyname()**
sendmailvars	**sendmail**
services	**getservbyname()**

The following sources may be used:

Source	*Uses*
files	**/etc/hosts**, **/etc/passwd**, and so forth
nis	NIS (YP)
nisplus	NIS+
dns	Valid only for **hosts**; uses the Internet Domain Name Service.
compat	Valid only for **passwd** and **group**; implements "+" and "-". (See "Interaction with +/- syntax" below)

There is an entry in **/etc/nsswitch.conf** for each database. Typically these entries will be simple, like "protocols: files" or "networks: files nisplus". However, when multiple sources are specified it is sometimes necessary to define precisely the circumstances under which each source will be tried. A source can return one of the following codes:

Status	*Meaning*
SUCCESS	Requested database entry was found
UNAVAIL	Source is not responding or corrupted
NOTFOUND	Source responded "no such entry"
TRYAGAIN	Source is busy, might respond to retries

For each status code, two actions are possible:

Action	*Meaning*
continue	Try the next source in the list
return	Return now

The complete syntax of an entry is

```
<entry>     ::= <database> ":" [<source> [<criteria>]]* <source>
<criteria>  ::= "[" <criterion>+ "]"
<criterion> ::= <status> "=" <action>
<status>    ::= "success" | "notfound" | "unavail" | "tryagain"
<action>    ::= "return"  | "continue"
```

Each entry occupies a single line in the file. Lines that are blank, or that start with white space character are ignored. Everything on a line following a **#** character is also ignored; the **#** character can begin anywhere in a line, to be used to begin comments. The <database> and <source> names are case-sensitive, but <action> and <status> names are case-insensitive.

The library functions contain compiled-in default entries that are used if the appropriate entry in nsswitch.conf is absent or syntactically incorrect.

The default criteria are to continue on anything except SUCCESS; in other words, [SUCCESS=return NOTFOUND=continue UNAVAIL=continue TRYAGAIN=continue].

The default, or explicitly specified, criteria are meaningless following the last source in an entry; and are ignored since the action is always to return to the caller irrespective of the status code the source returns.

Interaction with netconfig

In order to ensure that they all return consistent results based on the **inet** family of entries, **gethostbyname()**, **getservbyname()**, and **netdir_getbyname()** functions are all implemented in terms of the same internal switch library functions. These functions obtain the system-wide source lookup policy for **hosts** and **services** based on the **inet** family entries in **netconfig()**. For **services** and **hosts** only the "-" in the last column, which represents nametoaddr libraries, is supported.

Interaction with NIS+ YP-compatibility Mode

The NIS+ server can be run in "YP-compatibility mode", where it handles NIS (YP) requests as well as NIS+ requests. In this case, the clients get much the same results from the "nis" source as from "nisplus"; however, "nisplus" is recommended instead of "nis".

Interaction with NIS (YP) server in DNS-forwarding Mode

The NIS (YP) server can be run in "DNS-forwarding mode", where it forwards lookup requests to DNS for host-names and -addresses that do not exist in its database. In this case, specifying "nis" as a source for "hosts" is sufficient to get DNS lookups; "dns" need not be specified explicitly as a source.

The NIS+ server in "YP-compatibility mode" can also be run in "DNS-forwarding mode" (see *rpc.nisd*(1M)). Forwarding is effective only for requests originating from its YP clients; "hosts" policy on these clients should be configured appropriately.

Interaction with +/- syntax

Releases prior to HP-UX 10.30 did not have the name-service switch support for passwd and group but did allow the user some policy control. In **/etc/passwd** one could have entries of the form +*user* (include the specified user from NIS passwd.byname), -*user* (exclude the specified user) and + (include everything, except excluded users, from NIS passwd.byname). The desired behavior was often "everything in the file followed by everything in NIS", expressed by a solitary + at the end of **/etc/passwd**. The switch provides an alternative for this case ("passwd: files nis") that does not require + entries in **/etc/passwd**

If this is not sufficient, the "compat" source provides full +/- semantics. It reads **/etc/passwd** for **getpwnam()** functions and, if it finds +/- entries, invokes an appropriate source. By default the source is "nis", but this may be overridden by specifying "nisplus" as the source for the pseudo-database **passwd_compat**.

The **compat** source also provides full +/- semantics for **group**; the relevant pseudo-database is **group_compat**.

Useful Configurations

The compiled-in default entries for all databases use NIS (YP) as the enterprise level name-service and are identical to those in the default configuration of this file:

passwd:	files nis
group:	files nis
hosts:	nis [NOTFOUND=return] files
networks:	nis [NOTFOUND=return] files
protocols:	nis [NOTFOUND=return] files
rpc:	nis [NOTFOUND=return] files
publickey:	nis [NOTFOUND=return] files
netgroup:	nis
automount:	files nis
aliases:	files nis
services:	files nis
sendmailvars:	files

The policy "nis [NOTFOUND=return] files" implies "if **nis** is UNAVAIL, continue on to **files**, and if **nis** returns NOTFOUND, return to the caller; in other words, treat **nis** as the authoritative source of information and try **files** only if **nis** is down."

n

If compatibility with the +/- syntax for passwd and group is required, simply modify the entries for **passwd** and **group** to:

```
passwd:        compat
group:         compat
```

If NIS+ is the enterprise level name-service, the default configuration should be modified to use **nisplus** instead of **nis** for every database on client machines. The file **/etc/nsswitch.nisplus** contains a sample configuration that can be copied to **/etc/nsswitch.conf** to set this policy.

If the use of +/- syntax is desired in conjunction with **nisplus**, use the following four entries:

```
passwd:        compat
passwd_compat: nisplus
group:         compat
group_compat:  nisplus
```

In order to get information from the Internet Domain Name Service for hosts that are not listed in the enterprise level name-service, NIS+, use the following configuration and set up the **/etc/resolv.conf** file (see *resolver*(4) for more details):

```
hosts:         nisplus dns [NOTFOUND=return] files
```

Enumeration -- getXXXent()

Many of the databases have enumeration functions: **passwd** has **getpwent()**, **hosts** has **gethostent()**, and so on. These were reasonable when the only source was **files** but often make little sense for hierarchically structured sources that contain large numbers of entries, much less for multiple sources. The interfaces are still provided and the implementations strive to provide reasonable results, but the data returned may be incomplete (enumeration for **hosts** is simply not supported by the **dns** source), inconsistent (if multiple sources are used), formatted in an unexpected fashion (for a host with a canonical name and three aliases, the **nisplus** source will return four hostents, and they may not be consecutive), or very expensive (enumerating a **passwd** database of 5000 users is probably a bad idea). Furthermore, multiple threads in the same process using the same reentrant enumeration function (**get***XXX***ent_r()** are supported) share the same enumeration position; if they interleave calls, they will enumerate disjoint subsets of the same database.

n

In general the use of the enumeration functions is deprecated. In the case of **passwd**, and **group**, it may sometimes be appropriate to use **fgetgrent()**, **fgetpwent()**, and **fgetspent()** (see *getgrent*(3C), and *getpwent*(3C), respectively), which use only the **files** source.

WARNINGS

Within each process that uses **nsswitch.conf()**, the entire file is read only once. If the file is later changed, the process will continue using the old configuration.

Programs that use the **get***XX***by***YY***()** functions cannot be linked statically since the implementation of these functions requires dynamic linker functionality to access the shared objects **/usr/lib/nss_SSS.sl.1** at run time.

The use of both **nis** and **nisplus** as sources for the same database is strongly discouraged since both the name-services are expected to store similar information and the lookups on the database may yield different results depending on which name-service is operational at the time of the request.

Misspelled names of sources and databases will be treated as legitimate names of (most likely nonexistent) sources and databases.

The following functions do *not* use the switch: **fgetgrent()**, **fgetpwent()**, **fgetspent()**, **getpw()**, and **putpwent()**.

Applications linked with libc.1 will display different default actions for **NOTFOUND** and **TRYAGAIN**. Applications linked with libc.1 will have the switch search terminate if the Name Service returns a result of **NOTFOUND** or **TRYAGAIN**.

This will be an issue for exisiting nsswitch.conf files that specify name service lookup criteria that contains no <criterion> between <source> entries.

Example: hosts: dns files

For applications linked with libc.1, the fallback to files will only occur if **DNS** returns **UNAVAIL**. For all other applications, the fallback to files will occur unless **DNS** returns **SUCCESS**.

For applications linked with libc.1 and other applications to have the same behavior, a <criterion> must be specified between <source>.

For libc.1 behavior:

hosts: dns [NOTFOUND=return TRYAGAIN=return] files

For the default system behavior:

hosts: dns [NOTFOUND=continue TRYAGAIN=continue] files

AUTHOR

nsswitch.conf was developed by Sun Microsystems, Inc.

FILES

A source named SSS is implemented by a shared object named nss_SSS.1 that resides in **/usr/lib**.

/etc/nsswitch.conf	configuration file
/usr/lib/nss_compat.1	implements "compat" source
/usr/lib/nss_dns.1	implements "dns" source
/usr/lib/nss_files.1	implements "files" source
/usr/lib/nss_nis.1	implements "nis" source
/usr/lib/nss_nisplus.1	implements "nisplus" source
/etc/netconfig	configuration file for **netdir()** functions that redirects hosts/services policy to the switch
/etc/nsswitch.files	sample configuration file that uses "files" only
/etc/nsswitch.nis	sample configuration file that uses "files" and "nis"
/etc/nsswitch.nisplus	sample configuration file that uses "files" and "nisplus"

SEE ALSO

nis+(1), automount(1M), rpc.nisd(1M), sendmail(1M), getgrent(3C), getpwent(3C), gethostent(3N), getnetent(3N), getnetgrent(3C), getprotoent(3N), getpublickey(3N), getrpcent(3C), getservent(3N), netdir(3N), secure_rpc(3N), netconfig(4), resolver(4), ypfiles(4).

NAME

pam.conf - configuration file for pluggable authentication modules

SYNOPSIS

/etc/pam.conf

DESCRIPTION

pam.conf is the configuration file for the Pluggable Authentication Module architecture, or PAM. A PAM module provides functionality for one or more of four possible services: *authentication, account management, session management,* and *password management*.

An authentication service module provides functionality to authenticate a user and set up user credentials. A account management module provides functionality to determine if the current user's account is valid. This includes checking for password and account expiration, as well as verifying access hour restrictions. A session management module provides functionality to set up and terminate login sessions. A password management module provides functionality to change a user's authentication token or password.

Simplified PAM.CONF configuration file

The **pam.conf** file contains a listing of services. Each service is paired with a corresponding service module. When a service is requested, its associated module is invoked. Each entry has the following format:

service_name module_type control_flag module_path options

Below is an example of the **pam.conf** configuration file with support for authentication, account management, and session management modules.

```
login     auth     required   /usr/lib/security/libpam_unix.1  debug
login     session  required   /usr/lib/security/libpam_unix.1
login     account  required   /usr/lib/security/libpam_unix.1
dtlogin   session  required   /usr/lib/security/libpam_unix.1
other     auth     required   /usr/lib/security/libpam_unix.1
other     password required   /usr/lib/security/libpam_unix.1
```

service_name The *service_name* denotes the service (for example, **login**, or **dtlogin**). The keyword, **other**, indicates the module all other applications which have not been specified should use. The **other** keyword can also be used if all services of the same *module_type* have the same requirements. In the example above, since all of the services use the same session module, they could have been replaced by a single **other** line.

p

module_type *module_type* denotes the service module type: authentication (*auth*), account management (*account*), session management (*session*), or password management (*password*).

control_flag The *control_flag* field determines the behavior of stacking, and will be discussed in more detail below.

module_path The *module_path* field specifies the pathname to a shared library object which implements the service functionality. If the pathname is not absolute, it is assumed to be relative to **/usr/lib/security**.

options The *options* field is used by the PAM framework layer to pass module specific options to the modules. It is up to the module to parse and interpret the options. This field can be used by the modules to turn on debugging or to pass any module specific parameters such as a TIMEOUT value. It can also be used to support unified login. The options supported by the modules are documented in their respective manual pages. For example, *pam_unix*(5) lists the options accepted by the UNIX module.

Integrating Multiple Authentication Services With Stacking

When a service_name of the same *module_type* is defined more than once, the service is said to be *stacked*. Each module referenced in the *module_path* for that service is then processed in the order that it occurs in the configuration file. The *control_flag* field specifies the continuation and failure semantics of the modules, and may be **required**, **optional**, or **sufficient**.

The PAM framework processes each service module in the stack. If all **required** modules in the stack succeed, then success is returned (**optional** and **sufficient** error values are ignored). If one or more **required** modules fail, then the error value from the first **required** module that failed is returned.

If none of the service modules in the stack are designated as **required**, then the PAM framework requires that at least one **optional** or **sufficient** module succeed. If all fail then the error value from the first service module in the stack is returned.

The only exception to the above is caused by the **sufficient** flag. If a service module that is designated as **sufficient** succeeds, then the PAM framework immediately returns success to the application (all subsequent services modules, even **required** ones, in the stack are ignored), given that all prior **required** modules had also succeeded. If a prior **required** module failed, then the error value from that module is returned.

If a module does not exist or can not be opened, then the **pam.conf** entry is ignored and an error will be logged through *syslog*(3C) at the LOG_CRIT level.

Below is a sample configuration file that stacks the **login**, and **dtlogin** services.

```
login    auth    required      /usr/lib/security/libpam_unix.1. debug
login    auth    optional      /usr/lib/security/libpam_inhouse.1
dtlogin  auth    sufficient    /usr/lib/security/libpam_unix.1  debug
dtlogin  auth    required      /usr/lib/security/libpam_inhouse.1
```

In the case of **login**, the user is authenticated by the UNIX and inhouse authentication modules. The **required** keyword for *control_flag* requires that the user be allowed to login only if the user is authenticated by the UNIX service module. Inhouse authentication is optional by virtue of the **optional** keyword in the *control_flag* field. The user can still log in even if inhouse authentication fails.

In the case of **dtlogin**, the **sufficient** keyword for *control_flag* specifies that if the UNIX authentication check succeeds, then PAM should return success to **dtlogin**. The inhouse authentication module (the next module in the stack) will only be invoked if the UNIX authentication check fails.

Some modules may return PAM_IGNORE in certain situations. In these cases the PAM framework ignores the entire entry in **pam.conf** regardless of whether or not it is **required**, **optional** or **sufficient**.

Configuration Per User

pam.conf contains information to configure all the users on a system. But sometimes it is necessary to configure user by user. A user policy definition is made through a specific module named **libpam_updbe.1**. This module reads a file named **/etc/pam_user.conf** which describes the user's configurations.

Below is a sample configuration file (**/etc/pam.conf**) that uses the module **libpam_updbe.1**.

```
login    auth     required     /usr/lib/security/libpam_updbe.1
login    auth     required     /usr/lib/security/libpam_unix.1
su       auth     required     /usr/lib/security/libpam_updbe.1
su       auth     required     /usr/lib/security/libpam_unix.1
OTHER    auth     required     /usr/lib/security/libpam_unix.1

login    password required     /usr/lib/security/libpam_updbe.1
login    password required     /usr/lib/security/libpam_unix.1
passwd   password required     /usr/lib/security/libpam_updbe.1
passwd   password required     /usr/lib/security/libpam_unix.1
OTHER    password required     /usr/lib/security/libpam_unix.1
```

The module **libpam_updbe.1** searches the configuration file **/etc/pam_user.conf** and reads the configuration associated with the login name of the current user. If there is no configuration concerning the current user in the **pam_user.conf** file, the PAM framework ignores the line containing **libpam_updbe.1**. The pam.conf applies for those users who are not configured in **pam_user.conf**.

NOTES

If an error is found in an entry due to invalid *service_name*, *module_type*, or *control_flag*, then the entry is ignored. If there are no valid entries for the given *module_type*, the PAM framework returns an error to the application.

EXAMPLES

The following is a sample **pam.conf** configuration file. Lines that begin with the **#** symbol are treated as comments, and therefore ignored.

```
#
# PAM configuration
#
# Authentication management for login service is stacked.
# Both UNIX and inhouse authentication functions are invoked.
login   auth   required   /usr/lib/security/libpam_unix.1
login   auth   required   /usr/lib/security/libpam_inhouse.1  try_first_pass
dtlogin auth   required   /usr/lib/security/libpam_unix.1
dtlogin auth   required   /usr/lib/security/libpam_inhouse.1  try_first_pass
#
# Other services use UNIX authentication
other   auth     required     /usr/lib/security/libpam_unix.1
#
# Account management for login service is stacked.
# UNIX account management is required; inhouse account management is optional
login   account  required     /usr/lib/security/libpam_unix.1
login   account  optional     /usr/lib/security/libpam_inhouse.1
dtlogin account  required     /usr/lib/security/libpam_unix.1
dtlogin account  optional     /usr/lib/security/libpam_inhouse.1
other   account  required     /usr/lib/security/libpam_unix.1
#
# Session management
other   session  required     /usr/lib/security/libpam_unix.1
#
# Password management
other   password required     /usr/lib/security/libpam_unix.1
```

The following is a sample **pam.conf** configuration which uses the **libpam_updbe.1** module to configure a user. Lines that begin with the **#** symbol are treated as comments, and therefore ignored.

```
#
# PAM configuration
#
# Authentication management for login service is stacked.
# Both UNIX and inhouse authentication functions are invoked.
login   auth   required   /usr/lib/security/libpam_updbe.1
login   auth   required   /usr/lib/security/libpam_unix.1
login   auth   required   /usr/lib/security/libpam_inhouse.1  try_first_pass
dtlogin auth   required   /usr/lib/security/libpam_updbe.1
dtlogin auth   required   /usr/lib/security/libpam_unix.1
dtlogin auth   required   /usr/lib/security/libpam_inhouse.1  try_first_pass
#
# Other services use UNIX authentication
other   auth     required     /usr/lib/security/pam_unix.so.1
#
# Account management for login service is stacked.
# UNIX account management is required; inhouse account management is optional
login   account  required     /usr/lib/security/libpam_unix.1
login   account  optional     /usr/lib/security/libpam_inhouse.1
dtlogin account  required     /usr/lib/security/libpam_unix.1
dtlogin account  optional     /usr/lib/security/libpam_inhouse.1
other   account  required     /usr/lib/security/libpam_unix.1
#
# Session management
other   session  required     /usr/lib/security/libpam_unix.1
#
# Password management
passwd  password required     /usr/lib/security/libpam_updbe.1
passwd  password required     /usr/lib/security/libpam_unix.1
other   password required     /usr/lib/security/libpam_unix.1
```

Utilities and Files

A list of utilities that are known to use PAM include: **login**, **passwd**, **su**, and **dtlogin**.

The PAM configuration file does not dictate either the name or the location of the service specific modules. The convention, however, is the following:

/usr/lib/security/libpam_*service_name***.x**
Implements various function of specific authentication services.

/etc/pam.conf
Configuration file.

/usr/lib/libpam.1
Implements the PAM framework library.

SEE ALSO

dtlogin(1), login(1), passwd(1), su(1), pam(3).

p

NAME

pam_user.conf - users configuration file for pluggable authentication modules

SYNOPSIS

/etc/pam_user.conf

DESCRIPTION

pam_user.conf is the user configuration file for the Pluggable Authentication Module architecture, or PAM. It is not designed to replace the PAM system configuration file, **pam.conf**. For PAM to work properly, **pam.conf** is mandatory (see *pam.conf*(4)). **pam_user.conf** is optional. It is used only when a user basis configuration is needed. It mainly specifies *options* to be used by service modules on a user basis.

The *options* defined in **pam.conf** indicate the default for users who are not configured in **pam_user.conf** or if the module type is not configured for some users. For the configuration in **pam_user.conf** to take effect, **pam.conf** needs to configure service module **libpam_updbe** (see *pam.conf*(4)).

Simplified PAM_USER.CONF Configuration File

The **pam_user.conf** file contains a listing of login names. Each login name is paired with a corresponding service module with or without options specified. Each entry has the following format:

login_name module_type module_path options

Below is an example of the **pam_user.conf** configuration file.

```
tom     auth     /usr/lib/security/libpam_unix.1   debug use_psd
tom     auth     /usr/lib/security/libpam_dce.1    use_first_pass
tom     account  /usr/lib/security/libpam_unix.1   use_psd
tom     account  /usr/lib/security/libpam_dce.1    try_first_pass

susan   auth     /usr/lib/security/libpam_unix.1
susan   auth     /usr/lib/security/libpam_dce.1    try_first_pass
```

The *login_name* denotes the login name of a user (for example, tom, susan). For detailed information on *module_type*, *module_path*, and *options*, see *pam.conf*(4).

The first entry indicates that when the UNIX authentication is invoked for *tom*, the *options* "debug" and "use_psd" will be used. The second entry indicates that when the DCE authentication is invoked for *tom* , the *option* "use_first_pass" will be used. The module type "password" is not configured for *tom*, therefore, the **/etc/pam.conf** options will take effect. For those users who are not configured, the **/etc/pam.conf** options apply.

p

NOTES

If an error is found in an entry due to invalid *login_name* or *module_type*, then the entry is ignored. If there are no valid entries for the given *module_type*, the PAM framework ignores **pam_user.conf** and reads the configuration in **pam.conf**.

EXAMPLES

The following is a sample **pam_user.conf** configuration file. Lines that begin with the **#** symbol are treated as comments, and therefore ignored.

```
#
# PAM user configuration
#

# Authentication management
john    auth     /usr/lib/security/libpam_unix.1
john    auth     /usr/lib/security/libpam_inhouse.1  try_first_pass

david   auth     /usr/lib/security/libpam_unix.1     use_psd
david   auth     /usr/lib/security/libpam_inhouse.1  try_first_pass

susan   auth     /usr/lib/security/libpam_unix.1     use_psd
susan   auth     /usr/lib/security/libpam_inhouse.1  try_first_pass

# Password management
john    password /usr/lib/security/libpam_unix.1
david   password /usr/lib/security/libpam_unix.1     use_psd
```

```
susan   password /usr/lib/security/libpam_unix.1     use_psd
```

SEE ALSO

pam(3), pam.conf(4).

p

NAME

passwd - password file, pwd.h

DESCRIPTION

passwd contains the following information for each user:

- login name
- encrypted password
- numerical user ID
- numerical group ID
- reserved field, which can be used for identification
- initial working directory
- program to use as shell

This is an ASCII file. Each field within each user's entry is separated from the next by a colon. Each user is separated from the next by a newline. This file resides in the **/etc** directory. It can and does have general read permission and can be used, for example, to map numerical user IDs to names. If the password field is null and the system has not been converted to a trusted system, no password is demanded.

If the shell field is null, **/usr/bin/sh** is used.

The encrypted password consists of 13 characters chosen from a 64-character set of "digits" described below, except when the password is null, in which case the encrypted password is also null. Login can be prevented by entering in the password field a character that is not part of the set of digits (such as *****).

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

Password aging is put in effect for a particular user if his encrypted password in the password file is followed by a comma and a nonnull string of characters from the above alphabet. (Such a string must be introduced in the first instance by a superuser.) This string defines the "age" needed to implement password aging.

The first character of the age, *M*, denotes the maximum number of weeks for which a password is valid. A user who attempts to login after his password has expired is forced to supply a new one. The next character, *m*, denotes the minimum period in weeks that must expire before the password can be changed. The remaining characters define the week (counted from the beginning of 1970) when the password was last changed (a null string is equivalent to zero). *M* and *m* have numerical values in the range 0 through 63 that correspond to the 64-character set of "digits" shown above. If $m = M = 0$ (derived from the string **.** or **..**), the user is forced to change his password next time he logs in (and the "age" disappears from his entry in the password file). If $m > M$ (signified, for example, by the string **./**), then only a superuser (not the user) can change the password. Not allowing the user to ever change the password is discouraged, especially on a trusted system.

Trusted systems support password aging and password generation. For more information on converting to trusted system and on password, see *Managing Systems and Workgroups* and *sam*(1M).

getpwent(3C) designates values to the fields in the following structure declared in **<pwd.h>**:

```
struct passwd {
    char    *pw_name;
    char    *pw_passwd;
    uid_t   pw_uid;
    gid_t   pw_gid;
    char    *pw_age;
    char    *pw_comment;
    char    *pw_gecos;
    char    *pw_dir;
    char    *pw_shell;
    aid_t   pw_audid;
    int     pw_audflg;
};
```

It is suggested that the range 0–99 not be used for user and group IDs (**pw_uid** and **pw_gid** in the above structure) so that IDs that might be assigned for system software do not conflict.

The user's full name, office location, extension, and home phone stored in the **pw_gecos** field of the **passwd** structure can be set by use of the **chfn** command (see *chfn*(1)) and is used by the *finger*(1) command. These two commands assume the information in this field is in the order listed above. A portion of the user's real name can be represented in the **pw_gecos** field by an **&** character, which some utilities (including **finger**) expand by substituting the login name for it and shifting the first letter of the login name to uppercase.

SECURITY FEATURES

On trusted systems, the encrypted password for each user is stored in the file **/tcb/files/auth/***c***/***user_name* (where *c* is the first letter in *user_name*). Password information files are not accessible to the public. The encrypted password can be longer than 13 characters . For example, the password file for user **david** is stored in **/tcb/files/auth/d/david**. In addition to the password, the user profile in **/tcb/files/auth/***c***/***user_name* also contains:

- numerical audit ID
- numerical audit flag

Like **/etc/passwd**, this file is an ASCII file. Fields within each user's entry are separated by colons. Refer to *authcap*(4) and *prpwd*(4) for details. The passwords contained in **/tcb/files/auth/***c***/*** take precedence over those contained in the encrypted password field of **/etc/passwd**. User authentication is done using the encrypted passwords in this file . The password aging mechanism described in *passwd*(1), under the section called **SECURITY FEATURES**, applies to this password .

NETWORKING FEATURES

NIS

The **passwd** file can have entries that begin with a plus (**+**) or minus (**-**) sign in the first column. Such lines are used to access the Network Information System network database. A line beginning with a plus (**+**) is used to incorporate entries from the Network Information System. There are three styles of **+** entries:

+	Insert the entire contents of the Network Information System password file at that point;
+*name*	Insert the entry (if any) for *name* from the Network Information System at that point
+@*name*	Insert the entries for all members of the network group *name* at that point.

If a **+** entry has a nonnull password, directory, gecos, or shell field, they override what is contained in the Network Information System. The numerical user ID and group ID fields cannot be overridden.

The *passwd* file can also have lines beginning with a minus (**-**), which disallow entries from the Network Information System. There are two styles of **-** entries:

-*name*	Disallow any subsequent entries (if any) for *name*.
-@*name*	Disallow any subsequent entries for all members of the network group *name*.

WARNINGS

User ID (uid) 17 is reserved for the Pascal Language operating system. User ID (uid) 18 is reserved for the BASIC Language operating system. These are operating systems for Series 300 and 400 computers that can coexist with HP-UX on the same disk. Using these uids for other purposes may inhibit file transfer and sharing.

The login shell for the root user (uid 0) must be **/sbin/sh.** Other shells such as sh, ksh, and csh are all located under the **/usr** directory which may not be mounted during earlier stages of the bootup process. Changing the login shell of the root user to a value other than **/sbin/sh** may result in a non-functional system.

The information kept in the **pw_gecos** field may conflict with unsupported or future uses of this field. Use of the **pw_gecos** field for keeping user identification information has not been formalized within any of the industry standards. The current use of this field is derived from its use within the Berkeley Software Distribution. Future standards may define this field for other purposes.

The following fields have character limitations as noted:

- Login name field can be no longer than 8 characters;
- Initial working directory field can be no longer than 63 characters;
- Program field can be no longer than 44 characters.
- Results are unpredictable if these fields are longer than the limits specified above.

p

The following fields have numerical limitations as noted:

- The user ID is an integer value between –2 and **UID_MAX** inclusive.
- The group ID is an integer value between 0 and **UID_MAX** inclusive.
- If either of these values are out of range, the *getpwent*(3C) functions reset the ID value to (**UID_MAX**).

EXAMPLES

NIS Example

Here is a sample **/etc/passwd** file:

```
root:3Km/o4Cyq84Xc:0:10:System Administrator:/:/sbin/sh
joe:r4hRJr4GJ4CqE:100:50:Joe User,Post 4A,12345:/home/joe:/usr/bin/ksh
+john:
-bob:
+@documentation:no-login:
-@marketing:
+:::Guest
```

In this example, there are specific entries for users **root** and **joe**, in case the Network Information System are out of order.

- User **john**'s password entry in the Network Information System is incorporated without change.
- Any subsequent entries for user **bob** are ignored.
- The password field for anyone in the netgroup **documentation** is disabled.
- Users in netgroup **marketing** are not returned by *getpwent*(3C) and thus are not allowed to log in.
- Anyone else can log in with their usual password, shell, and home directory, but with a **pw_gecos** field of **Guest**.

NIS Warnings

The plus (**+**) and minus (**-**) features are NIS functionality; therefore, if NIS is not installed, they do not work. Also, these features work only with **/etc/passwd**, but not with a system that has been converted to a trusted system. When the system has been converted to a trusted system, the encrypted passwords can be accessed only from the protected password database, **/tcb/files/auth/*/***. Any user entry in the Network Information System database also must have an entry in the protected password database.

p

The uid of –2 is reserved for remote root access by means of NFS. The **pw_name** usually given to this uid is **nobody**. Since uids are stored as signed values, the following define is included in **<pwd.h>** to match the user **nobody**.

```
UID_NOBODY  (-2)
```

FILES

/tcb/files/auth/*/*	Protected password database used when system is converted to trusted system.
/etc/passwd	Standard password file used by HP-UX.

SEE ALSO

chfn(1), finger(1), login(1), passwd(1), a64l(3C), crypt(3C), getprpwent(3), getpwent(3C), authcap(4), limits(5).

STANDARDS CONFORMANCE

passwd: SVID2, SVID3, XPG2

NAME

pcf - port configuration file used by DDFA software

Description

A port configuration file is used by the Datacommunications and Terminal Controller Device File Access (DDFA) software to configure individual terminal server ports. The generic name of the template file is **pcf**. In practice, it is renamed for each port that needs different configuration values and the values are altered appropriately for the device attached to the port. A port configuration file is referenced by an entry in the Dedicated Ports file (**dp**). The Dedicated Port Parser (**dpp**) parses the **dp** file and spawns an Outbound Connection Daemon (**ocd**) for each valid entry in the **dp** file. A valid entry is one in which the fourth field is the name of a port configuration file.

The master port configuration file is **/usr/examples/ddfa/pcf** and it should only be referenced in the **dp** file if the default values it contains are correct for the ports. If different values are needed, **/usr/examples/ddfa/pcf** should be copied to another directory and the copy should be modified and referenced in the **dp** file. The recommended procedure is to create a directory to hold the port configuration files and the modified **dp** file.

See *ddfa*(7) for more information on how to configure the DDFA software.

A port configuration file consists of the names of variables and their values. The variables are shown terminated by a colon (**:**), but this is not mandatory. A variable and its value can be separated by spaces or tabs. Only one variable-value pair is allowed per line. Only the value should be altered. The variable name should not be changed.

A file contains the following information:

telnet_mode: This can have the value **disable** or **enable**. When it is enabled, data transfer over the network uses the Telnet protocol. This option *must* be enabled for a DTC.

timing_mark: This can have the value **disable** or **enable**. When it is enabled, a telnet timing mark negotiation is sent to the terminal server after all user data has been transferred. **ocd** waits for a reply to the timing mark negotiation before closing the connection. This ensures that all data has been output from the terminal server to the device before the buffers are flushed. It should be enabled for a DTC.

telnet_timer: This defines the time in seconds during which the software waits for a response to the telnet timing mark and binary negotiation. If the timer expires, an error message is logged to **/var/adm/syslog** and the error is transmitted to the user application.

binary_mode: This can have the value **disable** or **enable**. When it is enabled, data transfer over the network is in binary mode and treatment of special characters (such as XON/XOFF) is disabled.

Due to the absence of flow control, data integrity cannot be guaranteed when **binary_mode** is enabled.

Note that even if **binary_mode** is disabled, it can be negotiated at any time by the application setting **IXON** to **0** in the **termio** data structure.

open_tries: This defines the number of times the software tries to open a connection before giving up. If the value is **0** the software tries "forever" (approximately 68 years). If the retry process fails, an error message is logged to **/var/adm/syslog** and the error is transmitted to the user application.

The retry process can be interrupted by sending the **SIGUSR2** signal to the **ocd** process using **kill -17** *pid*.

Note that if the application exits after asking **ocd** to open the connection to the terminal server, **ocd** continues trying to open until the combination of the **open_tries** and **open_timer** are exceeded.

open_timer: This defines the time in seconds between open tries. If the value is **0**, **ocd** uses an exponential retry period algorithm up to 32 seconds (i.e., 1 2 4 8 16 32 32 32 ...).

close_timer: This defines the time in seconds between the close call made by the application on the pty slave and the moment when the connection is actually closed. Setting this value to, for example, 5 seconds avoids the overhead of opening and closing the connection when a spooler spools several files at a time. Setting a sufficiently high value effectively leaves the connection permanently open.

p

status_request:
This can have the value **disable** or **enable**. When it is enabled, the software sends a status request to the device attached to the terminal server and processes the reply as follows:

LP_OK *(0x30)*	**ocd** continues processing.
LP_NO_PAPER *(0x31)*	**ocd** retries within the limits of the status timer.
LP_BUSY *(0x32)*	**ocd** retries within the limits of the status timer.
LP_OFF_LINE *(0x34)*	**ocd** retries within the limits of the status timer.
LP_DATA_ERROR *(0x38)*	**ocd** retries within the limits of the status timer.

status_timer: This defines the time in seconds during which the software waits for the reply to the status request. If the timer expires, an error message is logged to **/var/adm/syslog** and the error is transmitted to the user application.

eight_bit: This can have the value **disable** or **enable**. Normally, data bytes processed by the pty have bit 7 stripped. If **eight_bit** is enabled, the stripping is disabled. If **eight_bit** is disabled, stripping is enabled and bit 7 is stripped. This can also be achieved by changing the termio structure of the pseudonym using ioctl() commands.

tcp_nodelay: This can have the value **disable** or **enable**. When it is enabled, data is sent to the LAN as it is received. It can be disabled if the software is sending packets faster than the server can accept them.

The default values are:

```
telnet_mode      enable
timing_mark      enable
telnet_timer     120
binary_mode      disable
open_tries       1500
open_timer       30
close_timer      5
status_request   disable
status_timer     30
eight_bit        disable
tcp_nodelay      enable
```

p

WARNINGS

In order to ensure that commands (such as *ps*) display the correct device file name (that is, the *pseudonym*), all pseudonyms should be placed into the directory **/dev/telnet**. If pseudonyms are not specified for placement in this directory, the correct display of device file names with many commands is not guaranteed.

In addition, in order to ensure that commands (such as **w**, **passwd**, **finger**, and **wall**) work correctly, each pseudonym must be unique in its first 17 characters (including the directory prefix **/dev/telnet/**). If pseudonyms are not unique in their first 17 characters, the correct functioning of many commands is not guaranteed.

FILES

/usr/sbin/dpp
/usr/sbin/ocd
/usr/sbin/ocdebug
/var/adm/dpp_login.bin
/var/adm/utmp.dfa
/usr/examples/ddfa/dp
/usr/examples/ddfa/pcf

SEE ALSO

dpp(1M), ocd(1M), ocdebug(1M), dp(4), ddfa(7).

NAME

pdf - Product Description File

DESCRIPTION

A **Product Description File** describes product files contained in the HP-UX operating system. It consists of a file containing a single line entry for each file described, where each entry contains the following fields:

pathname
owner
group
mode
size
links
version
checksum
linked_to

Fields are separated by a colon (**:**), and contain the information indicated:

pathname — Absolute pathname of the file (starts with **/**). If pathname is preceded by **?**, it is an optional file that may or may not be present on the system.

owner — Symbolic or numeric ID of the owner of the file.

group — Symbolic or numeric ID of the group of the file.

mode — Symbolic representation of file type and permission information as displayed by the **ls -l** command.

size — Size of the file in bytes. In the case of device special files, it is the major/minor number. Directory sizes are not recorded.

links — Number of hard links to *pathname*.

version — Numeric value of the revision of the file. Commands supporting PDFs determine this value by invoking the **what** command on the file and searching for a revision number (see *what*(1)). If no revision is found, **ident** invoked (see *ident*(1)). The version number recorded is the first one encountered. If no version number is found, the field is empty.

checksum — Result of the application of the Ethernet (and hence IEEE 802.3) CRC checksum algorithm to the file's contents.

linked_to — File to which *pathname* is linked, whether with a **hard** or **symbolic** link. If *pathname* is not a link, this field is empty.

Some commands (namely **pdfdiff** and **pdfck**) rely on the convention that one file in a set of hard links is considered the primary file, indicating no *linked_to* file in the PDF, while the remaining files in the set all indicate the primary file as the *linked_to* (see *pdfdiff*(1M) and *pdfck*(1M)). This convention prevents double counting in size calculations, and allows some efficiencies in algorithms for checking consistency of links.

Empty fields indicate a "don't care" status. Any field except *pathname* can be empty.

comment lines in the file begin with the percent character (**%**). The first line of the file is always the comment:

```
% Product Description File
```

The second comment line is produced by the **mkpdf** command's **-c** option. For HP-UX files, this comment usually indicates the product name and release.

EXAMPLE

Here is an example product description file:

```
% Product Description File
% fileset TEST, Release 1.0
/usr/bin/basename:bin:bin:-r-xr-xr-x:2244:1:66.2:4066520052:
/usr/bin/cat:bin:bin:-r-xr-xr-x:4740:1:66.2:2516588651:
/usr/bin/ccat:bin:bin:-r-xr-xr-x:24576:2:66.12:330130894:
/usr/bin/dirname:bin:bin:-r-xr-xr-x:1936:1:64.3:549465715:
/usr/bin/grep:bin:bin:-r-xr-xr-x:11988:3:66.11:2104745188:
```

p

```
/usr/bin/ls:bin:bin:-r-xr-xr-x:24576:6:66.3:312786007:
/usr/bin/ll:::::6:::/usr/bin/ls
/usr/bin/su:root:bin:-r-sr-xr-x:90112:1:66.2:3088851439:
% total size is 160172 bytes.
% total size is 158 blocks.
```

WARNINGS

The checksum algorithm is different than that used by the 7.0 Release version of the commands.

Use of PDFs is discouraged since this functionality is obsolete and is being replaced with Software Distributor (see *sd*(4)).

AUTHOR

The specification of PDF is derived from an early draft proposal for *Bill of Materials* in *IEEE POSIX P1003.2* (Draft 2). This proposal was later dropped from the standard. The implementation is by HP.

SEE ALSO

mkpdf(1M), pdfdiff(1M), pdfck(1M).

p

NAME

pdgwcfg.conf - HPDPS gateway printer configuration file

DESCRIPTION

When invoked, the **pdgwcfg** utility (see *pdgwcfg*(1M)) reads the configuration information from the **/etc/pdgwcfg.conf** configuration file. It is used to assist in administering the creation and/or deletion of gateway printers in an HPDPS Basic (non-DCE) environment.

/etc/pdgwcfg.conf contains the following configurable values:

GatewayPrinter:[*LocalHostField*],*LocalSpooler*,*RemoteHost*,*RemoteLogicalPrinter*[,'*Attributes*']

GatewayPrinter The name of the gateway printer. Should be a contiguous string of characters with no metacharacters. Can be the same name as the *RemoteLogicalPrinter* for naming consistency across the enterprise.

LocalHostField An optional field to specify on which hosts this gateway printer should exist. It has the format:

LocalHost1 | *LocalHost2* | ...

or

-*LocalHost1* | -*LocalHost2* | ...

If the field is empty, it is assumed this entry applies to all hosts. One can explicitly define the applicable hosts by separating the hosts with the '|' character. One can also explicitly define an exclude list by prepending the hostname with a '-' character.

LocalSpooler The name of the spooler in the local environment to which this gateway printer is to be added. The spooler must be operational at the time the gateway printer is created.

RemoteHost Foreign host where the foreign logical printer exists. If the local host on which this file is being processed matches the *RemoteHost* value, the entry is ignored (the logical printer already exists on this local host).

RemoteLogicalPrinter
Name of the existing foreign logical printer. It must have already been created and be accessible at the time the gateway printer is created.

'*Attributes*' An optional field to allow specification of attributes for the gateway printer. The entire list of attributes should be enclosed by one set of either '' or "" characters. It is taken as an entire string for input into the *pdcreate*(1) command following the -x option.

Do not separate the fields by whitespace. A **#** character in the first column indicates a comment line and will be ignored. Any line with only whitespace will be ignored. A continuation character **** can be used for entries that extend to the next line.

EXAMPLES

Each line in the following example **/etc/pdgwcfg.conf** file is preceded by a comment (beginning with **#**) that explains the entry. Please note that this is an example. Taken as a whole, it is not intended to represent a configuration that you should use. Its sole purpose is to show the flexibility of the configuration file.

```
# /etc/pdgwcfg.conf
# HPDPS Gateway Configuration File used by pdgwcfg(1M)

# Gateway printers named 'mopier' in spoolers 'local_spl'
# will only be created on 'host8' and 'host9'.
# The logical printer 'joesmopier' is located on 'host3'.
# Note:  the gateway printer can have a different name than
# the local printer.
mopier:host8|host9,local_spl,host3,joesmopier

# Gateway printers named 'hplaser' in spoolers 'dps_common_spl'
# will be created on all systems to which this file is propagated
# (since no local hosts are explicitly called out after the colon)
# except 'host2' (since that is where the printer exists).
```

p

```
# The logical printer is 'hplaser' located on 'host2'.
# Note: all systems have a DPS spooler named dps_common_spl running
# Note: the gateway printer can have the same name as the local
# printer for naming consistency across the enterprise.
hplaser:,dps_common_spl,host2,hplaser

# Gateway printers named 'hpdeskjet' in spoolers 'dps_common_spl'
# will be created on all systems to which this file is propagated
# except on host1, host2, and host7 (and host4 since that is
# where the printer exists).
# The logical printer is 'hpdeskjet' located on 'host4'.
# Note: all systems have a DPS spooler named dps_common_spl running
hpdeskjet:-host1|-host2|-host7,dps_common_spl,host4,hpdeskjet

# Gateway printers named 'hpcolorlaser' in spoolers 'dps_common_spl'
# will be created on all systems to which this file is propagated
# except on host5 (and host4 since that is where the printer exists).
# The logical printer is 'hpcolorlaser' located on 'host4'.
# A message attribute is provided for this gateway printer
# Note: all systems have a DPS spooler named dps_common_spl running
hpcolorlaser:\
     -host5,dps_common_spl,host4,hpcolorlaser,\
     "message='For Marketing Dept. use only'"
```

WARNINGS

Once a gateway printer is created, any subsequent modifications to the gateway printer entry in **pdgwcfg.conf** are ignored. If gateway printer attributes must be modified, then use **pdset** (see *pdset*(1)). If this is impractical and the gateway printer can be deleted and recreated without impact, one could temporarily comment out the entry and invoke **pdgwcfg** (causing the gateway printer to be deleted), then uncomment and modify the entry in **pdgwcfg.conf** and invoke **pdgwcfg** (causing the gateway printer to be recreated).

Both the local HPDPS environment and all applicable foreign HPDPS environments must be operating at the time the **pdgwcfg** utility is invoked.

AUTHOR

pdgwcfg.conf was developed by HP.

p

SEE ALSO

pdgwcfg(1M), pdcreate(1), pddelete(1), pdset(1).

HP Distributed Print Service Administration Guide (re: gateway printers)

NAME

pfs, PFS - portable file system

DESCRIPTION

The Portable File System, or PFS, allows access to a variety of CD-ROM file systems. Currently supported file systems include: **iso9660**, **high sierra**, **RockRidge Interchange**.

The PFS package consists of 7 programs:

- **pfs_mountd** is responsible for maintaining local and remote mounts. It must be running on both PFS clients and PFS servers. The **pfs_mountd** program validates arguments, and spawns **pfs_mountd.rpc**.
- **pfs_mountd.rpc** is the RPC server code associated with **pfs_mountd**. It should not be executed directly.
- **pfsd** responds to all client requests for a given mounted CD-ROM file system. **pfsd** needs to be running on all systems designated as PFS servers. **pfsd** validates arguments, and spawns **pfsd.rpc**.
- **pfsd.rpc** is the RPC server code associated with **pfsd**. It should not be executed directly.
- **pfs_exportfs** makes local directories available for mounting by PFS clients.
- **pfs_mount** mounts CD-ROM file system locally or from server.
- **pfs_umount** unmounts CD-ROM file system locally or from server.

Client file access calls are converted to PFS protocol requests, and are sent to the server system over the network. The server receives the request, performs the actual file system operation, and sends a response back to the client.

The Portable File System operates in a stateful fashion using remote procedure (RPC - rfc1057) calls built on top of external data representation (XDR - rfc1014) protocol. The RPC protocol provides for version and authentication parameters to be exchanged for security over the network.

A server can grant access to a specific filesystem to certain clients by adding an entry for that filesystem to the server's **/etc/pfs_exports** file and running *pfs_exportfs*(1M).

A client gains access to that filesystem with the **pfs_mount** command. Once the filesystem is mounted by the client, the server issues a file handle to the client for each file (or directory) the client accesses or creates. If the disc is unmounted at the server, the file handles becomes stale, and remote requests will return stale file handle messages.

A server may also be a client with respect to filesystems it has mounted over the network, but its clients cannot gain access to those filesystems. Instead, the client must mount a filesystem directly from the server on which it resides.

ERRORS

Generally physical disk I/O errors detected at the server are returned to the client for action. If the server is down or inaccessible, the client will see the message:

PFS server *host* **not responding, retrying...**

It will retry 4 times, and then finally return failure.

AUTHOR

pfs was developed by Young Minds, Inc.

FILES

/etc/pfs_exports

SEE ALSO

pfs_exports(5), fstab(4), pfs_mount(1M), pfs_exportfs(1M), pfsd(1M).

p

NAME

ppp.Auth - PPP authentication file format

DESCRIPTION

The file **/etc/ppp/Auth** contains values used by HP PPP's implementation of the link-level authentication protocols, **CHAP** (**Challenge Handshake Authentication Protocol**) and **PAP** (**Password Authentication Protocol**). This implementation of both CHAP and PAP conforms to RFC 1334, *PPP Authentication Protocols*.

CHAP is a stronger authentication mechanism and should be used whenever possible, in preference over PAP.

Format

Each authentication specification is on its own single line of up to 1023 characters. Comments begin with a '#' and extend to the end of the line; blank lines, or lines beginning with a '#', are ignored. Fields are separated by horizontal white space (blanks or tabs).

If **pppd** is using CHAP authentication, the first word on the line must match the peer's *Name* as received in a CHAP Challenge or Response packet and the second word is used for the *Secret*. If **pppd** is using PAP authentication, the first word on the line must match the **Peer-ID** in a transmitted or received PAP Authenticate-Request packet and the second word is used for the *Password*. The default value used for the Name in transmitted CHAP packets or for the Peer-ID in transmitted PAP packets is the *hostname*(1) of the machine **pppd** is running on.

In the midst of the Name/Peer-ID and Secret/Password strings, *^x* is translated into the appropriate control character before matching, and **\xxx** represents the character corresponding to the octal number **xxx**. Other special sequences are:

\s Matches a space character (ASCII 0x20).

\t Matches a horizontal tab character (ASCII 0x09).

\n Matches a line feed character (ASCII 0x0a).

\r Matches a carriage return character (ASCII 0x0d).

The fields have the following meaning:

name The Name field of a sent or received CHAP Challenge or Response message, or the Peer-ID field of a sent or received PAP Authenticate-Request message. For transmitted packets, this is the hostname unless overridden by the **pppd name** option.

secret The secret word that the peer also knows.

optional address restrictions
A set of zero or more patterns restricting the addresses that we will allow to be used with the named peer. Patterns are separated by spaces or tabs and are parsed from left to right. Each pattern may begin with an exclamation mark to indicate that the following pattern should not be allowed. The rest of the pattern consists of digits and periods, and optionally a leading or trailing asterisk, which will match anything. If none of the patterns match, then the address will be allowed if the last pattern began with an exclamation point, and will be disallowed otherwise.

EXAMPLE

The following **Auth** provides **pppd** with a secret for use when a peer claims to be other-host, robin, or 'Jack's machine'.

```
#
#  Auth - PPP authentication name/secret file
#  Format:
#name secret      optional address restrictions
other-host secret-key !137.175.9.2 137.175.9.*/0xffffff00
robin dK3ig8G8hs 137.175.11.4
Jack's\smachine I\sam\sa\sjelly\sdonut.
```

SECURITY CONCERNS

The file **/etc/ppp/Auth** should be mode 600 or 400, and owned by root.

AUTHOR

ppp.Auth was developed by the Progressive Systems.

SEE ALSO

tun(4), ppp.Devices(4), ppp.Dialers(4), ppp.Filter(4), ppp.Keys(4), ppp.Systems(4), services(4), pppd(1), RFC 792, RFC 1548, RFC 1332, RFC 1334.

NAME

ppp.Devices - PPP physical device description file format

DESCRIPTION

The file **/etc/ppp/Devices** associates dialer types with physical devices and speeds. **pppd** examines it when placing a call to a neighboring machine. If no suitable speed is found, or if all devices associated with that speed are busy, **pppd** will try again later.

Format

Entries are one to a line; blank lines are ignored. Comments begin with a '#' and extend to the end of the line. Upper/lower case distinctions are significant. Fields on a line are separated by horizontal white space (blanks or tabs).

Each entry must contain three or more fields, in this order:

dialer Either the string 'Direct', or the name of the modem dialing chat script (found in **Dialers**) to use with this device, or the name of an external dialer program.

device The name of the device in the **/dev** directory (**ttya**, **cua**, etc.). Device names for SnapLink connections are followed by a slash and the port number in use (**rsd2a/0**, **rrz4a/2**, etc.).

speed The baud rate of the synchronous connection, or a string to be matched against the speed field of entries in **Systems** when the **Systems** device field is set to ACU. Speeds must either be valid async baud-rate numbers (as found in **<sys/ttydev.h>**) or must begin with them (2400, 38400, 19200-PEP, etc.), or must be speeds of which the SnapLink hardware is capable (9600, 56000, 64000, 1536000, etc.)

optional parameters
Any special handling for this device. Currently supported values include:

xonxoff Specifies that the line be conditioned for in-band ('software') flow control, using the characters DC3 (^S, XOFF, ASCII 0x13) to stop the flow and DC1 (^Q, XON, ASCII 0x11) to resume. The default is to use no flow control. For an outbound connection, this may be specified either in **Devices** or on the **pppd** command line.

internal-clocking
The SnapLink will provide the synchronous clock signal. By default, it expects the modem, CSU/DSU or modem eliminator to provide the clock signal. Internal-clocking cannot be used with RS-232 cables on the SnapLink.

32-bit-fcs The SnapLink will calculate 32-bit FCS values for transmitted frames, and check received frames with 32-bit FCS calculations. This is not negotiable at connection establishment time. 32-bit FCS is only available when running synchronous PPP on the SnapLink.

min-flags= *minflags*
The number of additional HDLC flag characters the SnapLink should insert between data frames. The default and minimum is 2; the maximum is 16.

ignore-cd Ignore the state of the CD (Carrier Detect, also called DCD, Data Carrier Detect) signal. This is useful for systems that don't support CD but want to run PPP over a dedicated line.

External Dailer

The external dialer program is run with the following arguments:

device name The contents of the Device field from the Devices entry.

speed The contents of the Speed field of the Systems and Devices entries.

telephone number The contents of the Phone Number field of the Systems entry.

optional parameters Copied from the Optional Parameters section of the Devices entry.

If the external dialer program exits with status 0, then the dial attempt is considered to have succeeded. Any other exit status indicates a failure.

p

EXAMPLE

```
#
#       Devices - PPP devices file
#
#Dialer     device      speed Optional parameters
T2500-PEP   cua   19200-PEP   rtscts
T1600 cub   38400 rtscts
Direct      rsd0a/0     1536000     internal-clocking
Oddball     rsd0a/1     64000 cua 9600 5551212
```

In the last line of this example, the 64Kb synchronous modem on the SnapLink's port 1 has an asynchronous dialer interface attached to the workstation's port 'a'. The Systems line would look like

```
host Oddball rsd0a/1 64000 0
```

There must be a program (or an executable shell script) called **/etc/ppp/Oddball** that dials the modem when invoked as

```
Oddball rsd0a/1 64000 0 cua 9600 5551212
```

A warning message will be printed for each unrecognized optional parameter if the *debug* level is 2 or more.

The external dialer is invoked as **root**, so you should take appropriate security precautions with its content and file protection.

AUTHOR

ppp.Devices was developed by the Progressive Systems.

SEE ALSO

tun(4), ppp.Auth(4), ppp.Dialers(4), ppp.Filter(4), ppp.Keys(4), ppp.Systems(4), pppd(1), RFC 1548, RFC 1332, RFC 1144, RFC 1055.

p

NAME

ppp.Dialers - PPP dialer description file format

DESCRIPTION

The file **/etc/ppp/Dialers** describes how to dial each type of modem attached to the UNIX system that is to be made available for outbound PPP calls. **pppd** examines it when placing a call to a neighboring machine.

When **pppd** selects a line from **Systems**, it uses the 'speed' field to select an entry in **Devices**, from which it uses the 'dialer' field to select an entry in **Dialers**. **pppd** then interprets the 'chat script' field from that dialer description.

Format

Entries are one to a line; blank lines are ignored. Comments begin with a '#' and extend to the end of the line. Upper/lower case distinctions in the dialer field are significant for matching purposes, as are strings in the chat script. Fields on a line are separated by horizontal white space (blanks or tabs). If a chat script ends with a backslash ('\'), the next line is considered a continuation of the chat script. Continuations may only occur in the midst of a chat script.

Each entry must contain these fields, in this order:

dialer The name of this dialer, to be matched against the dialer field in **Devices**.

chat-script A description of the conversation that **pppd** holds with the modem.

Chat Script Particulars

A chat script takes the form of a space-separated list of expect-send pairs. Each pair consists (at minimum) of a field to expect the 'remote' end to send, then a field to send in response. Unless a 'send' string ends with **\c**, **pppd** will follow it by sending a carriage return character (ASCII 0x0d).

Chat scripts are 'expect send expect send ...' or 'expect-send-expect send ...', where the send following the hyphen is executed if the preceding expect fails to match received text.

Certain special words may be used in the chat script to control the behavior of **pppd** as it attempts to dial. Both ABORT and TIMEOUT must be in the 'expect' phase of the chat script.

ABORT *abort-string* If **pppd** sees *abort-string* while executing the remainder of the chat script, abort the dialing attempt and note the failure in the log file.

TIMEOUT *timeout-time* While executing the current chat script, wait *timeout-time* seconds for a response before considering the dialing attempt to have timed out. Writes have a fixed 60-second timeout.

The expect-send couplet of '**"**' **P_***WORD* sets the line parity accordingly:

P_AUTO Set transmission parity based on the parity observed in characters received in 'expect' strings. This is the default.

P_ZERO Transmit characters with the parity bit set to zero (8 bits, no parity).

P_ONE Transmit characters with the parity bit set to one.

P_EVEN Transmit characters with even parity.

P_ODD Transmit characters with odd parity.

In the midst of either an 'expect' string or a 'send' string, **^***x* gets translated into the appropriate control character, and *****x* gets translated into *x*. Other special sequences are:

\s Send or receive a space character (ASCII 0x20).

\t Send or receive a horizontal tab character (ASCII 0x09).

\n Send or receive a line feed character (ASCII 0x0a).

\r Send or receive a carriage return character (ASCII 0x0d).

**** Send or receive a backslash character (ASCII 0x5c).

\^ Send or receive a carat character (ASCII 0x5e).

^*character* Send or receive the single character Ctrl-*character* (ASCII 0x00 through 0x1f).

*****ddd*	Send or receive a character, specified in octal *d*igits.
\p	Pause for .25 second before proceeding (send only).
\d	Delay for two seconds before proceeding (send only).
\K	Send a break (.25 second of zero bits).
\M	Disable hangups (sets CLOCAL or LNOHANG).
\m	enable hangups (unsets CLOCAL or LNOHANG) (the default).
\c	Don't append a carriage return character after sending the preceding string (send only).
\q	Don't print succeeding send strings (e.g. a password) in any debugging or logging output. Subsequent **\q** sequences toggle 'quiet' mode.
\T	Insert the telephone number (found in the fifth field of **Systems**) here.

EXAMPLE

```
#
#       Dialers - PPP dialers file
#
#Dialer Chat script
T1600 ABORT NO\sCARRIER ABORT NO\sDIALTONE ABORT BUSY \
      ABORT RRING\r\n\r\nRRING\r\n\r\nRRING \
      ABORT ERROR TIMEOUT 5 "" AT OK-AT-OK \
      ATS111=0DT\T TIMEOUT 30 CONNECT
#
T2500-PEP \
      ABORT NO\sCARRIER ABORT NO\sDIALTONE ABORT BUSY \
      ABORT RRING\r\n\r\nRRING\r\n\r\nRRING \
      ABORT ERROR TIMEOUT 5 "" AT OK-AT-OK \
      ATS111=0DT\T TIMEOUT 30 CONNECT\sFAST
#
USRv32bis \
      ABORT ERROR ABORT NO\sANSWER ABORT NO\sCARRIER \
      ABORT BUSY ABORT RRING\r\n\r\nRRING\r\n\r\nRRING \
      ABORT NO\sDIAL\sTONE TIMEOUT 5 "" AT&F \
      OK-ATQ0-OK ATB0E0X7&B1&H1&I0&K3&R2&S1 OK-AT-OK \
      ATS01=1S02=255S19=0 OK-AT-OK ATDT\T TIMEOUT 30 \
      CONNECT
```

AUTHOR

ppp.Dialers was developed by the Progressive Systems.

SEE ALSO

tun(4), ppp.Auth(4), ppp.Devices(4), ppp.Filter(4), ppp.Keys(4), ppp.Systems(4), pppd(1), RFC 1548, RFC 1332, RFC 1144, RFC 1055.

p

NAME

ppp.Filter - PPP packet filter specification file format

DESCRIPTION

The file **/etc/ppp/Filter** describes how on-demand PPP links are to be managed. By default, any type of packet causes the link (if down) to be brought up (connected to its remote end); any packet is allowed to traverse the link; and any packet is sufficient to reset the idle timer, expiration of which would cause the link to be shut down. This combination is not always appropriate behavior, so the filter file allows individual control based on the packet type and its source or destination. These selection criteria may be specified for any of the three phases of operation: bringing up the link, passing packets on the link, and shutting down the link due to inactivity. Packet logging detail may also be selected using the same criteria.

Format

Comments begin with a '#' and extend to the end of the line; blank lines, or lines beginning with a '#', are ignored. Upper/lower case distinctions are ignored in hostname specifications, but are significant elsewhere. Fields are separated by horizontal or vertical white space (blanks or tabs or newlines).

If a line begins with a hostname or IP address or the special word 'default', that line is considered to be the beginning of a new set of filtering specifications. The filtering specifications will be applied to any packet crossing the point-to-point link connecting this host to the peer named by that initial hostname or IP address. The hostname or IP address in the first column of the filter file refers to the peer (system or router or terminal server) at the remote end of the point-to-point (PPP or SLIP) link. The hostname or IP address in the first column of the filter file, and associated with the link peer, is unrelated to the source or destination IP address of any packet crossing the link. If the link peer's address doesn't match any name or address specified in the first column of filter file, the filter specification following the special word 'default' will be used.

If a newline is followed by white space, that line is a continuation of the filtering specification already in progress.

There are four keywords to describe the actions taken by **pppd** in response to a particular packet:

bringup Describes those packets that will cause a call to be placed and a connection initiated. Packets of this sort also must qualify to 'pass' across the link, either by being explicitly mentioned or by inclusion in a larger class in the 'pass' section.

pass Describes those packets that will be allowed to traverse the link on an already-established connection. Only packets which would be passed can cause the link to be brought up. Any packet that is not passed is optionally logged, then discarded.

keepup Describes packets that will reset the idle timer, thereby keeping the line connected.

log Describes packets whose headers or contents are to be noted in the log file.

After each action keyword comes stanzas, separated by white space, describing packets that fit the criteria for that action. Each stanza is processed in the order shown in the file, and contain restrictions or permissions on the packets encountered. As soon as a pattern or a condition is found that matches the packet in question, **pppd** takes the indicated action and ignores the rest of the listed stanzas (i.e. inclusive *or* with shortcut evaluation).

Stanzas may contain IP protocol numbers, optionally hyphen-separated ranges of TCP or UDP port numbers along with the **/tcp** or **/udp** qualifier, numbers representing ICMP message types or codes (which can be found in **<netinet/ip_icmp.h>**) along with the '/icmp' qualifier, service names corresponding to entries in **/etc/services**, or names or IP addresses of hosts or networks, or the special keyword 'all', which is the default for all actions except 'log', where the default is '!all'. (Usually, it is unnecessary to use 'all'; as a convenience, **pppd** automatically adds a '!all' at the end of a stanza list if the last stanza *is not* negated, and add an 'all' at the end of a stanza list if the last stanza *is* negated. For example, in the typical case of 'log' this sensibly results in *only* those packets matching the stanzas shown being logged, and no others. In the typical case of 'pass', this results in certain listed packets being restricted, but allowing the passage of all others.)

If a network is specified, either by name or by address, then the corresponding network mask must also be specified if it is of a different size than the default for that class of network. The network mask and additional 'and' conditions within a stanza are separated by slashes ('/'), and may be specified either as a series of decimal numbers separated by periods, or as a single 32-bit hexadecimal number. The sense of a stanza may be negated by prefixing it with an exclamation mark ('!').

In the 'log' filter specification, the special keyword 'trace' causes the *contents* (as well as headers) of the indicated type of packet to be written to the log file. Also in the 'log' filter specification, the special flag 'rejected' signifies that the packet is to be logged only if it was rejected by the 'pass' filter.

Since TCP data streams are opened when the initiator sends a SYN packet to the intended recipient, **pppd** can distinguish between outbound (sent from this host) and inbound (coming from the other end of the link) uses of TCP applications such as telnet or FTP. The special keyword 'syn' allows filtering or logging these connection starters. Qualifying it with 'recv' or 'send' allows sessions to be started or logged only if they are initiated in the indicated direction. The special keyword 'fin' allows filtering or logging the packets that close TCP connections.

The 'src' and 'dst' keywords serve to distinguish ports, addresses or hostnames, as applying to the source or destination, respectively, of the packet. If both are applied to the same stanza (e.g. **.../src/dst**), then both the source and destination address and/or port must match.

The **unreach=** keyword causes an ICMP Destination Unreachable message (RFC 792 and RFC 1122 section 3.2.2.1) to be sent to the packet's source address, bearing the indicated code field, which may be chosen from

net	The destination network is unreachable.
host	The destination host is unreachable.
prot	The designated transport protocol is not supported.
protocol	The designated transport protocol is not supported.
port	The designated transport protocol (e.g., UDP) is unable to demultiplex the datagram but has no protocol mechanism to inform the sender.
needfrag	Fragmentation is needed and the Don't Fragment flag is set.
srcfail	Source route failed.
net-unknown	The destination network is unknown.
host-unknown	The destination host is unknown.
host-isolated	The source host is isolated.
net-prohibited	Communication with the destination network is administratively prohibited.
host-prohibited	Communication with the destination host is administratively prohibited.
net-tos	The destination network is unreachable for the designated type of service.
host-tos	The destination host is unreachable for the designated type of service.

The **ip-opt=** keyword can be used to select packets based on whether they bear various IP options (RFC 1122 section 3.2.1.8 and RFC 791 section 3.1 (pps 16ff)), selected from

rr	Record Route is used to trace the route an internet datagram takes.
ts	Time Stamp.
security	Security is used to carry Security, Compartmentation, User Group (TCC), and Handling Restriction Codes compatible with DOD requirements.
lsrr	Loose Source Routing is used to route the internet datagram based on information supplied by the source.
ssrr	Strict Source Routing is used to route the internet datagram based on information supplied by the source.
srcrt	Either Loose Source Routing or Strict Source Routing.
any	Any IP option - could even match the No Operation option.

EXAMPLES

Default Behavior

The following **Filter** file describes the default behavior of **pppd**, either in the absence of a filter specification file or in the case of an empty file:

```
#       Filter - PPP configuration file,
#       binding packet types to actions.
```

p

```
#       Describes the default behavior of the daemon:
default     bringup all pass all keepup all log !all
```

The default behavior is no restriction of packets, and no logging.

Internet Firewall

A ‘pass’ line like this might be appropriate as a security firewall between an organizational network and the larger Internet:

```
internet-gateway
        bringup !ntp !3/icmp !5/icmp !11/icmp !who !route
                !nntp !89
        pass    nntp/137.39.1.2 !nntp
                telnet/syn/recv/137.175.0.0
                !telnet/syn/recv !ftp/syn/recv
                !login/syn/recv !shell/syn/recv !who
                !sunrpc !chargen !tftp !supdup/syn/recv
                !exec !syslog !route !6000/tcp/syn/send
        keepup  !send !ntp !3/icmp !5/icmp !11/icmp
                !who !route !89
        log     rejected
```

This ‘pass’ specification allows NNTP (Usenet news) transactions with one peer and no others. It allows incoming Telnet sessions from hosts on only one network, disallows all other incoming Telnet, SUPDUP, and FTP sessions, and allows all outgoing Telnet SUPDUP, and FTP sessions.

It allows X Window System clients running elsewhere to display on local window servers, but it allows no local X clients to use displays located elsewhere. It disallows all SUN RPC traffic, thereby guarding the local YP/NIS and NFS servers from outside probes and filesystem mounts. Alas, it also disallows local machines from mounting filesystems resident on NFS servers elsewhere, but this can’t be helped because NFS uses RPC which is a UDP service, and therefore without the SYN and FIN packets that can be used to characterize the direction in which a TCP stream is being initiated. It blocks several other sorts of traffic that could be used for nefarious purposes, and the absence of a trailing ‘!all’ means that any traffic not explicitly blocked is permitted to pass.

The ‘bringup’ and ‘keepup’ lines are appropriate for an intermittent dial-up connection, so that various error conditions won’t cause the link to be established, nor to keep the call open beyond its usefulness. OSPF (Open Shortest Path First) routing packets (IP protocol number 89, from RFC-1340) will cross the link, but won’t cause it to be brought up, nor keep it up if it’s otherwise idle. Usenet news traffic won’t bring up the link, but once started, the link won’t be shut off in the middle of a news batch. The ‘log rejected’ line keeps a record of every packet that is blocked by the ‘pass’ line, so that unsuccessful penetration attempts will be noted.

An Extremely Complex Example

The following **Filter** file instructs the daemon that a connection to any neighbor except the host ‘backbone’ be brought up in response to any packet except for those generated by NTP, ICMP Destination Unreachable, and **rwhod**. If those are the only types of packets flowing across the link, it will not be kept up, but all packets are allowed to cross the link while it is up. Packets sent out will not reset the idle timer, but packets received from the peer will. If the peer goes down and modem problems cause the phone not to be hung up, (and the *idle* command-line argument has been specified) **pppd** will hang up the connection and retry.

In the special case of the host ‘backbone’ (perhaps a server belonging to a network connectivity vendor), only telnet and FTP sessions, SMTP electronic mail, NNTP network news, and Domain Name System queries are considered sufficient cause to bring the link up or to keep it up if otherwise idle.

Once the link is up, all the above plus NTP clock chimes and ICMP messages may flow across the link. No packets to or from a particular host, nor any packets except Domain Name System queries and responses for any host on subnet 42 of the class B network 137.175 are ever allowed to cross the link, nor would they cause the link to be initiated. We allow telnet and FTP sessions only if they are initiated in the outbound direction.

We log one-line descriptions of various ICMP problem messages (Unreachable, Time Exceeded), and the complete contents of ICMP messages reporting IP header problems. We log all telnet and FTP sessions, including inbound attempts (though they will fail because they are excluded in the ‘pass’ specification above). We also log the header of the first packet of any electronic mail message flowing over this link on its way to or from a specific host.

```
#
#     Filter -    PPP configuration file binding packet
#           types to actions.
#
#     For packets that would pass, these services
#     will bring up the link:
#
backbone bringup smtp nntp domain telnet ftp
#
#     Once brought up, these will pass (or not):
#
      pass  !131.119.250.104
            domain/137.175.42.0/255.255.255.0
            !137.175.42.0/0xffffff00
#                 (alternative ways of
#                  expressing subnet mask)
            !telnet/syn/recv !ftp/syn/recv
            domain smtp nntp ntp icmp telnet ftp
#
#     Packets received for the services shown will
#     reset the idle timer.
#
      keepup      !send smtp nntp domain telnet ftp
#
#     Only these messages will have headers or contents
#     logged, unless higher-level debugging is set:
#
      log   3/icmp 11/icmp 12/icmp/trace
            telnet/syn ftp/syn
            smtp/syn/terminus.netsys.com
#
default bringup   !ntp !3/icmp !who
        keepup          !send !ntp !3/icmp !who
```

RECOMMENDATIONS

Simpler filter specifications allow **pppd** to start up quicker and run faster, with less processing overhead for each packet, but that overhead is likely to present a problem only at very high line speeds (like T1). The 'backbone' example shown above is severe overkill for the sake of illustration, evolved over a period of several weeks, and took the authors several tries to get right. Start with a simple filter specification and add each special case only as the need arises, usually as the result of watching packet logs. Then test carefully to ensure that your change had only the desired effect.

Be very careful with header logging and even more careful with packet content tracing. Make the selection criteria very narrow, or the log file will grow extremely large in a short period of time. Also, if the daemon is running on a diskless workstation or if the log file is on a NFS-mounted file system, excessive amounts of logging information will drastically impede the daemon's ability to process at high packet rates. Remember, NFS writes are synchronous.

If you specify host names, be sure that their addresses are available locally, even with the connection down. If you find that you must bring up a connection to resolve a domain name, consider using that host's IP address (decimal numbers separated by periods) in both **Filter** and **Systems** instead.

If you want to specify all Domain Name System traffic, use 'domain' which will be expanded to entries for both **53/tcp** and **53/udp**. (Some DNS traffic uses each transport.) To allow queries but disable domain transfers, use **!domain/tcp**. Similarly, some systems' older **/etc/services** files, as distributed by the manufacturer, list NTP as a TCP service. When the current UDP NTP implementation was installed on your system, the administrator may have left the old **123/tcp** entry along with the correct **123/udp**. The correct solution is to remove the **123/tcp** entry from **/etc/services**. A workaround would be to specify **123/udp** in **Filter**.

DEC ULTRIX 4.2 and some other systems may have no entry for FTP's data socket in their **/etc/services** file. If you want to log the bulk data connections as well as the control connections, you'll need to either add an entry for 'ftp-data' to **/etc/services**, or use **20/tcp** explicitly in **Filter**. The former is preferable because it will cause the log file entry to contain the symbolic name

p

('ftp-data') rather than the socket/protocol notation.

If your **/etc/services** file is missing some application-level protocols that you consider useful, you can populate it with entries from the Assigned Numbers RFC, number 1340. For example, you may find it useful to add lines like

```
gopher      70/tcp
gopher      70/udp
kerberos    88/tcp
kerberos    88/udp
snmp        161/tcp
snmp        161/udp
nextstep    178/tcp
nextstep    178/udp
prospero    191/tcp
prospero    191/udp
x11         6000/tcp
```

if you're using those applications, and if they're not already in your **/etc/services** file as received from your system's manufacturer. If you augment your **/etc/services** this way, then instead of using entries like

```
pass  !6000/tcp/syn/send
```

your **Filter** could use entries like

```
pass  !x11/syn/send
```

which is much more readable. A list of TCP and UDP service numbers and names, culled from the Assigned Numbers RFC, is available in **Examples/services.ex**.

AUTHOR

ppp.Filter was developed by the HP.

SEE ALSO

tun(4), ppp.Auth(4), ppp.Devices(4), ppp.Dialers(4), ppp.Keys(4), ppp.Systems(4), services(4), pppd(1), RFC 791, RFC 792, RFC 1055, RFC 1548, RFC 1332, RFC 1122, RFC 1144, RFC 1340.

NAME

ppp.Keys - PPP encryption keys file format

RESTRICTIONS

Encryption is not available in software exported from the USA. The HP's **pppd** command does not support **gw-crypt** option, customer may contact sales@progressive-systems.com to obtain encryption functionality.

DESCRIPTION

The keys file named in the **gw-crypt** option on the **pppd** command line contains key values used by HP PPP's implementation of link-level encryption. Before transmission, packets with source and destination addresses matching the endpoints on a keys file line are encrypted using DES with the key specified on that keys file line. Upon reception, packets with source and destination addresses matching those on a keys file line are decrypted using DES with the key specified on that keys file line.

Format

Each key specification is on its own single line of up to 1023 characters. Comments in the keys file begin with a '#' and extend to the end of the line; blank lines, or lines beginning with a '#', are ignored. Fields are separated by horizontal white space (blanks or tabs).

The first two words on a key line are compared with the source and destination addresses of each packet to be transmitted and each received packet. The endpoint address specifications may contain either host or network names, or host or network addresses. If a network is specified, either by name or by address, then the corresponding network mask must also be specified if it is of a different size than the default for that class of network. The mask is separated from the network name or address by a slash ('/'), and may be specified either as a series of decimal numbers separated by periods, or as a single 32-bit hexadecimal number, optionally with a C-style '0x' prefix.

The remainder of the key line is a 56 bit (14 digit) hexadecimal number (without the C-style '0x' prefix), used as the DES key between the specified pair of hosts or networks. The digits may be separated by horizontal white space for readability. If the key contains fewer or more than 14 hexadecimal digits, the line is ignored. If the key is weak or semi-weak, a warning message will be printed in the log file and the specified key will be used for encryption anyway.

EXAMPLE

The following keys file provides **pppd** with keys for use when encrypting or decrypting traffic between the indicated pairs of hosts or networks:

```
#
#  Keys - PPP encryption keys file
#
#  Format:
#endpoint               endpoint                 key
frobozz.foo.com         glitznorf.baz.edu        feed face f00d aa
147.225.0.0             38.145.211.0/0xffffffc0 b1ff a c001 d00d 1
128.49.16.0/0xffffff00  198.137.240.100                  0123456789abcd
193.124.250.136         143.231.1.0/0xffffff00  e1c3870e1c3870
```

RECOMMENDATIONS

Avoid using weak or semi-weak keys. These are weak DES keys:

```
00000000000000
FFFFFFFFFFFFFF
1E3C78F1E3C78F
E1C3870E1C3870
```

These are semi-weak DES keys:

```
01FC07F01FC07F
FE03F80FE03F80
1FC07F00FE03F8
E03F80FF01FC07
01C007001E0078
E003800F003C00
1FFC7FF0FFC3FF
```

p

```
FE3FF8FFE1FF87
003C00F001C007
1E007800E00380
E1FF87FF1FFC7F
FFC3FF0FFE3FF8
```

SECURITY CONCERNS

The keys file should be mode 600 or 400, and owned by root.

Packets' IP headers are not encrypted, though their TCP, UDP, or ICMP headers are encrypted along with the user data portion. This allows encrypted packets to traverse normal internetworks, but permits snoopers to analyze traffic by its endpoints.

Since the TCP, UDP, or ICMP header is encrypted, protocol-based filters along the packet's path will be unable to discern whether it is SMTP, Telnet, or any other network service. This means that encrypted traffic will only permeate packet-filtering firewalls if the firewall allows all traffic between the endpoints, regardless of traffic type. HP PPP/SLIP software for HP-UX systems, when deployed as the endpoint gateways of the encrypted traffic, decrypt incoming encrypted traffic before applying their configured packet filtering rules.

AUTHOR

ppp.Keys was developed by the Progressive Systems.

SEE ALSO

tun(4), ppp.Auth(4), ppp.Devices(4), ppp.Dialers(4), ppp.Filter(4), ppp.Systems(4), pppd(1), RFC 792, RFC 1548, RFC 1332, RFC 1334.

p

NAME

ppp.Systems - PPP neighboring systems description file format

DESCRIPTION

The file **/etc/ppp/Systems** describes how to connect with neighboring systems via PPP.

Format

Entries are one to a line; blank lines are ignored. Comments begin with a '#' and extend to the end of the line. Upper/lower case distinctions are ignored in hostname specifications, but are significant elsewhere. Fields on a line are separated by horizontal white space (blanks or tabs). If a chat script ends with a backslash ('\'), the next line is considered a continuation of the chat script. Continuations may only occur in the midst of a chat script.

Each entry must contain six fields, in the following order:

name The hostname or IP address of the destination machine, which should be resolvable locally.

when A string that indicates the days of the week and the times of day when the system can be called (for example, MoTuTh0800-1740). The day portion may be a list containing any of Su, Mo, Tu, We, Th, Fr or Sa. The day may also be Wk for any weekday (same as MoTuWeThFr) or Any for any day (same as SuMoTuWeThFrSa).

You can indicate hours in a range (for example, 0800-1230). If you do not specify a time, calls will be allowed at any time.

Note that a time range that spans 0000 is permitted. For example, 0800-0600 means that all times are allowed except times between 6 AM and 8 AM.

Multiple date specifications that are separated by a vertical bar (|) are allowed. For example, Any0100-0600|Sa|Su means that the system can be called any day between 1 AM and 6 AM or any time on Saturday and Sunday.

The entire (sequence of) days and times may be followed by a semicolon and up to three decimal numbers separated by hyphens:

one If only one number follows the semicolon, it is used as the redial delay, which is the initial time (in seconds) before a failed call will be retried. For example, Any;60 means call any time, but wait at least 60 seconds after a failure has occurred before trying to call again. If a call retry fails, **pppd** will double the delay before trying again. If no initial retry delay is specified, 10 seconds is assumed.

two If two numbers follow the semicolon, the second number is used as the maximum redial delay, which is the maximum time (in seconds) to delay before retrying a call. The retry time will double with each unsuccessful call until it reaches this value, after which the call will be retried every time the maximum number of seconds passes. If no maximum retry delay is specified, 3600 seconds is assumed.

three If three numbers follow the semicolon, the first is used as the callback delay, the second as the redial delay, and the third as the maximum redial delay. The callback delay is the time (in seconds) to wait before attempting to re-establish a previously active connection that ended because of an abrupt line disconnection (a Hangup or SIGHUP event in the log file). The default is not to delay before calling back.

During the delay following an unsuccessful call, any level 7 debugging messages written to **pppd.log** will have the message 'dial failed' appended.

device If set to 'ACU', any device in **Devices** with a matching speed may be used. The device's dialer chat script will be executed first, followed by the **Systems** chat script.

If set to the name of a device in the **/dev** directory (**tty00**, **cua**, etc.), then there may be an optional corresponding **Direct** entry in **Devices**, **Dialers** will not be consulted, and only the **Systems** chat script will be executed.

If set to 'tcp', then it must be followed by a slash, then the hostname or IP address of the system that will serve as the destination of the PPP link, then another slash, then the socket number on which to contact the remote PPP daemon.

speed The speed of the connection. If the device field is ACU, the speed field will be string matched against entries in **Devices**. Speeds must either be valid speed numbers or must begin with them (2400, 38400, 19200-PEP, etc.). If the device field is 'tcp...' or 'telnet...', the speed field is

p

ignored, but must be present as a place-holder.

phone number
The value to replace the **\T** escape sequence in the dialer script. If the device field names an entry in **/dev**, the phone number field is optional. If the device field is 'tcp...' or 'telnet...', the phone number field is ignored if present, but must be present as a placeholder.

chat script
A description of the conversation that **pppd** holds with the remote machine.

Chat Script Particulars

A chat script takes the form of a word to expect the remote end to send, followed by a word to send in response. Unless a 'send' string ends with **\c**, **pppd** will follow it by sending a carriage return character (ASCII 0x0d).

Chat scripts are 'expect send expect send ...' or 'expect-send-expect send ...', where the send following the hyphen is executed if the preceding expect fails to match received text.

Certain special words may be used in chat script 'send' strings to control the behavior of **pppd** as it attempts to dial. Both ABORT and TIMEOUT must be in the 'expect' phase of the chat script.

ABORT *abort-string* — If **pppd** sees *abort-string* while executing the remainder of the chat script, abort the dialing attempt and note the failure in the log file.

TIMEOUT *timeout-time* — While executing the current chat script, wait *timeout-time* seconds for an expected response before regarding the dialing attempt as having failed. Writes have a fixed 60-second timeout.

The expect-send couplet of '**"**' **P_***WORD* sets the line parity accordingly:

P_AUTO — Set transmission parity based on the parity observed in characters received in 'expect' strings. This is the default.

P_ZERO — Transmit characters with the parity bit set to zero (8 bits, no parity).

P_ONE — Transmit characters with the parity bit set to one.

P_EVEN — Transmit characters with even parity.

P_ODD — Transmit characters with odd parity.

The backquote character (**'**) surrounds the name of a program that is to be run before proceeding. If the program is run in the 'send' phase of a chat script couplet, its standard output will be sent to the peer when the program exits. Chat script processing continues when the program exits.

In the midst of either an 'expect' string or a 'send' string, **^***x* gets translated into the appropriate control character, and *****x* gets translated into *x*. Other special sequences are:

\s	Send or receive a space character (ASCII 0x20).
\t	Send or receive a horizontal tab character (ASCII 0x09).
\n	Send or receive a line feed character (ASCII 0x0a).
\r	Send or receive a carriage return character (ASCII 0x0d).
****	Send or receive a backslash character (ASCII 0x5c).
\^	Send or receive a carat character (ASCII 0x5e).
^*character*	Send or receive the single character Ctrl-*character* (ASCII 0x00 through 0x1f).
*****ddd*	Send or receive a character, specified in octal *d*igits.
\p	Pause for .25 second before proceeding (send only).
\d	Delay for two seconds before proceeding (send only).
\K	Send a break (.25 second of zero bits).
\M	Disable hangups (sets CLOCAL or LNOHANG).
\m	enable hangups (unsets CLOCAL or LNOHANG) (the default).
\c	Don't append a carriage return character after sending the preceding string (send only).

p

\q Don't print following send strings (e.g. a password) in any debugging or logging output. Subsequent **\q** sequences toggle 'quiet' mode.

\A Parse the incoming string as an IP address, written as four decimal numbers separated by periods, and use it for the local end of the point-to-point connection (receive only).

EXAMPLE

In the example below, we call host 'everyone' using a Telebit PEP modem with its DTE interface set at 19200 bps. We call host 'nobody' using a V.32/V.42/V.42bis modem that's capable of driving a 38400 DTE, and we are connected to host 'someone' via a direct cable attached to **/dev/ttya**, running asynchronous PPP. We talk to 'anyone' via a T1 CSU/DSU attached to port 0 on a SnapLink. And we connect with pseudo-one via a PPP connection tunneled across a TCP stream to port 77 on realone.somewhere.com.

If we are unsuccessful at connecting with 'someone' we will try again in two seconds. If that attempt fails, we will wait four seconds before the next attempt; then eight, then sixteen, then thirty two, then forty seconds. We will continue attempting to contact 'someone' every forty seconds. Our retry intervals and maximum backoff values for 'everyone' and 'nobody' are the default '10-3600'.

The notation "" "" means to expect nothing, then send nothing (followed by a carriage return). The implicit carriage return is often useful for eliciting a response from a remote system.

```
#
#       Systems - PPP systems file
#
everyone Any ACU 19200-PEP 5551212 in:--in: Pwe word: \qfoObar
nobody Any ACU 38400 5551213 in:--in: Pthey word: \qbaZz1ng
someone Any;2-40 cua 38400 0 in:--in: Pthem word: \qmeumBle
anyone Any rsd0a/0 1536000
pseudo-one Any;2-2 tcp/realone.somewhere.com/57
```

RECOMMENDATIONS

The default retry time and backoff (i.e. Any;10-3600) are appropriate for use with dialup connections where the PPP connection must be reestablished as quickly as possible after an interruption but where it is not desirable to continuously redial a host that may be down. A much shorter maximum would be appropriate for a dedicated line between two systems, or where call attempts cost nothing.

Moderate call retry times, such as 60 seconds, work well on systems that can establish connections in either direction using dialup modems, to avoid deadlocks waiting for telephone busy signals from each calling the other at the same time. Because of the difference between the behaviors of originating and answering modems, the 60-second clocks will usually start ticking at different times, allowing one side to call the other without interference. Alternatively, different call retry times may be specified at either end of a link to help keep the two systems from calling each other simultaneously.

If you specify host names, be sure that their addresses are available locally, even with the connection down. If you find that you must bring up a connection to resolve a domain name, consider using that host's IP address (decimal numbers separated by periods) in both **Filter** and **Systems** instead.

Automatic failover recovery can be arranged between systems that each have multiple modems, or multiple connection methods. If two systems are connected via a dedicated line (sync or async), that entry should be first in **Systems**, followed by another entry describing an on-demand dial-up connection. See the *HP PPP User Guide* for more details.

SECURITY CONCERNS

The file **/etc/ppp/Systems** should be mode 600.

AUTHOR

ppp.Systems was developed by the Progressive Systems.

SEE ALSO

tun(4), ppp.Auth(4), ppp.Services(4), ppp.Dialers(4), ppp.Filter(4), ppp.Keys(4), pppd(1), RFC 1548, RFC 1332, RFC 1144, RFC 1055.

NAME

privgrp - format of privileged values

SYNOPSIS

#include <sys/privgrp.h>

DESCRIPTION

setprivgrp() sets a mask of privileges, and **getprivgrp()** returns an array of structures giving privileged group assignments on a per-group-ID basis (see *getprivgrp*(2)). <**privgrp.h**> contains the constants and structures needed to deal with these system calls, and contains:

```
/*
 * Privileged group definitions --
 * the numeric values may vary between implementations.
 */
#define PRIV_RTPRIO             1
#define PRIV_MLOCK              2
#define PRIV_CHOWN              3
#define PRIV_LOCKRDONLY         4
#define PRIV_SETRUGID           5

/* Maximum number of privileged groups in system */
#define PRIV_MAXGRPS           32

/*
 * Size of the privilege mask,
 * based on largest numbered privilege
 */
#define PRIV_MASKSIZ            1

/*
 * Structure defining the privilege mask
 */
struct privgrp_map {
    int             priv_groupno;
    unsigned int    priv_mask[PRIV_MASKSIZ];
};
```


Privileges are as follows:

PRIV_RTPRIO	Allows access to the **rtprio()** system call (see *rtprio*(2)).
PRIV_MLOCK	Allows access to the **plock()** system call (see *plock*(2)).
PRIV_CHOWN	Allows access to the **chown()** system calls (see *chown*(2)).
PRIV_LOCKRDONLY	Permits the use of the **lockf()** system call for setting locks on files open for reading only (see *lockf*(2)).
PRIV_SETRUGID	Permits the use of the **setuid()** and **setgid()** system calls for changing respectively the real user ID and real group ID of a process (see *setuid*(2)).

Privileges are described in a multi-word mask. The value of the **#define** for each privilege is interpreted as a bit index (counting from 1). Thus a group-id can have several different privileges associated with it by having different bits ORed into the mask.

The system is configured with a specified maximum number of groups with special privileges. **PRIV_MAXGRPS** defines this maximum. Of this maximum, one is reserved for global privileges (granted to all processes), and the remainder can be assigned to actual group-ids.

PRIV_MASKSIZ defines the size of the multi-word mask used in defining privileges associated with a group-ID.

Privileges are returned to the user from the **getprivgrp()** system call in an array of structures of type **struct privgrp_map**. The structure associates a multi-word mask with a group-ID.

SEE ALSO

getprivgrp(2).

NAME

profile - set up user's environment at login time

DESCRIPTION

If the file **/etc/profile** exists, it is executed by the shell for every user who logs in. The file **/etc/profile** should be set up to do only those things that are desirable for *every* user on the system, or to set reasonable defaults. If a user's login (home) directory contains a file named **.profile**, that file is executed (via the shell's **exec .profile**) before the session begins. **.profile** files are useful for setting various environment parameters, setting terminal modes, or overriding some or all of the results of executing **/etc/profile**.

EXAMPLES

The following example is typical (except for the comments):

```
#  Make some environment variables global
    export MAIL PATH TERM
#  Set file creation mask
    umask 22
#  Tell me when new mail comes in
    MAIL=/var/mail/myname
#  Add my /bin directory to the shell search sequence
    PATH=$PATH:$HOME/bin
# Set terminal type
    echo "terminal: \c"
    read TERM
    case $TERM in
        300)      stty cr2 nl0 tabs; tabs;;
        300s)     stty cr2 nl0 tabs; tabs;;
        450)      stty cr2 nl0 tabs; tabs;;
        hp)       stty cr0 nl0 tabs; tabs;;
        745|735)  stty cr1 nl] -tabs; TERM=745;;
        43)       stty cr1 nl0 -tabs;;
        *)        echo "$TERM unknown";;
    esac
```

A more complete model **.profile** can be found in **/etc/skel/.profile**.

FILES

$HOME/.profile
/etc/profile

SEE ALSO

env(1), login(1), mail(1), sh(1), stty(1), su(1), environ(5), term(5).

p

NAME

proto - prototype job file for *at*(1)

SYNOPSIS

/var/adm/cron/.proto

/var/adm/cron/.proto. *queue*

DESCRIPTION

When a job is submitted to **at** or **batch**, the job is constructed as a Bourne shell script (see *at*(1)). The job file is created in **/var/spool/cron/atjobs** as follows:

- **at** creates a header describing the job as an **at** job or a **batch** job. **at** jobs submitted to all queues other than queue **a** are listed as **batch** jobs. The header is:

  : at job for an **at** job, or

  : batch job for a **batch** job.

- A set of Bourne shell commands is added to make the environment (see *environ*(5)) for the **at** job the same as the current environment.

- **at** then copies text from the prototype file to the job file, except for special *variables* that are replaced by other text:

  $d Replaced by the current working directory.

  $l Replaced by the current file size limit (see *ulimit*(2)).

  $m Replaced by the current umask (see *umask*(2)).

  $t Replaced by the time at which the job should be run, expressed as seconds since January 1, 1970, 00:00 Coordinated Universal Time, preceded by a colon.

  $< Replaced by text read by **at** from the standard input (that is, the commands provided to **at** to be run in the job).

- When a job is submitted to queue *queue*, **at** uses the file **/var/adm/cron/.proto.***queue* as the prototype file if it exists. Otherwise it uses the file **/var/adm/cron/.proto**.

p

EXAMPLES

The following **.proto** file creates commands to change the current directory, file size limit, and umask in the job to their respective values as they existed when **at** was originally run. These commands are inserted before the commands in the job:

```
cd $d
ulimit $l
umask $m
$<
```

SEE ALSO

at(1), queuedefs(4).

STANDARDS CONFORMANCE

proto: SVID2, SVID3

NAME

protocols - protocol name data base

DESCRIPTION

This file associates protocol numbers with official protocol names and aliases. This allows the user to refer to a protocol by a symbolic name instead of a number. For each protocol a single line should be present with the following information:

<*official protocol name*> <*official protocol number*> <*aliases*>

These mappings are defined in RFC 1700 *Assigned Numbers*.

Aliases are other names under which the protocol is also known. For example:

```
tcp     6       TCP
```

In this example, the library call **getprotobyname()** can be invoked as:

```
p = getprotobyname("TCP");
```

instead of

```
p = getprotobyname("tcp");
```

Both produce the same results.

A line cannot start with a space. Items are separated by any number of blanks and/or tab characters. A **#** character indicates the beginning of a comment. Characters from the **#** to the end of the line are not interpreted by routines which search the file.

Protocol names can contain any printable character other than a white space, new-line, or comment character. Trailing blanks or tabs are allowed at the end of a line.

EXAMPLES

```
tcp     6       TCP     # transmission control protocol
udp     17      UDP     # user datagram protocol
```

AUTHOR

protocols was developed by the University of California, Berkeley.

SEE ALSO

getprotoent(3N).

p

NAME

prpwd - protected password authentication database files used for trusted systems

SYNOPSIS

/tcb/files/auth/...

DESCRIPTION

An authentication profile is maintained for each user on the system. A user profile is kept in a protected password database file that is accessible only to the System Administrator. The protected password database files contain among other things the encrypted password for the user account. On a trusted system, the passwords are hidden from normal users.

The protected password database files do not obviate the need for the **/etc/passwd** and the **/etc/group** files. Users must be defined in the **/etc/passwd** file in order to use the system. The protected password database file for a user contains the user name and user id to provide a correlation to the user's **/etc/passwd** entry. These must match or the user account will be treated as invalid.

Protected password database files are maintained in the **/tcb/files/auth** hierarchy. This directory contains other directories each named with a single letter from the alphabet. User authentication profiles are stored in these directories based on the first letter of the user account name. This enables an efficient search operation to locate the file for a specific user name. For instance, the authentication profile for the **root** account is located in the **/tcb/files/auth/r** directory and can be accessed by opening the file **/tcb/files/auth/r/root**.

Fields defined in a file are user specific values. These values override the system default values. Trusted programs check first for the existence of user specific parameters before using a system default value.

A protected password database file contains keyword field identifiers and, depending on the field type, a value for that field (certain field types do not require an explicit value). The exact syntax for field specifications is described in *authcap*(4). Field specification is consistent for all system authentication databases. The keyword field identifiers supported by the protected password database file and their associated function are given in the following descriptions:

u_name This is the user name for the account which must match the name of the file and the user name from the corresponding **/etc/passwd** entry.

u_id This is the user id for the account which must match the user id field of the corresponding **/etc/passwd** entry.

p

u_pwd This field contains the encrypted password for the account if the account has a password.

u_owner This field contains the owner of the account.

u_booauth If this field exists and contains a value greater than zero (typically 1), and the boot authenticate flag is set in the system default file, then this user has authority to boot the system. If the boot authenticate flag is not set in the system default file then this field is not used.

u_audid This field contains the audit ID for the user.

u_auditflag This field contains the audit flag for the user.

u_minchg This field specifies the minimum password change time in seconds. If non-zero, the password cannot be changed until the specified number of seconds since the last successful password change have passed unless the person changing the password is authorized to override this constraint.

u_maxlen This field specifies the maximum length of the user account password and should be less than the system-wide maximum value defined by the *<prot.h>* constant **AUTH_MAX_PASSWD_LENGTH.**

u_exp This field is a *time_t* value that specifies when the account password will expire. When a password expires, system authentication programs will request that the password be changed when the user logs into the system. If the password lifetime expires before the password is changed, the account will be locked.

u_life This field is a *time_t* value that specifies the lifetime of a password. If this time is reached, the account will be locked and can only be unlocked by an authorized system administrator.

u_succhg This field is a *time_t* value that indicates the time of the last successful password change. This field should only be set by programs that can be used to change the account password.

u_unsucchg This field is a *time_t* value that indicates the time of the last unsuccessful password change. This field should only be set by programs that can be used to change the account password.

u_acct_expire
This field is the *time_t* time is seconds that the account will be usable. After this time interval the user will no longer be allowed to login. This field is different from the **u_life** field in that the **u_life** field is the time from the last password change. **u_acct_expire** is not affected by the changing of the password.

u_max_llogin This value, in secconds, is the maximum time allowed between logins. If the time between the last login and the current time exceeds this value, the account is locked and the user can no longer logon.

u_pw_expire_warning
This value, in seconds, is the time at which a warning will appear prior to the expiration of the users password.

u_pickpw This value is a flag that controls the ability of the user to pick a password for the account. This permits an account to be configured so that a user can not pick a password but instead has a password generated by the system for the account.

u_genpwd This flag field controls the ability of a user to generate a password for the account. The system is capable of generating passwords containing random letters, characters, or words.

u_restrict This flag field controls whether password triviality checks are performed on any user chosen passwords. Triviality checks performed include verifying that the password does not represent a login or group name, a palindrome, or a word recognized by the *spell*(1) program. See *acceptable_password*(3) for more information on triviality checks for passwords.

u_nullpw This flag controls the ability of the user to choose a null password for the account.

u_pwchanger This field records the user id of the last person to change the account password if that user was not the same the account's user. This is used to warn the user at login time if the account password has been changed possibly without the knowledge of the user.

u_pw_admin_num
This field holds the random number the user must supply to login after the account is reset by the system administrator. This field is removed after a successful login.

u_genchars This flag field controls the ability of the user to generate random characters for a password.

u_genletters This flag field controls the ability of the user to generate random letters for a password.

u_tod This field contains a comma separated list of time-of-day specification entries that controls when the user account can be used for login. For more information on the format of a list entry, see *tod*(3).

u_suclog This field is a *time_t* value that contains the system time of the last successful login to the account.

u_unsuclog This field is a *time_t* value that contains the system time of the last unsuccessful login to the account.

u_suctty This field is a character string that identifies the name of the terminal or remote host associated with the last successful login to the account.

A remote host specification consists of the ASCII representation of the Internet address of the host. This field is converted into an Internet address and is converted to a hostname using *gethostbyaddr*(3).

u_numunsuclog
This field contains a count of the number of unsuccessful login attempts to the account. This field is reset when a successful login to the account occurs.

u_unsuctty This field is a character string that identifies the name of the terminal or remote host associated with the last unsuccessful login attempt to the account.

u_maxtries This field specifies the maximum number of consecutive unsuccessful login attempts to the account that are permitted until the account is locked.

u_lock This flag field is used to administratively lock an account. A user cannot login to a locked account.

EXAMPLES

The following is an example of a typical protected password database file:

```
perry:u_name=perry:u_id#101:\
        :u_pwd=aZXtu1kmSpEzm:\
        :u_minchg#0:u_succhg#653793862:u_unsucchg#622581606:u_nullpw:\
        :u_suclog#671996425:u_suctty=tty1:\
        :u_unsuclog#660768767:u_unsuctty=tty1:\
        :u_maxtries#3:chkent:
```

This protected password database file is for the user **perry**. The user id for **perry** is 101. This value must match the **/etc/passwd** entry for this user. The account has a password and its encrypted form is specified by the **u_pwd** field.

The database file specifies a minimum password change time of 0, indicating the password can be changed at any time. Furthermore, the account is permitted to have a null password (**u_nullpw**). The account has a maximum consecutive unsuccessful login threshold of 3 attempts indicating that the account will be locked after three failed attempts (**u_maxtries**). The remaining fields provide account information such as the last successful and unsuccessful password change times as well as the last successful and unsuccessful login times and terminal names.

AUTHOR

SecureWare Inc.

SEE ALSO

login(1), acceptable_password(3), getprpwent(3), tod(3), authcap(4), default(4), users(4)

NOTES

The *getprpwent*(3) routines are used to parse the protected password database files into a structure that can used by programs. A flag in the structure indicates whether a particular field in the structure and hence the field is defined. System default values are also provided in the structure. These values are derived from the **/tcb/files/auth/system/default** field and can be used by programs in the absence of a user specific value.

NAME

publickey - public key database

SYNOPSIS

etc/publickey

DESCRIPTION

/etc/publickey is a local public key database that is used for secure RPC. The **/etc/publickey** file can be used in conjunction with or instead of other publickey databases, including the NIS publickey map and the NIS+ publickey map. Each entry in the database consists of a network user name (which may refer to either a user or a hostname), followed by the user's public key (in hex notation), a colon, and then the user's secret key encrypted with a password (also in hex notation).

The **/etc/publickey** file contains a default entry for **nobody**.

AUTHOR

publickey was developed by Sun Microsystems, Inc.

SEE ALSO

chkey(1), newkey(1M), getpublickey(3N), nsswitch.conf(4).

p

NAME

queuedefs - queue description file for at, batch, and crontab

SYNOPSIS

/var/adm/cron/queuedefs

DESCRIPTION

The **queuedefs** file describes the characteristics of the queues managed by **cron** (see *cron*(1M)). Each non-comment line in this file describes one queue. The format of the lines are as follows:

q.[*njob*j][*nice*n][*nwait*w]

The fields in this line are:

q The name of the queue, such that **a** is the default queue for jobs started by **at** (see *at*(1)), **b** is the queue for jobs started by **batch** (see *at*(1)), and **c** is the queue for jobs run from a **crontab** file (see *crontab*(1)). Queue names **d** through **y** designate user-defined queues.

njob The maximum number of jobs that can be run simultaneously in that queue. Although any number can be specified here, the total number of jobs that can be run on all the queues is limited to 100.

nice The **nice** value to give to all jobs in that queue that are not run with a user ID of super-user (see *nice*(1)). The default value is 2.

nwait The number of seconds to wait before rescheduling a job that was deferred because more than *njob* jobs were running in that job's queue, or because more than 100 jobs were running in all the queues (see *njob* above).

EXAMPLES

Consider the following **queuedefs** file:

```
a.4j1n
b.2j2n90w
```

The file is interpreted as follows:

a.4j1n The **a** queue, for **at** jobs (see *at*(1)), can have up to 4 jobs running simultaneously, and those jobs will be run with a **nice** value of 1.

Since no *nwait* value is given, if a job cannot be run because too many other jobs are running, **cron** will wait 60 seconds before trying again to run it (see *cron*(1M)).

b.2j2n90w The **b** queue, for **batch** jobs (see *at*(1)), can have up to 2 jobs running simultaneously. Those jobs will be run with a **nice** value of 2. If a job cannot be run because too many other jobs are running, **cron** will wait 90 seconds before trying again to run it.

All other queues can have up to 100 jobs running simultaneously. They will be run with a **nice** value of 2, and if a job cannot be run because too many other jobs are running, **cron** will wait 60 seconds before trying again to run it.

SEE ALSO

at(1), nice(1), crontab(1), cron(1M), proto(4).

STANDARDS CONFORMANCE

queuedefs: SVID2, SVID3

NAME

rc.config, rc.config.d/ - files containing system configuration information

SYNOPSIS

/etc/rc.config

/etc/rc.config.d/*

/etc/TIMEZONE

DESCRIPTION

The system configuration used at startup is contained in files within the directory **/etc/rc.config.d**. The file **/etc/rc.config** sources all of the files within **/etc/rc.config.d** and **/etc/TIMEZONE** and exports their contents to the environment.

/etc/rc.config

The file **/etc/rc.config** is a script that sources all of the **/etc/rc.config.d/*** scripts, and also sources **/etc/TIMEZONE**. To read the configuration definitions, only this file need be sourced. This file is sourced by **/sbin/rc** whenever it is run, such as when the **init** command is run to transition between run states. Each file that exists in **/etc/rc.config.d** is sourced, without regard to which startup scripts are to be executed.

/etc/rc.config.d

The configuration information is structured as a directory of files, rather than as a single file containing the same information. This allows developers to create and manage their own configuration files here, without the complications of shared ownership and access of a common file.

/etc/rc.config.d/* Files

This is where files containing configuration variable assignments are located.

Configuration scripts must be written to be read by the POSIX shell, and not the Bourne shell, **ksh**, or **csh**. In some cases, these files must also be read and possibly modified by **sd** control scripts or the *sam* program. See *sd*(4) and *sam*(1M). For this reason, each variable definition must appear on a separate line, with the syntax:

variable=*value*

No trailing comments may appear on a variable definition line. Comment statements must be on separate lines, with the **#** comment character in column one. This example shows the required syntax for configuration files:

```
# Cron configuration. See cron(1M)
# Cron configuration. See cron(1M)
#
# CRON: Set to 1 to start cron daemon
#
CRON=1
```

Configuration variables may be declared as array parameters when describing multiple instances of the variable configuration. For example, a system may contain two network interfaces, each having a unique IP address and subnet mask (see *ifconfig*(1M)). An example of such a declaration is as follows:

```
NET_CARDS=2
IP_ADDRESS[1]=15.1.55.2
SUBNET_MASK[1]=255.255.248.0

IP_ADDRESS[2]=15.1.55.3
SUBNET_MASK[2]=255.255.248.0
```

Note that there must be no requirements on the order of the files sourced. This means configuration files must not refer to variables defined in other configuration files, since there is no guarantee that the variable being referenced is currently defined. There is no protection against environment variable namespace collision in these configuration files. Programmers must take care to avoid such problems.

/etc/TIMEZONE

The file **/etc/TIMEZONE** contains the definition of the **TZ** environment variable. This file is required by POSIX. It is sourced by **/sbin/rc** at the same time the **/etc/rc.config.d/*** files are sourced.

r

SEE ALSO

rc(1M).

NAME

rcsfile - format of RCS files

DESCRIPTION

An RCS file is an ASCII file. Its contents are described by the grammar below. The text is free format, i.e., spaces, tabs and newline characters have no significance except in strings. Strings are enclosed by **@** symbols. If a string contains the **@** symbol, the symbol must be doubled.

The meta syntax uses the following conventions:

|	(bar) Separates alternatives.
{...**}**	(braces) Encloses optional phrases.
{...**}***	Encloses phrases that may be repeated zero or more times.
{...**}+**	Encloses phrases that must appear at least once and may be repeated.
<...**>**	Encloses nonterminals.

RCS File Grammar

Identifiers are case sensitive. Keywords are in lowercase only. The sets of keywords and identifiers may overlap.

```
<rcstext>   ::= <admin> {<delta>}* <desc> {<deltatext>}*

<admin>     ::= head    {<num>};
                access  {<id>}*;
                symbols {<id> : <num>}*;
                locks   {<id> : <num>}*; {strict ;}
                comment {<string>};

<delta>     ::= <num>
                date    <num>;
                author  <id>;
                state   {<id>};
                branches {<num>}*;
                next    {<num>};

<desc>      ::= desc    <string>

<deltatext> ::= <num>
                log     <string>
                text    <string>

<num>       ::= {<digit>{.}}+

<digit>     ::= 0 | 1 | ... | 9

<id>        ::= <letter>{<idchar>}*

<letter>    ::= A | B | ... | Z | a | b | ... | z

<idchar>    ::= Any printing ASCII character except space,
                tab, carriage return, newline, and <special>.

<special>   ::= ; | : | , | @

<string>    ::= @{any ASCII character, with "@" doubled}*@
```

RCS File Structure

The **<delta>** nodes form a tree. All nodes whose numbers consist of a single pair (e.g., 2.3, 2.1, 1.3, etc.) are on the trunk, and are linked through the **next** field in order of decreasing numbers. The **head** field in the **<admin>** node points to the head of that sequence (i.e., contains the highest pair).

All **<delta>** nodes whose numbers consist of $2n$ fields (n>=2) (e.g., 3.1.1.1, 2.1.2.2, etc.) are linked as follows. All nodes whose first $(2n)$-1 number fields are identical are linked through the **next** field in order of increasing numbers. For each such sequence, the **<delta>** node whose number is identical to the first

r

2(*n*-1) number fields of the deltas on that sequence is called the branchpoint. The **branches** field of a node contains a list of the numbers of the first nodes of all sequences for which it is a branchpoint. This list is ordered in increasing numbers.

EXAMPLES

```
                           head
                             |
                             |
                             v
                         ---------
   / \          / \      |       |      / \          / \
  /   \        /   \     |  2.1  |     /   \        /   \
 /     \      /     \    |       |    /     \     __/     \__
/1.2.1.3\    /1.3.1.1\   |       |   /1.2.2.2\   /1.2.2.1.1.1\
---------    ---------   ---------   ---------   -------------
    ^            ^           |           ^             ^
    |            |           |           |             |
    |            |           v           |             |
   / \           |       ---------      / \            |
  /   \          |       \  1.3  /     /   \           |
 /     \         ---------\     /     /     \-----------
/1.2.1.1\                  \   /     /1.2.2.1\
---------                   \ /      ---------
    ^                        |           ^
    |                        |           |
    |                        v           |
    |                    ---------       |
    |                    \  1.2  /       |
    ----------------------\     /---------
                           \   /
                            \ /
                             |
                             |
                             v
                         ---------
                         \  1.1  /
                          \     /
                           \   /
                            \ /
```

WARNINGS

RCS is designed to be used with text (ASCII) files only. Using RCS with nontext (binary) files results in data corruption.

AUTHOR

rcsfile was developed by Walter F. Tichy, Purdue University, West Lafayette, IN 47907. Revision Number: 3.0. Release Date: 83/05/11. Copyright 1982 by Walter F. Tichy.

SEE ALSO

ci(1), co(1), ident(1), rcs(1), rcsdiff(1), rcsmerge(1), rlog(1), rcsintro(5).

NAME

resolver - resolver configuration file

SYNOPSIS

/etc/resolv.conf

DESCRIPTION

The **resolver** is a set of routines in the C library (see *resolver*(3N)) that provide access to the Internet Domain Name System. The resolver configuration file contains information that is read by the resolver routines the first time they are invoked by a process. The file is designed to be human-readable, and contains a list of keywords with values that provide various types of resolver information.

If the only name server to be queried is on the local machine, then this file is not always necessary. The domain name could be determined from the host name (see *hostname*(1)), if it has been set as a fully qualified domain name.

Recognized configuration options include:

nameserver Internet (IP) address, in dot notation, of a name server that the resolver should query. Up to **MAXNS** (currently 3) name servers can be listed, one per keyword. If there are multiple servers, the resolver library queries them in the order listed. If no **nameserver** entries are present, the default is to use the name server on the local machine. (The algorithm used is: Try a name server; if the query times out, try the next and continue until all name servers have been tried, then repeat trying all the name servers until a maximum number of retries have been made).

domain Local domain name. Most queries for names within this domain can use short names relative to the local domain. If no **domain** entry is present, the domain is determined from the local host name returned by **gethostname()** (see *gethostname*(2)); the domain part is interpreted as everything after the first dot (**.**). Finally, if the host name does not contain a domain part, the root domain is assumed.

search Search list for host-name lookup. If the search option is not used the search list will contain only the the local domain name. The search list can be changed by listing the desired domain search path following the **search** keyword with spaces or tabs separating the names. Most resolver queries will be attempted using each component of the search path in turn until a match is found. Note that this process may be slow and generates a lot of network traffic if the servers for the listed domains are not local, and that queries time out if no server is available for one of the domains.

The search list is currently limited to six domains with a total of 256 characters.

The first domain in the search list must be the local domain for short names to work properly in various files (such as **.rhosts** and **inetd.sec**)

options Options allows certain internal resolver variables to be modified. The syntax is

options *option* ...

where currently the *option* supported is the following:

ndots:*n* Set a threshold for the number of dots which must appear in a name given to **res_query** (see *resolver*(3N)) before an initial absolute query will be made. The default for *n* is “1”, meaning that if there are any dots in a name, the name will be tried first as an absolute name before any search list elements are appended to it.

The **domain** and **search** keywords are mutually exclusive. If more than one instance of these keywords is present, the last instance overrides.

The **search** keyword of a system’s **resolv.conf** file can be overridden on a per-process basis by setting the environment variable **LOCALDOMAIN** to a space-separated list of search domains. The **options** keyword of a system’s **resolv.conf** file can be amended on a per-process basis by setting the environment variable **RES_OPTIONS** to a space separated list of resolver options as explained above under **options**.

The keyword and value must appear on a single line, and the keyword (e.g. **nameserver**) must start the line. The value follows the keyword, separated by white space.

r

Note that the resolver routine **res_init()** silently ignores errors when reading this file (see *resolver*(3N)).

EXAMPLES

A typical **resolv.conf** file resembles the following:

```
domain div.inc.com
nameserver 15.19.8.119
nameserver 15.19.8.197
```

WARNING

In order to reduce situations that may cause connections to unintended destinations, the administrator should carefully select which domains are put in the search list in the resolv.conf file. HP recommends that the possible domains for the search list be limited to those domains administered within your trusted organization. For more information on the security implications of search lists please see *RFC 1535*, located in **/usr/share/doc**.

AUTHOR

resolver was developed by the University of California, Berkeley.

FILES

/etc/resolv.conf Resolver configuration file.

SEE ALSO

named(1M), resolver(3N), gethostent(3N), hostname(5), RFC 1535

NAME

rmtab - local file system mount statistics

DESCRIPTION

File **/etc/rmtab** contains a record of all clients that mounted remote file systems from this machine. Whenever a remote **mount** is done, an entry is made in the **rmtab** file of the machine serving that file system. **umount** removes the entry of a remotely mounted file system. **umount -a** broadcasts to all servers that they should remove all entries from **rmtab** created by the sender of the broadcast message. The table is a series of lines of the following form:

hostname:*directory*

This table only preserves information between crashes, and is read only by **mountd** when it starts (see *mountd*(1M)). **mountd** keeps an in-core table to handle requests from commands such as **showmount** and **shutdown** (see *showmount*(1M) and *shutdown*(1M)).

WARNINGS

Although the **rmtab** table is close to the truth, it is not always totally accurate.

AUTHOR

rmtab was developed by Sun Microsystems, Inc.

FILES

/etc/rmtab

SEE ALSO

mount(1M), mountd(1M), showmount(1M), shutdown(1M).

r

NAME

rpc - rpc program number data base

SYNOPSIS

/etc/rpc

DESCRIPTION

File **/etc/rpc** contains user-readable names that can be used in place of RPC program numbers. Each line has the following information:

- Name of server for the RPC program
- RPC program number
- Aliases

Items are separated by any number of blanks and tab characters. A **#** anywhere in the file indicates a comment extending to the end of that line.

EXAMPLES

Here is an example of an **/etc/rpc** file:

```
#
# rpc 12.0 89/09/25
#
rstatd          100001      rstat rup perfmeter
rusersd         100002      rusers
nfs             100003      nfsprog
ypserv          100004      ypprog
mountd          100005      mount showmount
ypbind          100007
walld           100008      rwall shutdown
yppasswdd       100009      yppasswd
etherstatd      100010      etherstat
rquotad         100011      rquotaprog quota rquota
sprayd          100012      spray
selection_svc   100015      selnsvc
dbsessionmgr    100016      unify netdbms dbms
rexd            100017      rex remote_exec
office_auto     100018      alice
```


AUTHOR

rpc was developed by Sun Microsystems, Inc.

FILES

/etc/rpc

SEE ALSO

getrpcent(3C).

NAME

sccsfile - format of SCCS file

DESCRIPTION

An SCCS file is an ASCII file consisting of six logical parts:

checksum	Sum of all characters in the file except the first line.
delta table	Contains information about each delta.
user names	Login names and/or numerical group IDs of users who are allowed to add deltas.
flags	Definitions of internal keywords. *comments* Arbitrary descriptive information about the file.
body	Actual text lines intermixed with control lines.

Throughout an SCCS file there are lines beginning with the ASCII SOH (start of heading) character (octal 001). This character is hereafter referred to as the **control character** and is represented graphically as **@**. Any line described below that is not depicted as beginning with the control character is prevented from beginning with the control character. All lines in the SCCS file are limited to **BUFSIZ** (defined in <**stdio.h**>) characters in length.

Entries of the form *DDDDD* represent a five-digit string (a number between 00000 and 99999).

The following describes each logical part of an SCCS file detail:

Checksum The checksum is the first line of an SCCS file. The form of the line is:

```
@hDDDDD
```

The value of the checksum is the sum of all characters except those in the first line. The **@h** sequence provides a **magic number** consisting of the two bytes 0x01 and 0x68. (Other versions of UNIX-like operating systems usually use this same value but it may be displayed or documented as a single number with a different byte order.)

Delta table The delta table consists of a variable number of entries of the form:

```
@s DDDDD/DDDDD/DDDDD
@d <type> <SID> yr/mo/da hr:mi:se <pgmr> DDDDD DDDDD
@i DDDDD ...
@x DDDDD ...
@g DDDDD ...
@m <MR number>
 .
 .
 .
@c <comments> ...
 .
 .
 .
@e
```

The first line (**@s**) contains the number of lines inserted/deleted/unchanged, respectively. The second line (**@d**) contains the type of the delta (currently, normal: **D**, and removed: **R**), the *SID* (SCCS ID) of the delta, the date and time when the delta was created, the login name corresponding to the real user ID at the time the delta was created, and the serial numbers of the delta and its predecessor, respectively.

The **@i**, **@x**, and **@g** lines contain the serial numbers of deltas included, excluded, and ignored, respectively. These lines are optional.

The **@m** lines (optional) each contain one MR (modification request) number associated with the delta; the **@c** lines contain comments associated with the delta.

The **@e** line ends the delta table entry.

User names The list of login names and/or numerical group IDs of users who are allowed to add deltas to the file, separated by new-lines. The lines containing these login names and/or numerical group IDs are surrounded by the bracketing lines **@u** and **@U**. An empty list allows anyone to make a delta. Any line starting with a **!** prohibits the specified group or user from making deltas.

Flags — Keywords used internally (see *admin*(1) for more information on their use). Each flag line takes the form:

@f *<flag>* *<optional text>*

The following flags are defined:

@f t *<type of program>*
@f v *<program name>*
@f i *<keyword string>*
@f b
@f m *<module name>*
@f f *<floor>*
@f c *<ceiling>*
@f d *<default-sid>*
@f n
@f j
@f l *<lock-releases>*
@f q *<user defined>*
@f z *<reserved for use in interfaces>*

The above flags function as follows:

- **t** Defines the replacement for the **%Y%** identification keyword.
- **v** Controls prompting for MR numbers in addition to comments. If the optional text is present, it defines an MR number-validity checking program.
- **i** Controls the warning/error aspect of the "No id keywords" message. When the **i** flag is not present, the message is only a warning; when the **i** flag is present, this message causes a fatal error (a *get* on the file fails, or the delta is not made).
- **b** When the **b** flag is present, the **-b** keyletter can be used on the *get* command to cause a branch in the delta tree.
- **m** Defines the first choice for the replacement text of the **%M%** identification keyword.
- **f** Defines the "floor" release; the release below which no deltas can be added.
- **c** Defines the "ceiling" release; the release above which no deltas can be added.
- **d** Defines the default SID to be used when none is specified on a *get* command.
- **n** Causes *delta* to insert a "null" delta (a delta that applies *no* changes) in those releases that are skipped when a delta is made in a *new* release (such as, when delta 5.1 is made after delta 2.7, releases 3 and 4 are skipped). The absence of the **n** flag causes skipped releases to be completely empty.
- **j** Causes *get* to allow concurrent edits of the same base SID. See *admin*(1) for restrictions.
- **l** Defines a *list* of releases that are *locked* against editing (*get*(1) with the **-e** keyletter).
- **q** Defines the replacement for the **%Q%** identification keyword.
- **z** Used in certain specialized interface programs.

Comments — Arbitrary text is surrounded by the bracketing lines **@t** and **@T**. The comments section typically contains a description of the file's purpose.

Body — Consists of text lines and control lines. Text lines do not begin with the control character; control lines do. There are three kinds of control lines:

Type	*Represented By:*
insert	**@I** *DDDDD*
delete	**@D** *DDDDD*

S

end **@E** *DDDDD*

The digit string is the serial number corresponding to the delta for the control line.

WARNINGS

SCCS files can be any length, but the number of lines in the text file itself cannot exceed 99 999 lines.

SEE ALSO

admin(1), delta(1), get(1), prs(1).

S

NAME

sd(4) - all objects that Software Distributor (SD) uses, their attributes and storage formats

DESCRIPTION

Remarks

- SD-UX commands are included with the HP-UX operating system and manage software on the *local* host only.
- To install and manage software simultaneously on multiple *remote* hosts (including HP-UX®, other UNIX® platforms, Windows NT®, and PCs) from a central controller, you must purchase the *HP OpenView Software Distributor* which provides extended software management capabilities. Information specific *only* to the OpenView product is marked with a heading similar the following:

  The following information applies to HP OpenView Software Distributor only.

Command Overview

The SD commands create, install, distribute and manage software *objects* (bundles, products, subproducts and filesets). In addition, they define and manage other objects in support of the software administration tasks which users perform. This manual page describes the SD software object classes, their attributes, and the file formats used to store their definitions.

For an overview of all SD commands, see the *sd*(5) manual page by typing: **`man 5 sd`**

Layout Version

The objects described here conform to *layout_version 1.0* of the *IEEE Standard 1387.2: Software Administration (POSIX)*. The previous SD layout_version 0.8 is also supported. For more details, see *swpackage*(4) or the **`layout_version`** option in *sd*(5).

OBJECT CLASSES

The SD object classes are:

host A machine at which software is installed, will be installed, or is being managed. A host contains one or more *roots* (installed filesystems) and zero or more *depots*.

depot A directory location which contains software *products* or *bundles* that are available for installation. It is a customizable source of software used for direct installation. It can also represent a distribution medium (e.g. tape or CD-ROM) which contains products or bundles available for installation. *Depot* corresponds to the **distribution** class defined in POSIX.

media Vehicle for software delivery. When a depot is located on one or more media in **`layout_version=1.0`**, the unique sequence number identifying each medium is in the *media* class.

root A set of installed software objects, usually the operational software installed in the primary root filesystem, "/". It also represents the set of software objects installed into an alternate root directory. Root corresponds to the **installed_software** class defined in POSIX.

vendor The vendor who packaged and distributed a *product* or *bundle*. It is an optional component of a *product* or a *bundle*.

category A classification for a product or bundle, such as "systems_management," "desktop," or "patch."

bundle A bundle is a way of encapsulating products, subproducts and filesets into a single software object. More than one bundle can contain the same software objects. A bundle can be thought of as a particular "configuration" of software. It is a convenient way to group software objects together for easy selection. Bundle is NOT a superset of product.

product A software object which vendors package and distribute, and which users purchase and install. A product contains one or more *filesets* and zero or more *subproducts*. A product can also contain zero or more *control_files*.

subproduct
A subset or partitioning of a software product. It is an optional component of a *product*. and contains one or more *filesets*.

S

fileset A grouping of one or more *files* contained in a *product* or *sub-product*. It groups a subset of a product's files into a manageable unit. A fileset can also contain zero or more *control_files*.

file The actual files that make up a *fileset* that get installed, configured, and removed.

control_files
The scripts developed by vendors to perform *product*- or *fileset*-specific operations during various software management tasks. Often called *control_scripts*.

OBJECT ATTRIBUTES

The following tables summarize the valid *attributes* for each software object class. A subset of these attributes can be defined for an object when creating products or bundles with **swpackage**. See *swpackage*(4) for details on this subset.

The attribute value types are defined in the next section, "VALUE TYPES".

Host Attributes

Attribute	Value Type	Size	Example
machine_type	uname_string	32	9000/720
name	one_line_string	64	newdist.fc.hp.com
os_name	uname_string	32	HP-UX
os_release	uname_string	32	A.09.01
os_version	uname_string	32	C
contained depots	list of depot objects		
contained roots	list of root directories		

machine_type
The host's machine and architecture designation. (**uname -m** field).

name
The official name of the network host.

os_name
The host's operating system name. (**uname -s**).

os_release
The host's operating system release. (**uname -r**).

os_version
The host's operating system version. (**uname -v**).

contained depots
The depots registered at the host.

contained roots
The root filesystems registered on this system.

S

Depot Attributes

Attribute	Value Type	Size	Example
copyright	multi_line_string	8K	"This depot ..."
data_model_revision	revision_string	64	2.40
description	multi_line_string	8K	"This depot ..."
dfiles	tag_string	64	dfiles
layout_version	revision_string	64	1.0
mod_date	one_line_string	64	Tue Jun 22 12:52:09 1997
mod_time	unsigned_integer		740774837
name_max	unsigned_integer		255
number	one_line_string	64	B2358-13601
path_max	unsigned_integer		1023
pfiles	tag_string	64	pfiles
tag	tag_string	64	APPLICATIONS_CD
title	one_line_string	256	Applications Software
uuid	one_line_string	64	25CA7C86-6F0C-9353
contained bundles	list of bundle objects		
contained products	list of product objects		
contained media	media object		
contained vendors	vendor objects		
contained categories	category objects		

copyright
The copyright information for the depot or tape.

data_model_revision
The HP specific format revision used to store the depot definition.

description
The multi-paragraph description of the distribution depot/tape.

dfiles
The name of a directory that contains any attributes that must be stored as files.

layout_version
The version of the IEEE Standard 1387.2
(**1.0** or **0.8**) to which the HP-specific data_model_revision conforms.

mod_date
The string format of the **mod_time**.

mod_time
The time of the last operation performed on the depot.

name_max
The maximum length of file basenames in the depot.

number
The part or manufacturing number of the depot/tape.

path_max
The maximum length of file pathnames in the depot.

pfiles
The name of a directory that contains any product **control_files** or any product attributes that must be stored as files.

tag The identifier (short name) for the distribution depot/tape.

title
The full name (one-line description) of the distribution depot/tape.

uuid
The depot's Universal Unique Identifier (UUID).

contained bundles
The bundles available from the depot.

contained products
The products available from the depot.

contained media
The object defining the sequence number used to identify each medium.

contained vendors
The objects defining a vendor object that is associated with subsequent bundle and product objects that define a **vendor_tag** attribute.

contained categories
The objects defining a vendor object that is associated with subsequent software objects and define a **category_tag** attribute.

Media Attributes

Generated by **swpackage.**

Attribute	Value Type	Size	Example
sequence_number	one_line_string	64	1

sequence_number
For a multiple tape distribution, this attribute defines the unique **sequence_number** of each medium.

Root Attributes

Attribute	Value Type	Size	Example
data_model_revision	revision_string	64	2.40
description	multi-line_string	2048	"This root is ..."
dfiles	tag_string	64	dfiles
layout_version	revision_string	64	1.0
mod_date	one_line_string	64	Mon Jun 14 13:01:19 1997
mod_time	unsigned_integer		740774837
path	one_line_string	256	/xx/xx/xx
pfiles	tag_string	64	pfiles
root_type	one_line_string	256	shared
contained bundles	list of bundle objects		
contained products	list of product objects		
contained vendors	vendor objects		
contained categories	category objects		

S

data_model_revision
The HP specific format revision used to store the root definition.

description
A multi-line description of the root.

layout_version
The version of the IEEE Standard 1387.2 (**1.0** or **0.8**) to which the HP-specific data_model_revision conforms.

mod_date
The string format of the **mod_time**.

mod_time
The time of the last operation performed on the root.

path
The path to the root.

root_type
The type of root: shared, private or none.

contained bundles
The bundles installed into the root.

contained products
The products installed into the root.

contained vendor
The object defining a vendor object that is associated with subsequent bundle and product objects that define a **vendor_tag** attribute.

contained category
The object defining a vendor object that is associated with subsequent software objects that define a **category_tag** attribute.

Vendor Attributes

Attribute	Value Type	Size	Example
description	multi_line_string	8K	"This vendor ..."
title	one_line_string	256	Hewlett-Packard Company
tag	tag_string	64	HP
uuid	one_line_string	64	1234567-CDEF-0123-4569

description
The multi-paragraph description of the vendor.

tag The identifier (short name) for the vendor. Used to associate a vendor object with subsequent product or bundle objects having a **vendor_tag** attribute of the same value.

title
The full name (one-line description) for the vendor.

uuid
The vendor's Universal Unique Identifier (UUID).

When listing the attributes of a vendor associated with the specified product or bundle using **swlist**, the option **-a vendor** lists all of the vendor attributes. The option **-a vendor.***attribute* can be used to list specific vendor attributes (e.g. **-a vendor.title**).

Category Attributes

Attribute	Value Type	Size	Example
description	multi_line_string	8K	normal patches
revision	revision_string	64	0.0
tag	tag_string	64	normal_patch
title	one_line_string	256	patches for normal use

description
A more detailed description of the category.

tag A short name identifying the category. Each category must have a unique tag. This attribute has no default value. The category tag **patch** is reserved. When **is_patch** is set to **true** for a software object, a built-in **category_tag** attribute of value **patch** is automatically included.

title
A longer name of the category used for presentation purposes.

revision
Determines which category object definition to maintain in a depot when a definition being installed or copied does not match a definition already in the depot with the same **category_tag**. The category definition with the higher revision is maintained.

S

Bundle and Product Attributes

NOTE:

- Attributes marked with a **+** apply only to the **product** class.
- Attributes marked with a **-** apply only to the **bundle** class.
- Attributes marked with a ***** determine the uniqueness of a product or bundle object. Their values may also be of the type **version_component** when used in a version component of a software specification.

Attribute	Value Type	Size	Example
+ all_filesets	one-line list of tag_string values		commands agent data man
* architecture	one_line_string*	64	HP-UX_A.09.00_800
category_tag	tag_string	64	normal
- contents	repeatable list of software_specs	8K	foo.bar,r=1.0,a=,v=HP x.y,r=2.0,a=,v=
control_directory	path_string	255	SD
copyright	multi_line_string	8K	"This product ..."
create_date	one_line_string	64	Mon Jun 14 13:01:19
create_time	unsigned_integer		740084479
data_model_revision	revision_string	64	2.40
description	multi_line_string	8K	"This product ..."
directory	path_string	1024	
install_date	one_line_string	16	199802241212.34
install_source	one_line_string	1024	zook.com:/depot
install_type	one_line_string	16	physical
instance_id	tag_string	64	1
+ is_locatable	boolean	8	true
is_patch	boolean	8	true
+ job_file	one_line_string	256	<pcjob
layout_version	revision_string	64	1.0
+* location	path_string	1024	/
machine_type	uname_string	64	9000/7* | 9000/8*
mod_date	one_line_string	64	Mon Jun 14 13:01:19 1997
mod_time	unsigned_integer		740084479
number	one_line_string	64	J2326AA
os_name	uname_string	64	HP-UX
os_release	uname_string	64	?.09.*
os_version	uname_string	64	[A-Z]
+ postkernel (future)	path_string	255	/usr/bin/kernel_build
+ qualifier	one_line_string	64	"My product"
+ readme	multi_line_string	1024K	< README
* revision	revision_string	64	2.0
share_link	one_line_string	256	"sbin"
size	unsigned_integer		14638872
software_spec	software_spec		SD,r=2.0
* tag	tag_string	64	SD
title	one_line_string	256	Software Distributor
* vendor_tag	tag_string	64	HP
+ contained control scripts	list of control script objects		
+ contained filesets	list of fileset objects		
+ contained subproducts	list of subproduct objects		

all_filesets
All the filesets originally packaged into the product.

architecture
The target system(s) on which the product will run.

category_tag
A repeatable tag-based attribute identifying a set of categories of which the software object is a member. This is used as a selection mechanism and can be used independent of patches. The default value is an empty list or **patch** if the **is_patch** attribute is set to **true**. The category tag **patch** is reserved.

Like **vendor_tag**, this attribute associates this product or bundle with a category object containing additional information about the category (for example, a one-line *title* definition and a *description* of the category).

contents
The list of fully qualified software_specs for a bundle. (Applies to bundles only).

control_directory
The relative pathname to the product catalog directory within the depot/root catalog that contains **control** scripts for the product.

copyright
The copyright information for the product.

create_date
The string format of the **create_time**.

create_time
The time at which the bundle or product was created.

data_model_revision
The format revision used to store the product definition.

directory
The default pathname in which the product's files will be installed.

description
The multi-paragraph description of the product.

install_date
The installation date of the product or bundle.

install_source
The source from which the product was installed.

install_type
The type of installation - "physical" or "NFS_mount".

instance_id
The secondary identifier for products which have the same **tag** (or truncated tag) value.

is_locatable
Defines whether (or not) the product can be installed into an arbitrary directory.

S

is_patch
Identifies a software object as a patch. The default value is **false**. When set to **true**, a built-in **category_tag** attribute of value **patch** is automatically included.

job_file
For HP OpenView Software Distributor PC products, this indicates the existence of the **pcjob***control_file*.

layout_version
The version of the IEEE Standard 1387.2 (**1.0** or **0.8**) to which the HP-specific data_model_revision conforms.

location
The installed pathname of the product.

machine_type
The machine(s) on which the product will run.

mod_date
The string format of the **mod_time**.

mod_time
The time of the last operation performed on the product.

number
The part or order number for the product.

os_name
The operating system(s) on which the product will run.

os_release
The operating system release(s) on which the product will run.

os_version
The operating system versions(s) on which the product will run.

postkernel
(Not yet implemented.)The path to a script that is run after kernel filesets are installed.

qualifier
A user-specified name that identifies a product or set of product versions.

readme
The most recent information for the product.

revision
The revision (release number, version number) of the product.

share_link
Sharing point(s) for product.

size
The size of the product in bytes.

software_spec
The fully qualified software specification for the bundle or product. This attribute can be used to identify a unique bundle or product.

tag The identifier (short name) for the product.

title
The full name (one-line description) of the product.

vendor_tag
Associates this product or bundle with a vendor object containing additional attributes describing the vendor. The vendor object must have a matching **tag** attribute.

contained control scripts
The scripts defined for the product.

contained filesets
The fileset defined for the product.

contained subproducts
The subproducts defined for the product.

Subproduct Attributes

Attribute	Value Type	Size	Example
contents	repeatable list of tag_string values		commands data man
description	multi_line_string	8K	"This subproduct ..."
size	unsigned_integer		14638872
software_spec	software_specification		Networking.Run,r=1.0
tag	tag_string	64	Manager
title	one_line_string	256	Management Utilities

contents
A list of the filesets that make up the subproduct.

description
The multi-paragraph description of the subproduct.

size
The size of the subproduct in bytes.

software_spec
The full software specification for the subproduct. This attribute can be used to identify a unique subproduct.

tag The identifier (short name) for the subproduct.

title
The full name (one-line description) of the subproduct.

S

Fileset Attributes

Attribute	Value Type	Size	Example
ancestor	repeatable list of software specs		oldprod.fs
applied_patches	software_spec		product.fileset,version
architecture	one_line_string	64	HP-UX_9.00_700/800
category_tag	tag_string	64	patch_normal
control_directory	path_string	255	SD
corequisites	repeatable list of software_specs		SD.man
create_date	one_line_string	64	Mon Jun 14 13:01:19
create_time	unsigned_integer		740084479
data_model_revision	revision_string	64	2.40
description	multi_line_string	8K	"This fileset ..."
exrequisites (future)	repeatable list of software_specs		SD.man
install_date	one-line string	16	199402241414.34
install_source	one_line_string	1024	zook.com:/depot
instance_id	tag_string		1
is_kernel	boolean	8	false
is_locatable (future)	boolean	8	true
is_packaged_in_place	boolean	8	false
vis_patch	boolean	8	true
is_reboot	boolean	8	false
is_secure	boolean	8	false
is_sparse	boolean	8	false
location (future)	path_string	1024	/
machine_type	uname_string	64	9000/[78]*
media_sequence_number	unsigned_integer		1
mod_date	one_line_string	64	Mon Jun 14 13:01:19 1997
mod_time	unsigned_integer		740084479
os_name	uname_string	64	HP-UX
os_release	uname_string	64	?.09.*
os_version	uname_string	64	?
patch_state	patch_state_string	16	applied
pose_as_os_name	uname string	64	HP-UX:64
pose_as_os_release	uname string	64	B.10.30
prerequisites	repeatable list of software_specs		SD.agent
revision	revision_string	64	2.15
size	unsigned_integer		14638872
state	state_enumeration		corrupt
software_spec	software_spec		Networking.Run,r=1.0
superseded_by	software_spec		product.fileset,version
supersedes	software_spec		product.fileset, fr=revision
tag	tag_string	64	commands
title	one_line_string	256	Commands
contained control scripts	list of control script objects		
contained files	list of file objects		

ancestor

A list of filesets that will match the current fileset when installed on a target system if the **match_target** installation option is specified. Also determines the base to which a patch is applied. (Note that an **ancestor** attribute is not the same as a **corequisite**, **exrequisite**, or **prerequisite**.)

applied_patches

Determines the list of patches that have been applied to a base fileset. This attribute applies to installed base (non-patch) software only.

S

architecture
Describes the target system(s) on which the fileset will run if filesets for multiple architectures are included in a single product. Provides a human-readable summary of the four *uname***(1)** attributes which define the exact target system(s) the fileset supports. Most filesets do not include an architecture; only a product architecture need be defined.

category_tag
A repeatable tag-based attribute identifying a set of categories of which the software object is a member. This is used as a selection mechanism and can be used independent of patches. The default value is an empty list or **patch** if the **is_patch** attribute is set to **true**. The category tag **patch** is reserved.

Like **vendor_tag**, this attribute can be used as a pointer to a category object that contains additional information about the category (for example, a one-line *title* definition and a *description* of the category).

control_directory
The relative pathname to the fileset catalog directory in the product catalog that contains the control_scripts for the fileset.

corequisites
A list of dependencies on software that must be installed before this software is run.

data_model_revision
The format revision used to store the fileset definition.

description
The multi-paragraph description of the fileset.

exrequisites
(Not yet implemented.) A list of dependencies on software that may not be installed when this software is installed.

install_date
The date the fileset was installed.

install_source
The source from which the product was installed.

instance_id
The secondary identifier for filesets which have the same **tag** (or truncated tag) value.

is_kernel
Defines whether or not the fileset contains kernel files.

is_locatable
(Not yet implemented.) Defines whether or not the fileset can be installed into an arbitrary directory.

S

is_packaged_in_place
For a fileset within a depot, this attribute defines whether or not the fileset is packaged in place. If **true**, then the fileset's contents are actually references to the original source files used to construct the fileset.

is_patch
Identifies a software object as a patch. The default value is **false**. When set to **true**, a built-in **category_tag** attribute of value **patch** is automatically included.

is_reboot
Defines whether or not the fileset requires a reboot after install.

is_secure
Defines whether or not the fileset is encrypted on a CD-ROM media, requiring a codeword to install.

is_sparse
Indicates that a fileset contains only a subset of files in the base (ancestor) fileset and that the contents are to be merged with the base fileset. The default value is **false**. If the **is_patch** attribute is **true**, **is_sparse** is also set to **true** for the fileset, although it can be forced to false.

location
(Not yet implemented.) The installed pathname of the fileset.

machine_type
Defines the machine(s) on which the product will run if a fileset architecture has been defined. (If not specified, **swpackage** assigns a value of "*", meaning the fileset runs on all machines.) If there are multiple machine platforms, use wildcards or the '|' character to identify them. This attribute should pattern match to the value of **uname -m** on the supported target machine(s).

media_sequence_number
For a multiple tape distribution, defines the tape on which the fileset is archived.

mod_date
The string format of the **mod_time**.

mod_time
The time of the last operation performed on the fileset.

os_name
Defines the operating system(s) on which the product will run if a fileset architecture has been defined. (If not specified, **swpackage** assigns a value of "*", meaning the fileset runs on all operating systems.) If there are multiple operating systems, use wildcards or the '|' character to identify them. This attribute should pattern match to the value of **uname -s** on the supported target system(s).

os_release
Defines the operating system release(s) on which the product will run if a fileset architecture has been defined. (If not specified, **swpackage** assigns a value of "*", meaning the fileset runs on all releases.) If there are multiple operating system releases, use wildcards or the '|' character to identify them. This attribute should pattern match to the value of **uname -r** on the supported target system(s).

os_version
Defines the operating system version(s) on which the product will run if a fileset architecture has been defined. (If not specified, **swpackage** assigns a value of "*", meaning the fileset runs on all versions.) If there are multiple operating system versions, use wildcards or the '|' character to identify them. This attribute should pattern match to the value of **uname -v** on the supported target system(s).

patch_state
Lists the current state of an installed patch, either **applied, committed,** or **superseded.** Applies to installed patches only.

pose_as_os_name
Overrides the existing **os_name** *uname* attribute of any target to which the given fileset is being installed. Used for filesets that define a new OS.

pose_as_os_release
Overrides the existing **os_version** *uname* attribute of any target to which the given fileset is being installed. Used for filesets that define a new OS.

prerequisites
A list of dependencies on software that must be installed before this software can be installed.

revision
The revision (release number, version number) of the fileset.

size
The size of the fileset in bytes.

state
The current state of the fileset (resulting from the most recent operation performed on it).

software_spec
The full software specification for the fileset. Used to identify a unique fileset.

superseded_by
Lists what patch superseded this patch. Applies to installed patches only.

S

supersedes
Used when a patch is replaced by (or merged into) a later patch. The attribute indicates which previous patches are replaced by the patch being installed or copied. This attribute value is a list of software specifications of other patches that this patch "supersedes".

tag The identifier (short name) for the fileset.

title
The full name (one-line description) of the fileset **tag** (or truncated tag) value.

contained control scripts
The scripts defined for the fileset.

contained files
The files defined for the fileset.

Control_File Attributes

Attribute	Value Type	Size	Example
interpreter	path_string		/usr/package/scripts
path	path_string	255	checkinstall
result	result_enumeration		success
tag	tag_string	16	checkinstall

Control_files can be defined for filesets and products.

interpreter
The name of an interpreter used to execute *control_files*. Currently, only a value of **sh** is supported for this attribute. Control scripts can still define their own interpreter in the first line of the script.

path
The relative pathname to the *control_script* in the *control_directory*.

result
The result obtained from executing the *control_script*.

tag The identifier (short name) for the *control_script*.

File Attributes

Attribute	Value Type	Size	Example
cksum	unsigned_integer		18355158
compressed_cksum	unsigned_integer		false
compressed_size	unsigned_integer		false
compression_state	compression_enumeration		compressed
compression_type	tag_string	64	gzip
file_attrs	unsigned_integer		32
gid	unsigned_integer		0
group	tag_string	64	sys
is_volatile	boolean	8	false
link_source	path_string	1024	/usr/sbin/swinstall
mode	unsigned integer		04555
mtime	unsigned_integer		740084479
owner	tag_string	64	root
path	path_string	255	/usr/sbin/swpackage
revision	revision_string	64	1 .3
size	unsigned_integer		2494464
source	path_string	1024	/mfg/sbin/swinstall
type	file_type_enumeration		f
uid	unsigned_integer		0

cksum
The 32-bit checksum of the file.

compressed_cksum
The checksum of the compressed file.

compressed_size
The size of the compressed file.

compression_state
The compression state of a file (compressed or not).

compression_type
The type of compression used to store the file.

file_attrs
The DOS file attributes (of a PC file).

gid The GID of the file's owner.

group
The name of the file's group.

is_volatile
Defines whether the file can be modified or removed.

link_source
The path to which a hard or symbolic link points.

mode
The file permission mode.

mtime
The last modification time of the file.

owner
The name of the file's owner.

path
The full pathname to the file.

revision
The *what*(1) or *ident*(1) revision of the file.

S

size
The size of the file in bytes.

source
The full path to the source file for files which have been "packaged in place"; see *swpackage*(1M).

type
The type of file.

uid The UID of the file's owner.

VALUE TYPES

The value for each attribute must be of a specific type. The types are:

boolean
Maximum length: 8 bytes
Examples: true, false

One of the following values:
true
false

compression_enumeration
Maximum length: none

One of the following values:
compressed
uncompressed
not_compressible

file_type_enumeration
Maximum length: none

One of the following values:
a archive
b blank special device
c character special device
d directory
f file
h hard link
p named pipe (FIFO)
s symbolic link

multi_line_string
Maximum length: 8K (1 Mbyte for **readme**)

Multi-line strings support all **isascii()** characters. They represent one or more paragraphs of text. They are usually specified in-line, surrounded by double-quotes. (The product **readme** is stored in a file, and specified using the "**<** *filename*" format).

one_line_string
Maximum length: 256 bytes
Examples: Hewlett-Packard Company

One-line strings support a subset of **isascii()** characters only.
No **isspace()** characters, except for space and tab, are allowed.

patch_state_string
Maximum length: 16

One of the following values:
applied
committed
superseded

path_string
Maximum length: 255 bytes for tapes, 1024 bytes for depots
Examples: /usr, /mfg/sd/scripts/configure

An absolute or relative path to a file.

S

result_enumeration

Maximum length: none

One of the following values:
none
error
warning
success

revision_string

Maximum length: 64 bytes
Examples: 2.0, A.09.00

Revision strings contain zero or more dot-separated one_line_strings (above).

software_specification

Maximum length: none
Examples: SD.agent or SD,r=2.0,a=HP-UX_A.09.00_800

Software specifications specify software in dependencies, ancestors and other attributes, as well as command line selections. The next section describes software specification syntax.

state_enumeration

Maximum length: none

One of the following fileset states:
transient
corrupt
available
installed
configured

tag_string

Maximum length: 64 bytes
Examples: HP, SD

Tag strings containing a subset of **isascii()** characters only.
Requires one or more characters from: A-Z, a-z, 0-9, including the first character.
The **isspace()** characters are not allowed; see *ctype*(3C).
Metacharacters not allowed: **. , = #**
Shell metacharacters not allowed: **; & ( ) { } | < >**
Shell quoting characters not allowed: **" ' ' **
Directory path character not allowed: **/**

uname_string

Maximum length: 64 bytes
Examples: 9000/7*|9000/8*, HP-UX, ?.09.*, [A-Z]

Uname strings containing a subset of **isascii()** characters only.
No **isspace()** characters are allowed.
Shell pattern matching notation allowed: **[ ] * ? !**
Patterns can be "ORed" together using the separator: **|**

unsigned_integer

Maximum length: none

An integer in the range >= 0 and <= 2ˆ32.

S

SOFTWARE SPECIFICATION SYNTAX

The SD commands and attributes support the following syntax for each *software_specification:*

bundle[**.** *product*[**.** *subproduct*][**.** *fileset*]][**,** *version*]

product[**.** *subproduct*][**.** *fileset*][**,** *version*]

The **version** component has the form:

[**,r** *<op> revision*][**,a** *<op> arch*][**,v** *<op> vendor*]
[**,c** *<op> category*][**,l=***location*][**,fr** *<op> revision*]
[**,fa** *<op> arch*]

- *location* applies only to installed software and refers to software installed to a location other than the default product directory.
- **fr** and **fa** apply only to filesets.
- The *<op>* (relational operator) component can be of the form:

  ==, **>=**, **<=**, **<**, **>**, or **!=**

  which performs individual comparisons on dot-separated fields.

  For example, **r>=B.10.00** chooses all revisions greater than or equal to **B.10.00**. The system compares each dot-separated field to find matches. Shell patterns are not allowed with these operators.
- The **=** (equals) relational operator lets you specify selections with the shell wildcard and pattern-matching notations:

  [ ], *****, **?**, **!**

  For example, the expression **r=1[01].*** returns any revision in version 10 or version 11.
- All version components are repeatable within a single specification (e.g. **r>=A.12**, **r<A.20**). If multiple components are used, the selection must match all components.
- Fully qualified software specs include the **r=**, **a=**, and **v=** version components even if they contain empty strings.
- No space or tab characters are allowed in a software selection.
- The software *instance_id* can take the place of the version component. It has the form:

  [*instance_id*]

  within the context of an exported catalog, where *instance_id* is an integer that distinguishes versions of products and bundles with the same tag.

HOST OBJECT FORMAT

S

The file **/var/adm/sw/host_object** defines the software depots on the local host which have been registered with the **swagentd** daemon running on the host. The file has this format:

host
  data_model_revision 2.40

  distribution Defines a contained (registered) depot
    path *directory*

  distribution
    path *directory*

  ...

DEPOT (DISTRIBUTION) FORMAT

A depot is formatted using the following directory structure:

catalog/	Catalog of depot's contents
INDEX	Global index (table of contents)
swlock	Controls simultaneous modification
dfiles/	Stores all depot-specific information
INDEX	Defines depot-specific attributes
INFO	Defines depot-specific files
_ACL	Access Control List (ACL) for the depot
_OWNER	Owner and group of the depot's creator
_LOCK_FILE	Controls simultaneous ACL modification
_PROD_DFLT_ACL	Default ACL for new products
product_directory/	Catalog for a product
pfiles/	Stores all product-specific information
INDEX	Defines product attributes
INFO	Defines product files and scripts
README	The product's README attribute
scripts	Zero or more product control files
pcjob	A PC product's job file (HP OpenView Software Distributor only)
_ACL	ACL for the product
_OWNER	Owner and group of the product's creator
_LOCK_FILE	Controls simultaneous ACL modification
fileset_directory/	Catalog for a fileset
INDEX	Defines fileset attributes
INFO	Defines fileset files and scripts
scripts	Zero or more fileset control files
fileset_directory/	Catalog for the next fileset
...	
product_directory/	Catalog for the next product
...	
product_directory/	Contents of a product
fileset_directory/	Contents of a fileset
...	Contents of the next fileset
product_directory/	Contents of the next product
...	

The format is divided into two areas:

- the **catalog**, which contains all the information which describes the products contained in the depot;
- the **contents**, which stores the actual files contained in each product (one *product_directory* per product.)

Product and Fileset Control Directory Names

The *product.control_directory* and *fileset.control_directory* values are unique storage directories for a given product and fileset. They have this syntax:

tag[.*instance_id*]

The *tag* component is the product or fileset **tag** attribute. If more than one product with the same tag exists in the software depot, the *instance_id* component is necessary. This component is the product or fileset *instance_id* attribute.

On a short-filename filesystem, the *tag* component is truncated to the first 9 characters (if necessary). All products with the same truncated tag will also have a unique storage directory based on the *instance_id* component.

S

The following DOS information applies only to HP OpenView Software Distributor.

On a DOS filesystem, the *tag* component is truncated to the first 4 characters (always). All products with the same truncated tag will also have a unique storage directory based on the *instance_id* component.

Distribution Tape Format

A distribution tape contains the depot format (above), archived to one or more tapes in *tar*(1) format. The entire **catalog** precedes the depot contents on a distribution tape.

If the distribution spans multiple tapes, then the first tape contains the entire catalog, and as many fileset contents directories as will fit on the tape. Each additional tape contains only the **catalog/INDEX** followed by as many fileset contents directories as will fit on the tape.

INSTALLED PRODUCTS DATABASE FORMAT

An Installed Products Database (IPD) describes the software installed in a primary or alternate root filesystem. The format of an IPD is similar to the format of a software depot. An IPD is formatted using the following directory structure:

var/adm/sw/products/	Catalog of root's contents
INDEX	Global index (table of contents)
swlock	Controls simultaneous modification
ifiles/	Stores all root-specific information
INDEX	Defines root-specific attributes
INFO	Defines root-specific files
_ACL	Access Control List (ACL) for the root
_OWNER	Owner and group of the root's creator
_LOCK_FILE	Controls simultaneous ACL modification
product_directory/	Catalog for a product
pfiles/	Stores all product-specific information
INDEX	Defines product attributes
INFO	Defines product files and scripts
README	The product's README attribute
scripts	Zero or more product control files
fileset_directory/	Catalog for a fileset
INDEX	Defines fileset attributes
INFO	Defines fileset files and scripts
SAVE	Defines file attributes for a fileset that has been patched.
SAVEIPD	Defines file attributes for a fileset that has been patched.
scripts	Zero or more fileset control files
fileset_directory/	Catalog for the next fileset
...	
product_directory/	Catalog for the next product
...	

Product and Fileset Directory Names

The *product_directory* and *fileset_directory* values are as described for the depot format above.

INDEX FORMAT

An INDEX file describes all of the attributes (except for contained scripts and contained files) of a fileset, product, depot, or root. In a depot, the fileset, product, and depot INDEX files are concatenated together to form the **catalog/INDEX** file. In a root, the fileset, product, and root INDEX files are concatenated together to form the **var/adm/sw/products/INDEX** file.

This section describes the keyword syntax used in an INDEX file. The *value* for each **keyword** must meet the type restrictions described in the *OBJECT ATTRIBUTES* section.

Depot INDEX Format

distribution

tag	*tag_value*
copyright	*copyright_value*
data_model_revision	*revision_value*
description	*description_value*
dfiles	*dfiles_value*
layout_version	*revision_value*
mod_date	*date_value*
mod_time	*time_value*
name_max	*name_max_value*
number	*number_value*
path_max	*path_max_value*
pfiles	*tag_value*
title	*title_value*
uuid	*uuid_value*

media

sequence_number	*sequence_number_value*

No attributes are required.

Root INDEX Format

root

data_model_revision	*revision_value*
description	*description_value*
dfiles	*dfiles_value*
layout_version	*revision_value*
mod_date	*date_value*
mod_time	*time_value*
path	*path_value*
pfiles	*tag_value*
root_type	*root_value*

No attributes are required.

S

Bundle or Product INDEX Format

NOTE:

- The **tag** attribute is always required for all objects.
- The **contents** attribute is required for **subproducts** and **bundles**.
- Keywords marked with a **+** apply only to **product** INDEX format.
- Keywords marked with a **-** apply only to **bundle** INDEX format.

vendor

tag	*tag_value*
description	*description_value*
title	*title_value*
uuid	*uuid_value*

category

tag	*tag_value*
description	*description_value*
revision	*revision_value*
title	*title_value*

product *or* **bundle**

tag	*tag_value*
+ all_filesets	*all_filesets_value*
architecture	*architecture_value*
category_tag	*tag_value*
- contents	*contents_value*
control_directory	*control_directory_value*
copyright	*copyright_value*
create_date	*date_value*
create_time	*time_value*
data_model_revision	*revision_value*
description	*description_value*
+ directory	*directory_value*
install_date	*install_date_value*
install_source	*install_source_value*
install_type	*install_type_value*
instance_id	*id_value*
+ is_locatable	*boolean_value*
is_patch	*boolean_value*
+ job_file	*job_file_value*
+ location	*location_value*
machine_type	*machine_type_value*
mod_date	*date_value*
mod_time	*time_value*
number	*number_value*
os_name	*os_name_value*
os_release	*os_release_value*
os_version	*os_version_value*
+ postkernel	*postkernel_value*
qualifier	*qualifier_value*
+ readme	*readme_value*
revision	*revision_value*
share_link	*share_link_value*
size	*size_value*
software_spec	*software_specification_value*
title	*title_value*
vendor_tag	*vendor_tag_value*

subproduct

tag	*tag_value*
contents	*contents_value*
description	*description_value*
size	*size_value*
software_spec	*software_specification_value*
title	*title_value*

Fileset INDEX Format

NOTE:

- The **tag** attribute is always required.
- The **media_sequence_number** attribute is required for filesets within a multiple tape distribution.
- Multiple definitions of **ancestor**, **corequisites**, **exrequisites** and **prerequisites** are allowed.

fileset

tag	*tag_value*
ancestor	*ancestor_value*
applied_patches	*software_specification_value*
architecture	*architecture_value*
category_tag	*tag_value*
control_directory	*control_directory_value*
corequisites	*corequisites_value*
create_date	*date_value*
create_time	*time_value*
data_model_revision	*revision_value*
description	*description_value*
exrequisites	*exrequisites_value*
install_date	*date_value*
install_source	*install_source_value*
instance_id	*id_value*
is_kernel	*boolean_value*
is_locatable	*boolean_value*
is_packaged_in_place	*boolean_value*
is_patch	*boolean_value*
is_reboot	*boolean_value*
is_secure	*boolean_value*
is_sparse	*boolean_value*
location	*location_value*
machine_type	*machine_type_value*
media_sequence_number	*sequence_value*
mod_date	*date_value*
mod_time	*time_value*
os_name	*os_name_value*
os_release	*os_release_value*
os_version	*os_version_value*
patch_state	*patch_state_value*
pose_as_os_release	*release_value*
prerequisites	*prerequisites_value*
revision	*revision_value*
size	*size_value*
state	*state_value*
software_spec	*software_specification_value*
superseded_by	*software_specification_value*
supersedes	*software_specification_value*
title	*title_value*

INFO FORMAT

An INFO file describes all of the control scripts and other files contained within a fileset or product. This section describes the keyword syntax used in an INFO file.

Control File Format

control_file

tag	*tag_value*
interpreter	*interpreter_value*
path	*path_value*
result	*result_value*

The **path**, and **tag** attributes are required.

File Format

file

cksum	*cksum_value*
compressed_cksum	*cksum_value*
compressed_size	*size_value*
compression_state	*compression_state_value*
compression_type	*type_value*
file_attrs	*file_attrs_value*
gid	*gid_value*
group	*group_value*
is_volatile	*boolean_value*
link_source	*source_value*
mode	*mode_value*
mtime	*mtime_value*
owner	*owner_value*
path	*path_value*
revision	*revision_value*
size	*size_value*
source_path	*source_value*
type	*file_type_value*
uid	*uid_value*

The **path**, and **type** attributes are always required. The **link_source** attribute is required for hard link and symbolic link files.

ACL FORMAT

An Access Control List (ACL) has this format:

 # default_realm = *host_name*
num_entries = *count*
entry_type:[*key*:]*permissions*
entry_type:[*key*:]*permissions*
entry_type:[*key*:]*permissions*

See the *swacl*(1M) manual page for details on the fields in an ACL entry. Note that the *permissions* of a stored ACL are literal octal strings, but the *permissions* displayed by **swacl** are symbolic.

The **_OWNER** file uses a subset of the ACL format. It contains an entry for the **user** and **group** names of the user who created the corresponding object.

S

LOCKING

These commands use a common locking mechanism for reading and modifying both root directories and software depots. This mechanism allows multiple readers but only one writer on a root or depot.

The commands which modify software in an (alternate) root directory are restricted from simultaneous modification using *fcntl*(2) locking on the file

`var/adm/sw/products/swlock`

relative to the root directory (e.g. **`/var/adm/sw/products/swlock`**).

The commands which modify software in a depot are restricted from simultaneous modification using *fcntl*(2) locking on the file

`catalog/swlock`

relative to the depot directory (e.g. **`/var/spool/sw/catalog/swlock`**).

All commands set *fcntl*(2) read locks on roots and depots using the **`swlock`** file mentioned above. When a read lock is set, it prevents other commands from performing modifications (i.e. from setting write locks).

The **`swacl`** command is restricted from simultaneous modification of an ACL using *fcntl*(2) read and write locks on the

`_LOCK_FILE`

associated with each ACL.

AUTHOR

Software Distributor was developed by the Hewlett-Packard Company. **`swagent`**, **`swcopy`**, **`swinstall`**, **`swlist`**, and **`swpackage`** were developed by the Hewlett-Packard Company and Mark H. Colburn (see *pax*(1)).

SEE ALSO

The *Managing HP-UX Software with SD-UX* manual, the *HP OpenView Software Distributor Administrator's Guide*, swpackage(4), sd(5), swacl(1M), swagent(1M), swagentd(1M), swask(1M), swconfig(1M), swcopy(1M), swgettools(1M), swinstall(1M), swjob(1M), swlist(1M), swmodify(1M), swpackage(1M), swpackage(4), swreg(1M), swremove(1M), swverify(1M).

NAME

securenet - NIS map security file

DESCRIPTION

The **/etc/securenets** file defines networks and hosts that may access the NIS maps on a server. Each line in the file gives a network mask and a net address, each in dotted quad format. For example:

255.255.255.255 133.33.33.33

The file may have any number of netmask/net pairs.

When **ypserv** is started on the server, it checks for the existence of **/etc/securenets** and reads its contents into memory if it exists. **ypserv** must be stopped and restarted for any changes in **/etc/securenets** to take effect.

Upon startup, the netmask and the net address are converted to binary format and logical ANDed. The result must equal the net address (the second address) to be legal.

If the netmask is 255.255.255.255 (all 1's in binary), any address in the net address argument will match it. If any field in the netmask is 0, the corresponding field in the net address must be 0. When used in this way, the portion of the addresses given as 0 acts as a wild card.

When a client attempts to bind to the server, **ypbind** checks the client's IP against those given in the **/etc/securenets** file. Again, the address is converted to binary and logical ANDed with the netmask. The result must equal the net address given in the file. If the client address doesn't match any pairs in the file, the binding is refused with the message "no such map in server's NIS domain".

The **securenets** file can be used to limit access to specific hosts or to subnets using the wildcard capability.

If there are syntax errors in the **/etc/securenets** file, messages are logged to the **ypserv** logging file (default **/usr/adm/syslog**), and **ypserv** is not started.

If a host has multiple interfaces, each interface address must be allowed in the securenets file for that host to have reliable NIS access.

EXAMPLES

This line in **/etc/securenets** provides access only to the host with address 192.33.33.33:

255.255.255.255 192.33.33.33

This entry allows access by any host on the 192.33.33 subnet:

255.255.255.0 192.33.33.0

For broader access, for instance for an entire enterprise, this entry allows any host whose address begins with "15" to be served:

255.0.0.0 15.0.0.0

SEE ALSO

ypserv(1M)

NAME

services - service name data base

DESCRIPTION

The file **/etc/services** associates official service names and aliases with the port number and protocol the services use. For each service a single line should be present with the following information:

<official service name> *<port number/protocol name>* *<aliases>*

Port numbers 0 through 1023 are assigned by RFC 1700. This RFC also lists the conventional use of various ports with numbers greater than 1023.

Aliases are other names under which a service is known. Library routines such as **getservbyname()** can be invoked with a service alias instead of the service official name. For example:

```
shell   514/tcp    cmd
```

In this example, **getservbyname()** can be invoked with **cmd** instead of **shell**:

```
sp = getservbyname("cmd", "tcp");
```

instead of

```
sp = getservbyname("shell", "tcp");
```

Both produce the same results.

A line cannot start with a space or tab. Items are separated by any number of blanks (space or tab characters in any combination). The port number and protocol name are considered a single *item*. A **/** is used to separate the port and protocol (for example, **512/tcp**). A **#** character indicates the beginning of a comment. Characters from the **#** to the end of the line are not interpreted by routines which search the file.

Service names can contain any printable character other than a white space, newline, or comment character. Trailing blanks (spaces or tabs) are allowed at the end of a line.

Not all services listed in this file are available on HP-UX.

EXAMPLES

```
shell           514/tcp   cmd
telnet          23/tcp
login           513/tcp
```

AUTHOR

services was developed by the University of California, Berkeley.

FILES

/etc/services

SEE ALSO

getservent(3N).

S

NAME

shells - list of allowed login shells

SYNOPSIS

`/etc/shells`

DESCRIPTION

`/etc/shells` is an ASCII file containing a list of legal shells on the system. Each shell is listed in the file by its absolute path name.

Lines or portions of lines beginning with `#` are assumed to be comments and are ignored. Blank lines are also ignored.

AUTHOR

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

FILES

`/etc/shells`

SEE ALSO

chsh(1), ftpd(1M), getusershell(3C).

NAME

sm, sm.bak, state - statd directory and file structures

SYNOPSIS

/var/statmon/sm

/var/statmon/sm.bak

/var/statmon/state

DESCRIPTION

/var/statmon/sm and **/var/statmon/sm.bak** are directories generated by **statd** (see *statd*(1M)). Each file in **/var/statmon/sm** represents one or more machines to be monitored by the **statd** daemon. Each file in **/var/statmon/sm.bak** represents one or more machines to be notified by the **statd** daemon upon its recovery.

/var/statmon/state is a file generated by **statd** to record its version number. This version number is incremented each time a crash or recovery takes place.

SEE ALSO

lockd(1M), statd(1M).

S

NAME

snmpd.conf - configuration file for the SNMP agent

DESCRIPTION

When invoked, the SNMP agent reads its configuration information from the **/etc/SnmpAgent.d/snmpd.conf** configuration file. The SNMP agent is either the *snmpd*(1M) (included with HP-UX) or the *snmpd.ea*(1M) (purchased with the OpenView product). The SNMP agent operates correctly if no values are configured in **/etc/SnmpAgent.d/snmpd.conf**.

/etc/SnmpAgent.d/snmpd.conf contains the following configurable values:

get-community-name: name IP: addr VIEW: mib-view

Specifies a community name for the agent. The agent responds to SNMP GetRequests with this community name. You can configure the agent to respond to more than one get community name. If a community name is not entered, the agent responds to SNMP GetRequests using any community name.

There are two, optional, fields that may be associated with a community name. They are IP: and VIEW:. These fields allow you to associate manager IP addresses and MIB views with community names.

Following **IP:** is a list of IP address in dot notation. The list MUST be space separated. Only SNMP Requests with one of these IP addresses, as the source address, will be accepted. Any source IP address will be allowed if; IP: does not appear on the line, no IP address appears after IP:, or an address of 0.0.0.0 is placed after IP:. No wildcarding is supported.

The **VIEW:** is used to associate a MIB view with the community string. Placing a name or OID from some portion of the MIB tree will cause that object plus all portions of the tree below that object to be included in the view. Several names may be placed on the line after VIEW: and MUST be space separated. If the '-' character proceeds an object name then it is excluded from the NIB view. For example: **VIEW: internet -mib-2** would allow access to all of internet except for mib-2. The default mib view will be assigned if VIEW: does not appear on the line or no MIB view appears after VIEW:. It is important to note that any MIB object(s) specified after VIEW: **override** the default MIB view and are not used in conjunction with it. The default MIB view is specified below under **default-mibVIEW:** and is configurable.

Lines may be continued using the "\" character.

set-community-name: name IP: addr VIEW: mib-view

Specifies community name for the agent. The agent responds to the SNMP SetRequests with this community name. You can configure the agent to respond to more than one set community name. If a community name is not entered, the agent returns an error. The IP: and VIEW: fields for set community names are the same as for get community names. See above.

S

trap-dest: Specifies a system name where traps are sent (that is, the trap destination). This system name is usually the host name or IP address of the manager. If traps should be sent to multiple systems then a **trap-dest** line should be included for each system.

location: Specifies the physical location of the agent.

contact: Specifies the person responsible for this agent and information on how to contact this person.

sys-descr: Specifies the system description. This value becomes the system.sysDescr MIB object.

default-mibVIEW:

Specifies the new default MIB view. By default, the MIB view of **internet** is set by the system. The default MIB view may be changed several times and anywhere in the configuration file. This new MIB view is "in effect" from the point in the file that it is defined until another **default-mibVIEW:** is encountered. To reset the system supplied MIB view, enter **default-mibVIEW:** with no MIB objects after it. The MIB view values are specified in the same way as for the **get-community-name:** above.

Separate the fields by blanks or tabs. A **#** character indicates the beginning of a comment; characters from the **#** character to the end of the line are ignored.

EXAMPLES

Each line in the following example **snmpd.conf** file is preceded by a comment (beginning with **#**) that explains the entry. Please note that this is an example. Taken as a whole in is not intended to represent a configuration that you should use. It's sole purpose is to show the flexibility of the configuration file.

```
# We'll specify the location, contact, and sysDescr first.
location: Somewhere in the building.
contact: Jane Doe
sys-descr: HP-UX testsys1 A.09.04 E 9000/887 400509201

# This community string has the system default MIB view
# and any management station can use it.
get-community-name: globalget

# Setting a new default MIB view.
default-mibVIEW: system

# The following get and set community names will have the new
# default MIB view of system. But are restricted to use by
# systems 15.2.2.1 and 15.2.2.3. Note that the sysset community
# string is only usable from 15.2.2.1.
get-community-name: sysget IP: 15.2.2.1 15.2.2.3
set-community-name: sysset IP: 15.2.2.1

# Resetting the default MIB view back to the original
# system default.
default-mibVIEW:

# Now allow some specific machines access to limited
# portions of the MIB.
# Note use of line continuation character '\'.
get-community-name: monitor IP: 15.3.2.1 15.4.23.1 VIEW: system \
interfaces at ip snmp
get-community-name: public IP: 0.0.0.0 VIEW: system

# Set up an administrative role and a root role.
set-community-name: admin IP: 15.3.2.1 15.4.23.1 VIEW: internet \
-private
set-community-name: root IP: 15.3.2.1 VIEW: internet

# Specify some trap dests.
trap-dest: 15.2.1.45
trap-dest: 15.3.2.1
trap-dest: 15.4.23.1
```

AUTHOR

snmpd.conf was developed by HP.

FUTURE DIRECTIONS

The following files are not currently used. They are place holders for the SNMPV2 configuration information:

snmpv2.acl, **snmpv2.ctx**, **snmpv2.party**, and **snmpv2.view**

SEE ALSO

snmpd(1M), snmpd.ea(1M).

RFC 1155, RFC 1157, RFC 1212, RFC 1213, RFC 1231, RFC 1398

NAME

softkeys - keysh softkey file format

BACKGROUND

keysh softkey information is stored in the form of a softkey node hierarchy. The top level of this hierarchy represents the softkey commands themselves; lower levels represent various command options and parameters.

The softkey labels form a **window** into this softkey node hierarchy through which the user can view and select **eligible** nodes. A node is eligible if it was:

- Enabled by default and has not been subsequently disabled by the selection of some sibling node, or
- Disabled by default, has not been subsequently disabled by the selection of some sibling node, but has been subsequently enabled by the selection of some sibling node.

When a softkey node is selected, it can enable or disable any of its siblings as appropriate. A new window into the softkey node hierarchy is then computed as follows:

- If the selected node was not a leaf node, its eligible children are displayed;
- Otherwise, if the node still has eligible siblings remaining, they are redisplayed;
- Otherwise, if the node's parent still has eligible siblings remaining, they are redisplayed, and so on, moving up the node hierarchy.

This process of node display and selection continues until the user has entered a complete command.

At that point, **keysh** performs the **editrules** associated with each of the selected softkey nodes. These editrules create the HP-UX command that is fed to the shell for execution.

SOFTKEY FILE FORMAT

Each softkey file contains one or more softkey definitions, each of which is represented as a sub-hierarchy of **softkey nodes**.

There are two basic types of softkey nodes:

option "Options" show up on softkey labels and insert literal text into the command-line when selected. Examples are command and option names.

string "Strings" (or "parameters") show up on softkey labels but do not insert text into the command-line when selected; rather, they display a hint message. The user must then type the desired text into the command-line. Examples are file and user names.

Note that the keyword **softkey** can be used as a synonym for the keyword **option**.

The basic softkey node definition is composed of the following components:

{**option** | **string**} *softkey*
attribute
.
.
.
;

Where *softkey* is the softkey node name from which the command-line text and softkey label are derived. If necessary, a single plus sign (**+**) within *softkey* can be used to force hyphenation of the softkey label at a syllable boundary.

If a softkey node has an associated sub-menu, its trailing **;** is replaced with a list of child nodes as follows:

{
softkey node
.
.
.
}

Each softkey node can have the following optional *attribute* fields:

disable *count* Selecting this node will disable *count* softkey nodes to the right of this one - default is 0.

S

enable *count* — Selecting this softkey will enable *count* softkey nodes to the right of this one - default is 0.

{**filter**|**command**} — This node is only active for filters or commands, respectively - default is either.

{**motorola**|**precision**} — This node is only active when **keysh** is running on a Motorola (MC680x0) or precision (PA-RISC) processor, respectively - default is either.

disabled — This node starts out disabled and must be enabled to be used - default is to start out enabled.

automatic — The command will be entered automatically when this node is selected.

editrule *editrule* — The editrule for this node.

cleanuprule *editrule* — An editrule to be executed *after* all other editrules associated with this softkey command - only one cleanuprule is allowed per softkey command.

hint *string* — The one line hint for this node - only valid for "string" softkey nodes.

help *helptext* — The help for this node (may be more than one line).

required *string* — The one-line error message to display if this node is not selected.

Arguments are as follows:

count — A signed integer, the word **none**, or the word **all**.

editrule — An editrule (described below).

helptext — **nroff**-style help enclosed in quotes (also described below).

string — An arbitrary string enclosed in quotes. Note that within quotes, **** escapes the next character as when using *awk*(1).

A typical backup softkey node definition resembles:

backup softkey *softkey* [**literal** *literal*] **;**

where *literal* is the literal text string to program the terminal function key with (if different than *softkey*).

An unquoted hash-mark character (**#**) in a softkey file delimits a comment to the end of the line.

Softkey Command Translation

To translate softkey commands into HP-UX commands, **keysh** executes the *editrules* associated with each softkey selected by the user. These editrules create a **word list** via an **awk**-like editing language. This translated word list is then passed to the shell for execution.

S

For a simple translation, this list might resemble:

ls	-l	*.c
word[0]	word[1]	word[2]

Every time an editrule is invoked, the special constants **last** and **next** are defined to the index of the last word in the list ("2" in this example) and the would-be-next word in the list ("3" in this example), respectively. In addition, the constant **argument** is set equal to the user input for the softkey (e.g., ***.c** for the softkey corresponding to the file name in this example).

Note that **keysh** automatically casts numbers and strings back and forth as necessary to carry out editrules. Also, variables are cleared only before the first editrule associated with a softkey command. All assigned variables are available to subsequent editrules.

Editrules

An editrule is a list of edit statements enclosed in curly-braces (i.e., **{** and **}**).

An edit statement is:

- an expression followed by a **;**,
- an **if** statement, or

- a word allocation statement.

Expressions

A simple expression can be any of:

variable	single letter from a to z
number	unsigned integer
string	enclosed in quotes
char	enclosed in quotes
last	see above
next	see above
argument	see above
motorola	boolean flag
precision	boolean flag
command	boolean flag
filter	boolean flag
word[*number* **]**	see above

Simple expressions can be combined with any of:

*string***[***number***]**	single-character substring
*string***[***number***,***number***]**	multiple-character substring
*number***+***number*	addition
*number***-***number*	subtraction
*number*******number*	multiplication
*number***/***number*	division
*number***%***number*	modulus
*string***&***string*	concatenation
-*number*	negation
*string***==***string*	equality
*string***!=***string*	inequality
*number***>=***number*	greater than or equal
*number***<=***number*	less than or equal
*number***>***number*	greater than
*number***<***number*	less than
*number***&&***number*	logical and
*number***||***number*	logical or
!*number*	logical not
(*string***)**	grouping

The following functions are also supported and return the indicated results:

strlen(*string***)**	number of characters in *string*
strchr(*string***,***char***)**	index of first *char* in *string*, or –1
strrchr(*string***,***char***)**	index of last *char* in *string*, or –1
trim(*string***)**	*string* without leading/trailing blanks
hex(*number***)**	*number* in hex with leading **0x**
octal(*number***)**	*number* in octal with leading **0**

Assignments can be done with any of:

*variable***=***string*	simple assignment
*variable***+=***number*	add and assign
*variable***-=***number*	subtract and assign
*variable****=***number*	multiply and assign
*variable***/=***number*	divide and assign
*variable***%=***number*	modulus and assign
*variable***&=***string*	concatenate and assign
word[*number***]=***string*	simple assignment
word[*number***]+=***number*	add and assign
word[*number***]-=***number*	subtract and assign

S

word[*number***]*=***number*	multiply and assign
word[*number***]/=***number*	divide and assign
word[*number***]%=***number*	modulus and assign
word[*number***1]&=***string*	concatenate and assign

if Statement

The **if** statement is similar to the full-block mode **if** statement in awk, and is structured as follows:

```
if(number) {
    edit statement
    .
    .
    .
} else {
    edit statement
    .
    .
    .
}
```

Where the **else** part is optional. If *number* is non-zero, the first block of *edit statement*s is executed. Otherwise, if the second block of *edit statement*s is present, it is executed.

Word Allocation Statements

Word allocation statements include the following:

insert(*number***,** *string***);**	Insert *string* as a new word in the word list immediately before **word[***number***]**.
append(*string***);**	Insert *string* as a new word in the word list immediately after the last word in the word list. Equivalent to **insert(next,** *string***);**.
dash(*string***);**	Append *string* to the last word in the word list *if* that word already begins with a dash. Otherwise, a dash is inserted as a new word in the word list immediately after the last word in the word list and *string* is appended to that.
delete(*number***);**	Delete **word[***number***]** from the word list.

Helptext

Each softkey node can have an associated **helptext**, to be displayed upon a user request for help. This helptext is formatted on-the-fly and presented to the user through the preferred pager.

The helptext format is an nroff-like language, supporting a subset of the *man*(5) macros used to write standard HP-UX manual entries. In particular, this subset includes:

.nf	Begin **no-fill** mode. Display text as-is, preserving new-lines and spaces, until a **.fi**.
.fi	Resume **fill** mode. Display text with words filled onto each output line, attempting to utilize 90% of the screen width. (This is the default mode.)
.br	Force a **break** in the current output line. Display subsequent text on the next line.
.sp	Force a break and then display a single blank line (a vertical **space**).
.P	Force a break, display a single blank line, and then begin a new **paragraph** with no indent.
.IP *tag indent*	Force a break, display a single blank line, and then display the specified *tag*, then begin a new **indented paragraph** with the specified *indent*.
.IL *tag indent*	Begin a new **indented line** (similar to **.IP** except no blank line is displayed).

Note that these macros are recognized *anywhere* in the input helptext, not just at the beginning of a line. Also, all macro arguments *must be present*, even if they consist of nothing more than a quoted empty string.

EXAMPLES

For a custom **cd** command (see *cd*(1)):

```
softkey cd
editrule { append("cd"); }
{
  softkey keysh-src disable all
  editrule { append("~/keysh/src"); }
  ;
  softkey keysh-test disable all
  editrule { append("~/keysh/test"); }
  ;
  softkey keysh-doc disable all
  editrule { append("~/keysh/doc"); }
  ;
  softkey demo disable all
  editrule { append("~/demo"); }
  ;
  softkey tmp disable all
  editrule { append("/tmp"); }
  ;
  string <dir> disable all
  editrule { append(argument); }
  required "Enter the name of the directory to move to."
  ;
}
```

For other examples, refer to the file **/usr/lib/keysh/C/softkeys**.

AUTHOR

keysh was developed by HP and AT&T.

FILES

$HOME/.softkeys	user softkey definitions file
/usr/lib/keysh/$LANG/softkeys	standard softkey definitions file

SEE ALSO

keysh(1), man(5).

NAME

swpackage - product specification file (PSF) format

DESCRIPTION

Introduction

The **swpackage** command packages software into:

- a distribution directory (which can be accessed directly or copied onto a CD-ROM),
- a distribution tape (such as DDS, nine-track or cartridge tapes).

Both directory and tape distributions use the same format. See *sd*(4) for details on this format.

The software is organized into a four-level hierarchy of software objects: **bundles**, **products**, **subproducts**, and **filesets**. The files that make up a software package are contained in filesets. Filesets are contained in subproducts and/or products. Currently, only HP creates software bundles to contain the entire application. The attribute tables that follow show the attributes of each level of the software packaging hierarchy.

A **Product Specification File** (PSF) defines how a product is structured and the attributes that apply to it. This manual page describes the syntax and semantics of a PSF.

Layout Version

SD object and attribute syntax conforms to the *layout_version 1.0* specification of the *IEEE POSIX 1387.2 Software Administration* standard. The previous SD layout_version 0.8 is also supported. SD for HP-UX version 10.10 and later can read or write either layout version. SD commands still accept the keyword names associated with the older layout version, but you should use layout_version 0.8 only to create distributions readable by older versions of SD.

What layout_version the SD commands write is controlled by the **layout_version** option for **swpackage**, **swmodify**, **swcopy**, and **swlist**.

The version used by **swpackage** can be also controlled by specifying the *layout_version* attribute in the PSF. However, if the *layout_version* attribute in the PSF is 1.0, the *is_locatable* attribute defaults to true in all cases, and must be explicitly set to false.

For a full description of the **swpackage** command, see the *swpackage*(1M) manual page.

Layout version 1.0 adds significant functionality not recognized by systems supporting only 0.8, including:

- Category class objects (formerly the **category** and **category_title** attributes within the bundle or product class).
- Patch-handling attributes, including **applied_patches**, **is_patch**, and **patch_state**.
- The fileset **architecture** attribute, which permits you to specify the architecture of the target system on which the product will run.

In addition to adding new attributes and objects, layout_version 1.0 changes the following preexisting 0.8 objects and attributes as follows:

- Replaces the depot **media_sequence_number** with the **media** object with a **sequence_number** attribute.
- Replaces the **vendor** definition within products and bundles with a **vendor_tag** attribute and a corresponding **vendor** object defined outside the product or bundle.
- Pluralizes the **corequisite** and **prerequisite** fileset attributes (to **corequisites** and **prerequisites**).
- Changes the **timestamp** attribute to **mod_time**.

PRODUCT SPECIFICATION FILE SYNTAX

A PSF is structured as follows:

```
[<distribution specification>]
    [<vendor specification>]
    [<category specification>]
    [<bundle specification>]
    ...
    <product specification>
        [<control script specifications>]
        [<subproduct specifications>]
        <fileset specification>
            [<control script specifications>]
            <file specifications>
        [<fileset specification>]
        ...
    [<vendor specification>]
    [<product specification>]
    ...
```

In summary, the **swpackage** user can:

- Specify one or more products.
- For each product, specify one or more filesets.
- For each fileset, specify one or more files.
- (optional) Specify attributes for the target depot or tape.
- (optional) Specify one or more bundles, defining the bundle contents.
- (optional) Specify vendor information to be used with subsequent products and bundles.
- (optional) For each product, specify one or more subproducts, defining the subproduct contents.
- (optional) For each product or fileset, specify one or more control scripts.

Each software object has user-defined attributes. Most attributes are optional. All objects and attributes are defined using a

keyword *value*

syntax. The **keyword** is an identifier for the attribute. Specific rules for each keyword are:

- All keywords require one or more values, except as noted. If the value is missing an error is given.
- Comments must be preceded by **#**. A comment can appear on a line by itself or following the keyword-value syntax on a command line.
- Use double quotes (") to define values that span multiple lines:

```
"This is an example of a
 two-line value."
```

- Double quotes (") are optional when defining a value that contains embedded whitespace.

Attribute Table

The following tables summarize the objects and attributes which can be defined in a PSF. These objects and attributes can appear in any order when defining a distribution, vendor, category, product, or bundle, except that the *layout_version* attribute must be first. Each object and attribute is identified by a keyword. Object keywords do not have associated values. Attribute keywords have one or more values.

NOTE:

- Attributes marked with a ***** determine the uniqueness of a product, bundle, or fileset. Their values may also be of the type **version_component** when used in a version component of a software specification.
- Keywords marked with a **+** apply to products only.
- Keywords marked with a **-** apply to bundles only.
- *control_files* can be defined within products or filesets or both.

Keyword	Type	Size	Example
distribution			
layout_version	revision_string	64	1.0
tag	tag_string	64	EXAMPLE_DEPOT
copyright	multi_line_string	8K	< data/copyr.depot
description	multi_line_string	8K	< data/descr.depot
number	one_line_string	64	B2358-13601
title	one_line_string	256	Example packages
end			
vendor			
tag	tag_string	64	HP
description	multi_line_string	8K	< data/descr.hp
title	one_line_string	256	Hewlett-Packard Co.
end			
category			
tag	tag_string	64	patch_normal
description	multi_line_string	8K	For normal problems
revision	revision_string	64	0.0
title	one_line_string	256	Category of Patches
end			
product or bundle			
*** tag**	tag_string	64	SD
*** architecture**	one_line_string	64	HP-UX_B.11.00_32/64
category_title	one_line_string	256	Systems Management
- contents	repeatable list of software_specs	8K	pr.fs,r=1.0,a=,v=
copyright	multi_line_string	8K	< data/copyr.sd
description	multi_line_string	8K	< data/descr.sd
directory	path_string	1024	/
is_locatable	boolean	9	false
is_patch	boolean	9	false
machine_type	uname_string	64	9000/[78]*:*
number	one_line_string	64	B1991A
os_name	uname_string	64	HP-UX
os_release	uname_string	64	?.11.*
os_version	uname_string	64	?
+ postkernel	path_string	255	/usr/bin/kernel_build
+ readme	multi_line_string	1024K	< data/README.sd
*** revision**	revision_string	64	A.01.00
+ share_link	one-line_string	256	
title	one_line_string	256	Software Distributor
*** vendor_tag**	tag_string	64	HP
control_files			
end			

S

Attribute Table (continued)

Keyword	Type	Size	Example
subproduct			
tag	tag_string	64	Manager
contents	one-line list of tag_string values		commands agent data data man
description	multi_line_string	8K	< data/desc.mgr
title	one_line_string	256	Management Utilities
end			
fileset			
* **tag**	tag_string	64	commands
ancestor	repeatable list of product.fileset		product.oldfileset oldproduct.fileset
architecture	one_line_string	80	HP-UX_B.11.00_32/64
category_tag	tag_string	64	patch_normal
corequisites	software_spec		SD.man,r>=2.0
description	multi_line_string	8K	< data/descr.cmd
is_kernel	boolean	9	false
is_patch	boolean	9	false
is_reboot	boolean	9	false
is_sparse	boolean	9	false
machine_type	uname_string	64	9000/[78]*:*
os_name	uname_string	64	HP-UX
os_release	uname_string	64	?.11.*
os_version	uname_string	64	?
prerequisites	software_spec		SD.agent,r>=2.0
* **revision**	revision_string	64	2.42
supersedes	software_spec	8192	product.fileset, fr=revision
title	one_line_string	256	SD Commands
control_files			
control_files			
directory	path_mapping_string		./commands = /usr/sbin
exrequisites			
file_permissions	permission_string		-u 0222 -o root -g sys
file	file specification		-m 04555 bin/swinstall (or) *
end			

Control File Attributes

Control files can be defined within filesets and/or products.

Keyword	Type	Size	Example
checkinstall	path_string	1024	./scripts/checkinstall
checkremove	path_string	1024	./scripts/checkremove
configure	path_string	1024	./scripts/configure
control_file	path_string	1024	./scripts/subscripts
postinstall	path_string	1024	./scripts/postinstall
postremove	path_string	1024	./scripts/postremove
preinstall	path_string	1024	./scripts/preinstall
preremove	path_string	1024	./scripts/preremove
request	path_string	1024	./scripts/request
unconfigure	path_string	1024	./scripts/unconfigure
unpreinstall	path_string	1024	./scripts/unpreinstall
unpostinstall	path_string	1024	./scripts/unpostinstall
verify	path_string	1024	./scripts/verify

VALUE TYPES

The value for each attribute must be of a specific type. The types are:

tag_string

Maximum length: 64 bytes
Examples: HP, SD

Tag strings support a subset of **isascii()** characters only:
Requires one or more characters from: "A-Z", "a-z", "0-9", including the first character.
The **isspace()** characters are not allowed.
SDU metacharacters not allowed: **. , : =**
Shell metacharacters not allowed: **# ; & ( ) { } | < >**
Shell quoting characters not allowed: **" ' `**
Directory path character not allowed: **/**

one_line_string

Maximum length: 256 bytes
Examples: Hewlett-Packard Company

One-line strings support a subset of **isascii()** characters only:

No **isspace()** characters, except for space and tab, are allowed.

multi_line_string

Maximum length: 8K (1Mb for **readme**)

Multi-line strings support all **isascii()** characters. They represent one or more paragraphs of text. They can be specified in-line, surrounded by double-quotes. They can also be stored in a file, and specified using the "**<** *filename*" format.

revision_string

Maximum length: 64 bytes
Examples: 2.0, B.11.00

Revision strings contain zero or more dot-separated one_line_strings (above).

boolean

Maximum length: 8 bytes
Examples: true, false

One of the values "true" or "false".

path_string

Maximum length: 255 bytes for tapes, 1024 bytes for depots
Examples: **/usr**, **/mfg/sd/scripts/configure**

An absolute or relative path to a file. Many attributes of this type are restricted to 255 bytes in length. This restriction is due to the *tar*(1) command, which requires a file's *basename*(1) be <= 100 bytes, and a file's *dirname*(1) to be <= 155 bytes. (Some implementations of **tar** enforce < and not <=.)

uname_string

Maximum length: 64 bytes
Examples: 9000/7*:*|9000/8*:*, HP-UX, ?.11.*

Uname strings containing a subset of **isascii()** characters only.
No **isspace()** characters are allowed.
Shell pattern matching notation allowed: **[ ] * ? !**
Patterns can be "ORed" together using the separator: **|**

path_mapping_string

Maximum length: none
Examples: **/mfg/sd/files/usr = /usr**

A value of the form: "*source*[**=***destination*]" where the source defines the directory in which subsequently defined files are located. The optional destination maps the source to a destination directory in which the files will actually be installed.

S

file_specification

Maximum length: none
Examples: -m 04555 sbin/swinstall or * (to denote all files and directories)

Explicitly specifies a file or directory to be packaged, using the format:

[**-m** *mode*] [**-o** [*owner*[**,**]][*uid*]]
[**-g** [*group*[**,**]][*gid*]] [**-v**] [*source*] [*destination*]

The source and destination can be paths relative to source and destination directories specified in the path_mapping_string.

You can also use ***** to include all files below the source directory specified by a **directory** keyword.

permission_string

Maximum length: none
Examples: -u 0222 -o root -g sys

A value of the form:

[**-m** *mode***|-u** *umask*] [**-o** [*owner*[**,**]][*uid*]]
[**-g** [*group*[**,**]][*gid*]]

where each component defines a default permissions value for each file and directory defined in a fileset. The default values can be overridden in each file's specific definition. The owner and group fields are of type tag_string. The uid and gid fields are of type unsigned integer. The mode and umask are unsigned integers, but only supports the octal character set: "0"-"7".

software_specification

Maximum length: none
Examples: SD.agent or SD,r=2.0,a=HP-UX_B.11.00_32

Software specifications are used to specify software in dependencies, ancestors and other attributes, as well as command line selections. The SD commands and attributes support the following syntax for each *software_specification:*

bundle[**.** *product*[**.** *subproduct*][**.** *fileset*]][**,** *version*]

product[**.** *subproduct*][**.** *fileset*][**,** *version*]

The **version** component has the form:
[**,r** <*op*> *revision*][**,a** <*op*> *arch*][**,v** <*op*> *vendor*]
[**,c** <*op*> *category*][**,l=***location*][**,fr** <*op*> *revision*]
[**,fa** <*op*> *arch*]

S

- *location* applies only to installed software and refers to software installed to a location other than the default product directory.
- **fr** and **fa** apply only to filesets.
- The <*op*> (relational operator) component can be of the form:

  ==, **>=**, **<=**, **<**, **>**, or **!=**

  which performs individual comparisons on dot-separated fields.

  For example, **r>=B.10.00** chooses all revisions greater than or equal to **B.10.00**. The system compares each dot-separated field to find matches. Shell patterns are not allowed with these operators.
- The **=** (equals) relational operator lets you specify selections with the shell wildcard and pattern-matching notations:

  [ ], *****, **?**, **!**

  For example, the expression **r=1[01].*** returns any revision in version 10 or version 11.
- All version components are repeatable within a single specification (e.g. **r>=A.12**, **r<A.20**). If multiple components are used, the selection must match all components.

- Fully qualified software specs include the **r=**, **a=**, and **v=** version components even if they contain empty strings.
- No space or tab characters are allowed in a software selection.
- The software *instance_id* can take the place of the version component. It has the form:

  [*instance_id*]

  within the context of an exported catalog, where *instance_id* is an integer that distinguishes versions of products and bundles with the same tag.

PRODUCT SPECIFICATION FILE SEMANTICS

The following sections describe the attributes which can be defined.

Distribution (Depot) Specification

The following is an example of a distribution specification:

distribution or **depot**

```
    layout_version        1.0
    tag                   APPLICATIONS_CD
    copyright             < data/copyright.cd
    description           < data/description.cd
    number                B2358-13601
    title                 HP-UX Applications Software Disc
```

[<*vendor specification*>]

...

[<*bundle specification*>]

...

<*product specification*>

[<*product specification*>]

...

end

distribution or **depot**

Keyword that begins the distribution specification. Each keyword defines an attribute of the distribution depot or tape itself. All keywords are optional, even if a distribution specification is included in a PSF.

layout_version

Defines the semantics to use when parsing the PSF. To ensure IEEE Standard 1387.2 semantics, define a **layout_version** of **1.0**, as the first attribute.

tag Defines the identifier (short name) for the distribution depot or tape.

copyright

Defines the copyright information for the distribution depot or tape; the value is either the text itself (within double-quotes) or a pointer to the filename containing the text.

description

Defines the multi-paragraph description of the distribution depot or tape; the value is either the text itself (within double-quotes) or a pointer to the filename containing the text.

distribution

If a distribution specification is included in the PSF, **swpackage** requires only the keyword plus one or more contained product definitions. The **depot** keyword can also be used in place of **distribution**.

number

Defines the part or manufacturing number of the distribution depot (e.g. CD-ROM) or tape.

title

Defines the full name (one-line description) of the distribution depot or tape.

end Ends the distribution specification. This keyword is optional.

Vendor Specification

The **layout_version** defined for the PSF file determines how vendor specifications are associated with products and bundles. If a **layout_version** is not defined or is defined as **1.0**, vendor specifications will be associated with all subsequent products and bundles that define a matching **vendor_tag** attribute.

If a **layout_version** of **0.8** is specified, all subsequent products and bundles will automatically be assigned a **vendor_tag** from the last vendor object defined at the distribution level, if any, or from a vendor object defined within a product or bundle, unless a **vendor_tag** is explicitly defined.

The following is an example of a vendor specification:

```
vendor
     tag                      HP
     description              < data/description.hp
     title                    Hewlett-Packard Company
end
```

Each keyword defines an attribute of a vendor object. If a vendor specification is included in the PSF, **swpackage** requires the **vendor** and **tag** keywords.

vendor
Keyword that begins the vendor specification.

tag Defines the identifier (short name) for the vendor.

title
Defines the full name (one-line description) for the vendor.

description
Defines the multi-paragraph description of the vendor; the value is either the text itself (within double-quotes) or a pointer to the filename containing the text.

end Ends the vendor specification. This keyword is optional.

Category Specification

The following is an example of a category specification.

```
category
    tag
    title
    description
    revision
end
```

S

category
Keyword that begins the category specification.

tag Defines the identifier (short name) for the category.

title
Defines the full name (one line description) for the category.

description
A more detailed description of the category.

revision
Determines which category object definition to maintain in a depot when a definition being installed or copied does not match a definition already in the depot with the same **category_tag**.

end Ends the category specification. This keyword is optional.

Product or Bundle Specifications

The following is an example of a product or bundle specification. Keywords marked with a **+** apply to **products** only and keywords marked with a **-** apply to **bundles** only. Products are assumed to be locatable unless they explicitly define the *is_locatable* attribute to **false**. Non-locatable products must define this attribute.

```
product or bundle
 tag                   SD
 architecture          HP-UX_B.11.00_32/64
 category_tag          system_mgt
 - contents            prod.fs1,r=1.0,a=,v=
 copyright             < data/copyright.sd
 description           < data/description.sd
 directory             /
 is_locatable          false
 is_patch              false
 machine_type          9000/7*:*
 number                J2326AA
 os_name               HP-UX
 os_release            ?.11.*
 os_version            [A-Z]
 postkernel            /usr/lbin/kernel_build
 + readme              < data/README.sd
 revision              2.0
 title                 HP OpenView Software Distributor
 vendor_tag            HP
```

+ [*<control script specifications>*]

+ [*<subproduct specifications>*]

+ *<fileset specification>*

+ [*<fileset specification>*]

...

end

Each keyword defines an attribute of a product or bundle object. For each product specified, **swpackage** requires only the **product** and **tag** keywords, plus one or more contained **fileset** definitions. For each bundle specified, **swpackage** requires the **bundle**, **tag**, and **contents** keywords.

product
Required keyword that begins the product specification.

tag Defines the identifier (short name) for the product or bundle.

architecture
Describes the target system(s) on which the product or bundle will run. Provides a human-readable summary of the four *uname*(1) attributes which define the exact target system(s) the product supports.

bundle
Required keyword that begins the bundle specification.

category_tag
A repeatable tag-based attribute identifying a set of categories of which the software object is a member. This is used as a selection mechanism and can be used independent of patches. The default value is an empty list or **patch** if the **is_patch** attribute is set to **true**.

Like **vendor_tag**, this attribute can be used as a pointer to a category object that contains additional information about the category (for example, a one-line *title* definition and a *description* of the category).

Note that the category tag **patch** is reserved. When **is_patch** is set to **true**, a built-in **category_tag** attribute of value **patch** is automatically included.

NOTE: You can only change the **patch** value by performing a **swpackage** operation or by using **swmodify** to change the value of the **is_patch** attribute.

contents
The list of **fully qualified** (all version-distinguishing attributes included) software_specs for the bundle.

copyright
Defines the copyright information for the product or bundle; the value is either the text itself (within double-quotes) or a pointer to the filename containing the text.

description
Defines the multi-paragraph description of the product or bundle; the value is either the text itself (within double-quotes) or a pointer to the filename containing the text.

directory
Defines the default, absolute pathname to the directory in which the product's files will be installed (i.e. the root directory of the product). If this attribute is not specified, **swpackage** assigns a value of "/".

is_locatable
Defines whether the product or bundle can be installed into any directory, or whether it must be installed into a specific directory. If this attribute is not specified, **swpackage** assigns a value of "true".

is_patch
Identifies a software object as a patch. The default value is **false**. When set to **true**, a built-in **category_tag** attribute of value **patch** is automatically included.

machine_type
Defines the machine(s) on which the product will run. (If not specified, **swpackage** assigns a value of "*", meaning the product runs on all machines.) If there are multiple machine platforms, use wildcards or use the '|' character to separate them. This attribute should pattern match to the value of

uname -m [: getconf HW_CPU_SUPP_BITS]

on the supported target machine(s).

number
Defines the part or order number for the product.

os_name
Defines the operating system(s) on which the product will run. (If not specified, **swpackage** assigns a value of "*", meaning the product runs on all operating systems.) If there are multiple operating systems, use wildcards or use the '|' character to separate them. This attribute should pattern match to the value of

uname -s [: getconf KERNEL_BITS]

on the supported target system(s).

S

os_release
Defines the operating system release(s) on which the product will run. (If not specified, **swpackage** assigns a value of "*", meaning the product runs on all releases.) If there are multiple operating system releases, use wildcards or use the '|' character to separate them. This attribute should pattern match to the value of **uname -r** on the supported target system(s).

os_version
Defines the operating system version(s) on which the product will run. (If not specified, **swpackage** assigns a value of "*", meaning the product runs on all versions.) If there are multiple operating system versions, use wildcards or use the '|' character to separate them. This attribute should pattern match to the value of **uname -v** on the supported target system(s).

readme
Defines the README information for the product or bundle; the value must be a pointer to the filename containing the text.

revision
Defines the revision (release number, version number) of the product or bundle.

title
Defines the full name (one-line description) of the product or bundle.

vendor_tag
Associates this product or bundle with the last defined vendor object, if that object has a matching **tag** attribute.

end Ends the product or bundle specification. This keyword is optional.

Subproduct Specification

The following is an example of a subproduct specification:

```
subproduct
    tag                    Manager
    contents               commands agent data man
    description            < data/description.manager
    title                  Management Utilities
end
```

Each keyword defines an attribute of a subproduct object. If a subproduct is specified, **swpackage** requires the **subproduct**, **tag**, and **contents** keywords.

subproduct
Keyword that begins the subproduct specification.

tag Defines the identifier (short name) for the subproduct.

contents
Defines the filesets that make up the subproduct. The value is a whitespace separated list of fileset **tag** values. In the PSF, fileset definitions are not contained within subproduct definitions. The **contents** keyword is used to assign filesets to subproducts.

description
Defines the multi-paragraph description of the subproduct; the value is either the text itself (within double-quotes) or a pointer to the filename containing the text.

title
Defines the full name (one-line description) of the subproduct.

end Ends the subproduct specification. This keyword is optional.

Fileset Specification

The following is an example of a fileset specification:

```
fileset
    tag                    commands
    ancestor               newprod.fs
    architecture           HP-UX_B.11.00_32/64
    category_tag           system_mgt
    description            < data/description.commands
    is_kernel              false
    is_patch               false
    is_reboot              false
    is_sparse              false
    machine_type           9000/[78]*:*
    os_name                HP-UX
    os_release             ?.11.*
    os_version             ?
    revision               2.15
    supersedes             product.fileset,fr=revision
    title                  Commands (management utilities)
```

[<*control file specifications*>]

[<*dependency specifications*>]

[<*file specifications*>]

end

Each keyword defines an attribute of a fileset object. For each fileset specified, **swpackage** requires the *fileset* and *tag* keywords, plus zero or more file specifications.

S

fileset
Keyword that begins fileset specification.

tag Defines the identifier (short name) for the fileset.

architecture
Describes the target system(s) on which the fileset will run if filesets for multiple architecture are included in a single product. Provides a human-readable summary of the four *uname(1)* attributes which define the exact target system(s) the product supports. Many filesets do not include an architecture; only a product architecture need be defined.

ancestor
A list of filesets that will match the current fileset when installed on a target system, if the **match_target** installation option is specified. Also determines the base to which a patch is applied.

category_tag
A repeatable tag-based attribute identifying a set of categories of which the software object is a member. This is used as a selection mechanism and can be used independent of patches. The default value is an empty list or **patch** if the **is_patch** attribute is set to **true**.

Like **vendor_tag**, this attribute can be used as a pointer to a category object that contains additional information about the category (for example, a one-line *title* definition and a *description* of the category).

Note that the category tag **patch** is reserved. When **is_patch** is set to **true**, a built-in **category_tag** attribute of value **patch** is automatically included.

NOTE: You can only change the **patch** value by performing a **swpackage** operation or by using **swmodify** to change the value of the **is_patch** attribute.

description
Defines the multi-paragraph description of the fileset; the value is either the text itself (within double-quotes) or a pointer to the filename containing the text.

is_kernel
A value of "true" defines the fileset as being a contributor to the operating system kernel; the target system(s) kernel build process will be invoked after the fileset is installed. If this attribute is not specified, **swpackage** assumes a default value of "false".

is_patch
Identifies a software object as a patch. The default value is **false**. When set to **true**, a built-in **category_tag** attribute of value **patch** is automatically included.

is_reboot
A value of "true" declares that the fileset requires a system reboot after installation. If this attribute is not specified, **swpackage** assumes a default value of "false".

S

is_sparse
Indicates that a fileset contains only a subset of files in the base (ancestor) fileset and that the contents are to be merged with the base fileset. The default value is **false**. If the **is_patch** attribute is **true**, **is_sparse** is also set to **true** for the fileset, although it can be forced to false.

machine_type
Defines the machine(s) on which the files will run if a fileset architecture has been defined. (If not specified, **swpackage** assigns a value of "*", meaning the files run on all machines.) If there are multiple machine platforms, use wildcards or use the '|' character to separate them. This attribute should pattern match to the value of

```
uname -m [: getconf HW_CPU_SUPP_BITS]
```

on the supported target machine(s).

os_name
Defines the operating system(s) on which the files will run if a fileset architecture has been defined. (If not specified, **swpackage** assigns a value of "*", meaning the files run on all operating systems.) If there are multiple operating systems, use wildcards or use the '|' character to separate them. This attribute should pattern match to the value of

uname -s [: getconf KERNEL_BITS]

on the supported target system(s).

os_release
Defines the operating system release(s) on which the files will run. (If not specified, **swpackage** assigns a value of "*", meaning the files run on all releases.) If there are multiple operating system releases, use wildcards or use the '|' character to separate them. This attribute should pattern match to the value of **uname -r** on the supported target system(s).

os_version
Defines the operating system version(s) on which the files will run. (If not specified, **swpackage** assigns a value of "*", meaning the files runs on all versions.) If there are multiple operating system versions, use wildcards or use the '|' character to separate them. This attribute should pattern match to the value of **uname -v** on the supported target system(s).

revision
Defines the revision (release number, version number) of the fileset.

supersedes
Used when a patch is replaced by (or merged into) a later patch. The attribute indicates which previous patches are replaced by the patch being installed or copied. This attribute value is a list of software specifications of other patches that this patch "supersedes".

title
Defines the full name (one-line description) of the fileset.

end Ends the fileset specification. This keyword is optional.

Dependency Specification

The following is an example of a dependency specification:

```
corequisites                SD.data
  ...
prerequisites               productA,r>=2.1
  ...
exrequisites                productB,r>=2.1
  ...
```

Each keyword/value defines a dependency relationship on another software object. The object can be within the same product as the dependent fileset, or it can be (within) another product. Dependency specifications are optional. Multiple dependency specifications are allowed.

corequisites
A list of dependencies on software that must be installed before this software is run. See also the **ancestor**, **exrequisites**, and **prerequisites** attributes.

prerequisites
A list of dependencies on software that must be installed before this software can be installed. See also the **ancestor**, **corequisites**, and **exrequisites** attributes.

exrequisites
(Not yet implemented.) A list of dependencies on software that may not be installed when this software is installed. See also the **ancestor**, **corequisites**, and **prerequisites** attributes.

S

Control Script Specification

The following is an example of a control script specification:

```
checkinstall        scripts/checkinstall
checkremove         scripts/checkremove
control_file        scripts/subscripts [=tag]"
configure           scripts/configure
postinstall         scripts/postinstall
postremove          scripts/postremove
preinstall          scripts/preinstall
preremove           scripts/preremove
request             scripts/request
unconfigure         scripts/unconfigure
unpostinstall       scripts/postinstall
unpreinstall        scripts/preinstall
verify              scripts/verify
```

Each script specification defines a control script object. The value of each keyword is the source *filename* for the control file.

Control scripts are optional. If present, **swpackage** will copy the specified source *filename* into the depot's storage directory for the associated product or fileset.

checkinstall
Defines the installation check script executed by **swinstall**. This script is executed during the analysis of each target, and it checks that the installation can be attempted. If the product or fileset check script returns 1 (ERROR), the product or fileset (respectively) will not be installed. If it returns 11 (GLOBAL_ERROR), no products will be installed.

checkremove
Defines the remove check script executed by **swremove**. This script is executed during the analysis of each target, and it checks that the remove can be attempted. If the check script returns 1 (ERROR), the product or fileset will not be removed.

control_file
Defines an arbitrary control file to be included with the product or fileset and stored alongside the named control files. It is used to include a subscript called by the named scripts or a data file read by these scripts. If the optional **tag** component of the value is not specified, **swpackage** uses the *basename*(1) of the source **filename** as the **tag** for the control file. Otherwise, the **tag** value is used.

configure
Defines the configuration script executed by **swinstall** and **swconfig**. This script configures the target host for the product or fileset (and the product or fileset for any required information about the target host).

postinstall
Defines the installation post-load script executed by **swinstall**. A fileset script is executed immediately after the fileset files are loaded. A product script is executed after all filesets for that product have been installed.

postremove
Defines the post-remove script executed by **swremove**. A fileset script is executed immediately after the fileset files are removed. A product script is executed after all filesets for that product have been removed.

preinstall
Defines the installation pre-load script executed by **swinstall**. A fileset script is executed immediately before the fileset files are loaded. A product script is executed before any filesets for that product have been installed.

preremove
Defines the pre-remove script executed by **swremove**. A fileset script is executed immediately before the fileset files are removed. A product script is executed before any filesets for that product have been removed.

request
The only script that may be interactive. This script may be run by **swask, swinstall,** or **swconfig** after selection and before the analysis phase in order to request information from the administrator that will be needed for the *configure_script* when that script is run later. The *request_script* writes all information into the *response_file*, which the scripts can then use.

unconfigure
Defines the un-configuration script executed by **swremove** and **swconfig**. This script unconfigures the target host for the product or fileset, undoing the configuration performed by the **configure** script.

unpostinstall
Defines the installation pre-restore script executed by **swinstall**. A fileset script is executed immediately before the fileset files are restored if there is an error and the **autorecover_product** option is set to true. Note that **unpostinstall** scripts are supported for filesets only. It should undo the steps taken by the **postinstall** script.

unpreinstall
Defines the installation post-restore script executed by **swinstall**. A fileset script is executed immediately after the fileset files are restored if there is an error and the **autorecover_product** option is set to true. A product script is executed after all filesets for that product have been restored. It should undo the steps taken by the **preinstall** scripts.

verify
Defines the verification script executed by **swverify**. This script verifies the configuration performed by the **configure** script.

File Specification

Within a fileset specification, the user can specify the following file types to be packaged into the fileset by **swpackage**:

control file
directory
hard link
regular file
symbolic link

If a recognized, unpackageable type or an unrecognized type is specified, an error is issued.

The **swpackage** command supports these mechanisms for specifying the files contained in a fileset:

Default permission specification
For some or all of the files and directories in the fileset, the user can define a default set of permissions.

Directory mapping
The user can point **swpackage** at a source directory in which the fileset's files are located. In addition, the user can map this source directory to the appropriate (destination) directory in which this subset of the product's files will be located.

Explicit file specification
For some or all of the files and directories in the fileset, the user can name each source file and destination location.

Recursive (implicit) file specification
If a directory mapping is active, the user can tell **swpackage** to include all files and directories in the fileset (recursively) below the specified directory.

These mechanisms can all be used in combination with the others.

Directory Mapping

The **directory** *source*[**=***destination*] keyword defines a **source** directory under which subsequently listed files are located. In addition, the user can map the **source** directory to a **destination** directory under which the packaged files will be installed. For example, the definition:

```
directory   ./commands = /usr/sbin
```

causes all files from the **./commands** directory to have the prefix **/usr/sbin** when installed. The **destination** directory must be a located within the **product.directory** attribute, if defined. (This attribute is defined by the **directory** keyword in the product specification.)

The *destination* directory must be an absolute pathname.

The **directory** keyword is optional.

Recursive File Specification

The **file*** keyword directs **swpackage** to recursively include every file (and directory) within the current source directory in the fileset. (Partial wildcarding is not supported—e.g., **file dm*** to indicate all files starting with "dm".)

The **directory** keyword must have been previously specified before the **file *** specification can be used.

All attributes for the destination file object are taken from the source file, unless a **file_permissions** keyword is active (this keyword is described below).

The user can specify multiple

directory *source*[**=***destination*]
file *

pairs to gather files from different source directories into a single fileset.

Explicit File Specification

Instead of, or in addition to, the recursive file specification, the user can explicitly specify the files and directories to be packaged into a fileset.

The user can use the *directory* keyword to define a source (and destination) for explicitly specified files. If no *directory* keyword is active, then the source path and the absolute destination path must be specified for each file.

An explicit file specification uses this form:

file [**-m** *mode*] [**-o** [*owner*[**,**]][*uid*]] [**-g** [*group*[**,**]][*gid*]] [**-t** *type*]
[**-v**] [*source*] [*destination*]

file
Specifies an existing file or directory (perhaps within the currently active source directory) to include in the fileset.

S

source
Defines the relative or absolute path to the source file.

If this is a relative path, **swpackage** will search for it relative to the source directory set by the *directory* keyword. If no source directory is active, **swpackage** will search for it relative to the current working directory in which the command was invoked.

All attributes for the destination file object are taken from the source file, unless a **file_permissions** keyword is active, or the **-m**, **-o**, or **-g**, options are also included in the file specification.

destination
Defines the destination path at which the file will be installed. If **destination** is a relative path, the active destination directory set by the **directory** keyword will be prefixed to it. If it is a relative path, and no destination directory is active, **swpackage** generates an error. If the destination is not specified, the *source* is used as the *destination*, with the appropriate mapping done with the active destination directory (if any).

-m *mode*
Defines the (octal) mode of a file or directory.

-o [*owner*[**,**]][*uid*]
Defines the destination file's owner name and/or or uid. If only the owner is specified, the *owner* and *uid* attributes are set for the destination file object, based on the packaging host's **/etc/passwd**. If only the uid is specified, it is set as the *uid* attribute for the destination object and no owner name is assigned. If both are specified, each sets the corresponding attribute for the file object. During an installation, the *owner* attribute is used to set the owner name and uid, unless the owner name is not defined in the target system's **/etc/passwd**. In this case, the *uid* attribute is used to set the uid.

-g [*group*[**,**]][*gid*]
Defines the destination file's group name and/or or gid. If only the group is specified, the *group* and *gid* attributes are set for the destination file object, based on the packaging host's **/etc/group**. If only the group is specified, and it contains digits only, it is interpreted as the gid, and is set as the *gid* attribute for the destination object; no group name is assigned to the object. If both are specified, each sets the corresponding attribute for the file object. During an installation, the *group* attribute is used to set the group name and gid, unless the group name is not defined in the target system's **/etc/group**. In this case, the *gid* attribute is used to set the gid.

-t *type*
Defines a file of type **d** (directory), **s** (symbolic), or **h** (hard link), for files that need not exist before packaging.

-v Marks the file as volatile, meaning it can be modified (i.e. deleted) after installed without impacting the fileset.

When processing existing files in a source directory, a number of problems may be encountered. Errors or warning messages are printed for each problem. (The **swpackage** command terminates when errors are encountered in reading the PSF or accessing the source files.)

Example File Specifications

The following examples illustrate the use of the **directory** and **file** keywords:

Include all files under **./commands/**, to be rooted under **/usr/sbin/**:

```
directory ./commands=/usr/sbin
file *
```

Include only certain files under **./commands/** and **./nls**, to be rooted under **/usr/sbin/** and **/var/lib/nls/C/**:

```
directory ./commands=/usr/sbin
file sbin/swinstall
file sbin/swcopy
...
directory ./nls=/usr/lib/nls/C/
file swinstall.cat
file swremove.cat
...
```

Explicitly list files and directories, no directory mapping specified:

```
file ./commands/swinstall /usr/sbin/swinstall
...
file ./nls  /usr/lib/nls/C
file ./nls/swinstall.cat  /usr/lib/nls/C/swinstall.cat
```

Use all specification types to include files:

```
directory ./commands=/usr/sbin
file *
directory ./nls=/usr/lib/nls/C
file swinstall.cat
...
file ./obam/obam.dm  /etc/interface.lib/obam/obam.dm
```

S

Redefine specific files previously defined using **file *** (e.g. to set explicit attributes):

```
directory ./commands=/usr/sbin
file *
file -m 04500 swcommand
file -o adm -g sys swfile
```

Default Permission Specification

By default, a destination file object will inherit the mode, owner, and group of the source file. The **file_permissions** keyword can be specified to set a default permission umask/mode, owner, and group for all the files being packaged into the fileset:

> **file_permissions** [**-m** *mode* **|-u** *umask*] [**-o**[*owner*[**,**]][*uid*]] ****
> [**-g**[*group*[**,**]][*gid*]] [**-t** *type*]

file_permissions
: Applies only to the fileset it is defined in. Multiple **file_permissions** can be specified, later definitions simply replace previous definitions.

-m *mode*
: Defines a default (octal) mode for all file objects.

-u *umask*
: Instead of specifying an octal mode as the default, the user can specify an octal *umask*(1) value which gets "subtracted" from an existing source file's mode to generate the mode of the destination file.

  By specifying a **umask**, the user can set a default mode for executable files, non-executable files, and directories. (A specific mode can be set for any file, as described above.)

-o [*owner*[**,**]][*uid*]
: Defines the destination file's owner name and/or or uid (as defined above).

-g [*group*[**,**]][*gid*]
: Defines the destination file's group name and/or or gid (as defined above).

-t *type*
: Defines files that need not exist before packaging.

Example Permission Specifications

The following examples illustrate the use of the *file_permission* keyword.

Set a read only 444 mode for all file objects (requires override for every executable file and directory):

```
file_permissions  -m 444
```

Set a read mode for non-executable files, and a read/execute mode for executable files and for directories:

```
file_permissions  -u 222
```

Set the same mode defaults, plus an owner and group:

```
file_permissions  -u 222  -o bin  -g bin
```

Set the same mode defaults, plus a uid and gid:

```
file_permissions  -u 222  -o 2  -g 2
```

Set the owner write permission in addition to the above:

```
file_permissions  -u 022  -o 2  -g 2
```

If the user defines no *file_permissions*, **swpackage** uses the default value:

```
file_permissions -u 000
```

for destination file objects based on existing source files. (Meaning the mode, owner/uid, group/gid are set based on the source file, unless specific overrides are specified for a destination file.)

EXAMPLES

This example illustrates a typical PSF.

```
# PSF file which defines an example product.
depot
  layout_version   1.0

# Vendor definition:
vendor
  tag           HP
  title         Hewlett-Packard Company
  description   < data/descr.hp
category
  tag           system_mgt
  title         Systems Management Applications
  description   These are the system management applications
  revision      1.0
end

# Product definition:
product
  tag            SD
  revision       A.01.00
  architecture   HP-UX_B.11.00_32/64
  vendor_tag     HP

  title          HP OpenView Software Distributor
  number         B1991A
  category_tag   system_mgt

  description    < data/descr.sd
  copyright      < data/copyr.sd
  readme         < data/README.sd

  machine_type   *
  os_name        HP-UX
  os_release     ?.11.*
  os_version     ?

  directory      /
  is_locatable   false

  # Create a product script which executes during the swremove
  # analysis phase.  (This particular script returns an ERROR,
  # which prevents the removal of the SD product.)
  checkremove     scripts/checkremove.sd

  # Subproduct definitions:

  subproduct
    tag          Manager
    title        Management Utilities
    contents     commands agent data man
  end

  subproduct
    tag          Agent
    title        Agent component
    contents     agent data man
  end
```

S

```
    # Fileset definitions:
    fileset
      tag           commands
      title         SD Commands (management utilities)
      revision      2.42

      description   < data/descr.commands

      # Dependencies
      corequisites SD.data
      corequisites SD.agent

      # Control files:
      configure     scripts/configure.commands

      # Files:
      directory     ./commands=/usr/sbin
      file          swinstall
      file          swcopy
      ...

      directory     ./nls=/usr/lib/nls/C
      file          swinstall.cat
      file          swpackage.cat

      directory     ./ui=/usr/lib/sw/ui
      file          *

      ...

    end # commands

    ... # other filesets

    fileset
      tag           man
      title         Manual pages for the Software Distributor
      revision      2.05

      directory     ./man/man1m=/usr/man/man1m.Z
      file          *

      directory     ./man/man4=/usr/man/man4.Z
      file          *

      directory     ./man/man5=/usr/man/man5.Z
      file          *

    end  # man

  end  # SD
```

S

AUTHOR

swpackage was developed by the Hewlett-Packard Company and Mark H. Colburn (see *pax*(1)).

SEE ALSO

The *Managing HP-UX Software with SD-UX* manual, the *HP OpenView Software Distributor Administrator's Guide*, swpackage(1M), sd(4), sd(5), swacl(1M), swagentd(1M), swconfig(1M), swcopy(1M), swgettools(1M), swinstall(1M), swjob(1M), swlist(1M), swmodify(1M), swreg(1M), swremove(1M), swverify(1M).

NAME

symlink - symbolic link

DESCRIPTION

A **symbolic** (or **soft**) link is a file whose name indirectly refers (points) to a relative or absolute path name.

During path name interpretation, a symbolic link to a relative path name is expanded to the path name being interpreted, and a symbolic link to an absolute path name is replaced with the path name being interpreted.

Thus, given the path name **/a/b/c/d**:

> If **c** is a symbolic link to a relative path name such as **../x/y**, the path name is interpreted as **/a/b/../x/y/d**.
>
> If **c** is a symbolic link to an absolute path name such as **/v/w**, the path name is interpreted as **/v/w/d**.

All symbolic links are interpreted in this manner, with one exception: when the symbolic link is the last component of a path name, it is passed as a parameter to one of the system calls: **readlink**, **rename**, **symlink**, **unlink**, **chown**, or **lstat** (see *readlink*(2), *rename*(2), *symlink*(2), *unlink*(2), *chown*(2) and *lstat*(2)). With these calls, the symbolic link, itself, is accessed or affected.

Unlike normal (hard) links, a symbolic link can refer to any arbitrary path name and can span different logical devices (volumes).

The path name can be that of any type of file (including a directory or another symbolic link), and may be invalid if no such path exists in the system. (It is possible to make symbolic links point to themselves or other symbolic links in such a way that they form a closed loop. The system detects this situation by limiting the number of symbolic links it traverses while translating a path name.)

The mode and ownership of a symbolic link is ignored by the system, which means that **chmod** affects the actual file, but not the file containing the symbolic link (see *chmod*(1)).

Symbolic links can be created using **ln** or **symlink** (see *ln*(1) and *symlink*(2)).

AUTHOR

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

SEE ALSO

cp(1), symlink(2), readlink(2), link(2), stat(2), mknod(1M).

S

NAME

tar - format of tar tape archive

DESCRIPTION

The *header* structure produced by **tar** (see *tar*(1)) is as follows (the array size defined by the constants is shown on the right):

```
struct {
    char name[NAMSIZ];            (100)
    char mode[MODE_SZ];             (8)
    char uid[UID_SZ];               (8)
    char gid[GID_SZ];               (8)
    char size[SIZE_SZ];            (12)
    char mtime[MTIME_SZ];          (12)
    char chksum[CHKSUM_SZ];         (8)
    char typeflag;
    char linkname[NAMSIZ];        (100)
    char magic[MAGIC_SZ];           (6)
    char version[VERSION_SZ];       (2)
    char uname[UNAME_SZ];          (32)
    char gname[GNAME_SZ];          (32)
    char devmajor[DEV_SZ];          (8)
    char devminor[DEV_SZ];          (8)
    char prefix[PREFIX_SZ];       (155)
} dbuf;
```

All characters are represented in ASCII. There is no padding used in the header block; all fields are contiguous.

The fields *magic*, *uname*, and *gname* are null-terminated character strings. The fields *name*, *linkname*, and *prefix* are null-terminated character strings except when all characters in the array contain non-null characters, including the last character. The *version* field is two bytes containing the characters **00** (zero-zero). The *typeflag* contains a single character. All other fields are leading-zero-filled octal numbers in ASCII. Each numeric field is terminated by one or more space or null characters.

The *name* and the *prefix* fields produce the pathname of the file. The hierarchical relationship of the file is retained by specifying the pathname as a path prefix, with a slash character and filename as the suffix. If the *prefix* contains non-null characters, *prefix*, a slash character, and *name* are concatenated without modification or addition of new characters to produce a new pathname. In this manner, pathnames of at most 256 characters can be supported. If a pathname does not fit in the space provided, the format-creating utility notifies the user of the error, and no attempt is made to store any part of the file, header, or data on the medium.

SEE ALSO

tar(1)

t

STANDARDS CONFORMANCE

tar: XPG4, FIPS 151-2, POSIX.1

NAME

term - format of compiled term file

SYNOPSIS

term

DESCRIPTION

Compiled terminfo descriptions are placed under the directory **/usr/share/lib/terminfo**. In order to avoid a linear search of a huge HP-UX system directory, a two-level scheme is used: **/usr/share/lib/terminfo/***c***/***name* where *name* is the name of the terminal, and *c* is the first character of *name*. Thus, **hp110** can be found in the file **/usr/share/lib/terminfo/h/hp110**. Synonyms for the same terminal are implemented by multiple links to the same compiled file.

The format has been chosen so that it is the same on all hardware. An 8-bit or longer byte is assumed, but no assumptions about byte ordering or sign extension are made.

The compiled file is created using the **tic** program (see *tic*(1M)), and read by the **setupterm()** routine. Both of these pieces of software are part of the *curses*(3X) package. The file is divided into the following six parts:

1. The header section begins the file and contains six short integers in the following format:

    1. Magic number (octal 0432);
    2. Size, in bytes, of the names section;
    3. Number of bytes in the Boolean section;
    4. Number of short integers in the numbers section;
    5. Number of offsets (short integers) in the strings section;
    6. Size, in bytes, of the string table.

    Short integers are stored in two 8-bit bytes. The first byte contains the least significant 8 bits of the value; the second byte contains the most significant 8 bits. (Thus, the value represented is 256 * second + first.) The value −1 is represented by **0377**, **0377**; other negative values are illegal. The −1 generally means that a capability is missing from this terminal. Note that this format corresponds to the hardware of the VAX and PDP-11. Machines where this does not correspond to the hardware read the integers as two bytes and compute the result.

2. The terminal names section comes next. It contains the first line of the terminfo description, listing the various names for the terminal, separated by the **|** character. The section is terminated with an ASCII NUL character.

3. In the Boolean section, the Boolean flags have one byte for each flag. This byte is either **0** or **1** as the flag is absent or present, respectively. The capabilities are in the same order as they are listed in the file <**term.h**>.

    Between the Boolean section and the number section, a null byte will be inserted, if necessary, to ensure that the number section begins on an even byte. All short integers are aligned on a short word boundary.

4. The numbers section is similar to the flags section. Each capability consists of two bytes, and is stored as a short integer. If the value represented is −1, the capability is considered missing.

5. The strings section is also similar. Each capability is stored as a short integer in the format above. A value of −1 means the capability is missing. Otherwise, the value is taken as an offset from the beginning of the string table. Special characters in **^***X* or *****c* notation are stored in their interpreted form, not the printing representation. Padding information **$***nn* and parameter information **%***x* are stored intact in uninterpreted form.

6. The final section is the string table. It contains all the values of string capabilities referenced in the string section. Each string is null terminated.

Note that it is possible for **setupterm()** to expect a different set of capabilities than are actually present in the file. Either the database might have been updated since **setupterm()** has been recompiled (resulting in extra unrecognized entries in the file) or the program may have been recompiled more recently than the database was updated (resulting in missing entries). The routine **setupterm()** must be prepared for both possibilities, which is why the numbers and sizes are included. Also, new capabilities must always be added at the end of the lists of Boolean, number, and string capabilities.

The following example is an octal dump of the description for the HP Portable Computer (HP-110):

t

```
110|hp110|hp110a portable computer,
    am, xhp, da, db, mir, cols#80, lines#16, lm#0,
    cbt=\Ei, bel=^G, cr=\r, tbc=\E3, clear=\E&a0y0C\EJ,
    el=\EK, ed=\EJ, hpa=\E&a%p1%dC, cup=\E&a%p1%dy%p2%dC,
    cud1=\EB, cub1=\b, cuf1=\EC, cuu1=\EA, cvvis=\E&j@,
    dch1=\EP, dl1=\EM, smir=\EQ, smso=\E&dB, sgr0=\E&d@,
    rmir=\ER, rmso=\E&d@, is2=\E&j@,
    if=/usr/share/lib/tabset/stdcrt, il1=\EL, kbs=\b, kcud1=\EB,
    khome=\Eh, kcub1=\ED, kcuf1=\EC, kcuu1=\EA, rmkx=\E&s0A,
    smkx=\E&s1A, vpa=\E&a%p1%dY, ind=\n, hts=\E1, ht=\t,

0000 032 001   #  \0 025  \0  \b  \0 223  \0 254  \0   1   1   0   |
0020   h   p   1   1   0   |   h   p   1   1   0   a       p   o   r
0040   t   a   b   l   e       c   o   m   p   u   t   e   r  \0  \0
0060 001  \0 001  \0  \0  \0  \0  \0  \0  \0 001 001 001  \0  \0  \0
0100  \0  \0  \0  \0   P  \0 377 377 020  \0  \0  \0 377 377 377 377
0120 377 377 377 377  \0  \0 003  \0 005  \0 377 377 007  \0  \n  \0
0140 024  \0 027  \0 032  \0 377 377   $  \0   4  \0 377 377 377 377
0160   7  \0 377 377 377 377   9  \0 377 377   <  \0   ?  \0   D  \0
0200   G  \0 377 377 377 377 377 377 377 377 377 377 377 377 377 377
0220 377 377   J  \0 377 377 377 377 377 377   M  \0 377 377 377 377
0240 377 377   R  \0 377 377 377 377   W  \0   Z  \0 377 377 377 377
0260 377 377 377 377 377 377   _  \0 377 377   d  \0 377 377   {  \0
0300 377 377   ~  \0 377 377 377 377 377 377 377 377 377 377 200  \0
0320 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377
0340 377 377 377 377 377 377 377 377 377 377 377 377 203  \0 377 377
0360 377 377 206  \0 377 377 377 377 377 377 211  \0 377 377 377 377
0400 377 377 214  \0 217  \0 225  \0 377 377 377 377 377 377 377 377
0420 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377

0520 377 377 233  \0 377 377 245  \0 377 377 377 377 247  \0 377 377
0540 252  \0 377 377 377 377 377 377 377 377 377 377 377 377 377 377
0560 377 377 377 377 377 377 377 377 377 377 033   i  \0 007  \0  \r
0600  \0 033   3  \0 033   &   a   0   y   0   C 033   J  \0 033   K
0620  \0 033   J  \0 033   &   a   %   p   1   %   d   C  \0 033   &
0640   a   %   p   1   %   d   y   %   p   2   %   d   C  \0 033   B
0660  \0  \b  \0 033   C  \0 033   A  \0 033   &   j   @  \0 033   P
0700  \0 033   M  \0 033   Q  \0 033   &   d   B  \0 033   &   d   @
0720  \0 033   R  \0 033   &   d   @  \0 033   &   j   @  \0   /   u
0740   s   r   /   l   i   b   /   t   a   b   s   e   t   /   s   t
0760   d   c   r   t  \0 033   L  \0  \b  \0 033   B  \0 033   h  \0
1000 033   D  \0 033   C  \0 033   A  \0 033   &   s   0   A  \0 033
1020   &   s   1   A  \0 033   &   a   %   p   1   %   d   Y  \0  \n
1040  \0 033   1  \0  \t  \0
1046
```

WARNINGS

Total compiled entries cannot exceed 4096 bytes.

The name field cannot exceed 128 bytes.

Hewlett-Packard Company supports only those terminals that are listed on the current list of supported devices. However, both non-supported and supported terminals may be in the terminfo database. If non-supported terminals are used, they may not work correctly.

FILES

/usr/share/lib/terminfo/?/* compiled terminal capability data base

SEE ALSO

tic(1M), untic(1M), curses(3X), terminfo(4).

NAME

term.h - terminal capabilities

DESCRIPTION

The header <**term.h**> contains definitions for each of the following symbolic constants and capability names in the following tables.

In the following table, a **Variable** is the name by which a **C** programmer accesses a capability (at the **terminfo** level). A **Capname** is the short name for a capability specified in the **terminfo** source file. It is used by a person updating the source file and by the **tput** command.

Booleans

Variable	Cap-name	Termcap Code	Description
auto_left_margin	**bw**	**bw**	**cub1** wraps from column 0 to last column
auto_right_margin	**am**	**am**	Terminal has automatic margins
back_color_erase	**bce**	**ut**	Screen erased with background color
buttons	**btns**	**BT**	Number of buttons on the mouse
can_change	**ccc**	**cc**	Terminal can re-define existing color
ceol_standout_glitch	**xhp**	**xs**	Standout not erased by overwriting (hp)
col_addr_glitch	**xhpa**	**YA**	Only positive motion for **hpa**/**mhpa** caps
cpi_changes_res	**cpix**	**YF**	Changing character pitch changes resolution
create_window	**cwin**	**CW**	Define win #1 to go from #2,#3 to #4,#5
cr_cancels_micro_mode	**crxm**	**YB**	Using **cr** turns off micro mode
dest_tabs_magic_smso	**xt**	**xt**	Destructive tabs, magic **smso** char (t1061)
dial_phone	**dial**	**DI**	Dial phone number #1
display_clock	**dclk**	**DK**	Display time-of-day clock
eat_newline_glitch	**xenl**	**xn**	Newline ignored after 80 columns (Concept)
erase_overstrike	**eo**	**eo**	Can erase overstrikes with a blank
fixed_pause	**pause**	**PA**	Pause for 2-3 seconds
flash_hook	**hook**	**fh**	Flash the switch hook
generic_type	**gn**	**gn**	Generic line type (e.g., dialup, switch)
get_mouse	**getm**	**Gm**	Curses should get button events
goto_window	**wingo**	**WG**	Got to window #1
hangup	**hup**	**HU**	Hang-up phone
hard_copy	**hc**	**hc**	Hardcopy terminal
hard_cursor	**chts**	**HC**	Cursor is hard to see
has_meta_key	**km**	**km**	Has a meta key (shift, sets parity bit)
has_print_wheel	**daisy**	**YC**	Printer needs operator to change character set
has_status_line	**hs**	**hs**	Has extra "status line"
hue_lightness_saturation	**hls**	**hl**	Terminal uses only HLS color notation (Tektronix)
insert_null_glitch	**in**	**in**	Insert mode distinguishes nulls
lpi_changes_res	**lpix**	**YG**	Changing line pitch changes resolution
memory_above	**da**	**da**	Display may be retained above the screen
memory_below	**db**	**db**	Display may be retained below the screen
move_insert_mode	**mir**	**mi**	Safe to move while in insert mode
move_standout_mode	**msgr**	**ms**	Safe to move in standout modes
needs_xon_xoff	**nxon**	**nx**	Padding won't work, xon/xoff required
no_esc_ctlc	**xsb**	**xb**	Beehive (f1=escape, f2=ctrl C)
no_pad_char	**npc**	**NP**	Pad character doesn't exist
non_dest_scroll_region	**ndscr**	**ND**	Scrolling region is nondestructive
non_rev_rmcup	**nrrmc**	**NR**	**smcup** does not reverse **rmcup**
over_strike	**os**	**os**	Terminal overstrikes on hard-copy terminal
print_rate	**cps**	**Ym**	Print rate in characters per second
prtr_silent	**mc5i**	**5i**	Printer won't echo on screen
row_addr_glitch	**xvpa**	**YD**	Only positive motion for **vpa**/**mvpa** caps
semi_auto_right_margin	**sam**	**YE**	Printing in last column causes **cr**
set_pglen_inch	**slength**	**YI**	Set page length to #1 hundredth of an inch (use tparm)
status_line_esc_ok	**eslok**	**es**	Escape can be used on the status line
tilde_glitch	**hz**	**hz**	Hazeltine; can't print tilde (~)
transparent_underline	**ul**	**ul**	Underline character overstrikes
xon_xoff	**xon**	**xo**	Terminal uses xon/xoff handshaking

t

Numbers

Variable	Cap-name	Termcap Code	Description
bit_image_entwining	bitwin	Yo	Number of passes for each bit-map row
bit_image_type	bitype	Yp	Type of bit image device
buffer_capacity	bufsz	Ya	Number of bytes buffered before printing
columns	cols	co	Number of columns in a line
dot_horz_spacing	spinh	Yc	Spacing of dots horizontally in dots per inch
dot_vert_spacing	spinv	Yb	Spacing of pins vertically in pins per inch
init_tabs	it	it	Tabs initially every # spaces
label_height	lh	lh	Number of rows in each label
label_width	lw	lw	Number of columns in each label
lines	lines	li	Number of lines on a screen or a page
lines_of_memory	lm	lm	Lines of memory if > **lines**; **0** means varies
max_attributes	ma	ma	Maximum combined video attributes terminal can display
magic_cookie_glitch	xmc	sg	Number of blank chars left by **smso** or **rmso**
max_colors	colors	Co	Maximum number of colors on the screen
max_micro_address	maddr	Yd	Maximum value in **micro_..._address**
max_micro_jump	mjump	Ye	Maximum value in **parm_..._micro**
max_pairs	pairs	pa	Maximum number of color-pairs on the screen
maximum_windows	Wnum	MW	Maximum number of definable windows
micro_char_size	mcs	Yg	Character step size when in micro mode
micro_line_size	mls	Yf	Line step size when in micro mode
no_color_video	ncv	NC	Video attributes that can't be used with colors
num_labels	nlab	Nl	Number of labels on screen (start at 1)
number_of_pins	npins	Yh	Number of pins in print-head
output_res_char	orc	Yi	Horizontal resolution in units per character
output_res_line	orl	Yj	Vertical resolution in units per line
output_res_horz_inch	orhi	Yk	Horizontal resolution in units per inch
output_res_vert_inch	orvi	Yl	Vertical resolution in units per inch
padding_baud_rate	pb	pb	Lowest baud rate where padding needed
virtual_terminal	vt	vt	Virtual terminal number
wide_char_size	widcs	Yn	Character step size when in double wide mode
width_status_line	wsl	ws	Number of columns in status line

Strings

Variable	Cap-name	Termcap Code	Description
acs_chars	acsc	ac	Graphic charset pairs aAbBcC
alt_scancode_esc	scesa	S8	Alternate escape for scancode emulation (default is for vt100)
back_tab	cbt	bt	Back tab
bell	bel	bl	Audible signal (bell)
bit_image_carriage_return	bicr	Yv	Move to beginning of same row (use tparm)
bit_image_newline	binel	Zz	Move to next row of the bit image (use tparm)
bit_image_repeat	birep	Xy	Repeat bit-image cell #1 #2 times (use tparm)
carriage_return	cr	cr	Carriage return
change_char_pitch	cpi	ZA	Change number of characters per inch
change_line_pitch	lpi	ZB	Change number of lines per inch
change_res_horz	chr	ZC	Change horizontal resolution
change_res_vert	cvr	ZD	Change vertical resolution
change_scroll_region	csr	cs	Change to lines #1 through #2 (vt100)
char_padding	rmp	rP	Like **ip** but when in replace mode
char_set_names	csnm	Zy	List of character set names
clear_all_tabs	tbc	ct	Clear all tab stops
clear_margins	mgc	MC	Clear all margins (top, bottom, and sides)
clear_screen	clear	cl	Clear screen and home cursor
clr_bol	el1	cb	Clear to beginning of line, inclusive
clr_eol	el	ce	Clear to end of line

clr_eos	**ed**	**cd**	Clear to end of display
code_set_init	**csin**	**ci**	Init sequence for multiple codesets
color_names	**colornm**	**Yw**	Give name for color #1
column_address	**hpa**	**ch**	Horizontal position absolute
command_character	**cmdch**	**CC**	Terminal settable cmd character in prototype
cursor_address	**cup**	**cm**	Move to row #1 col #2
cursor_down	**cud1**	**do**	Down one line
cursor_home	**home**	**ho**	Home cursor (if no **cup**)
cursor_invisible	**civis**	**vi**	Make cursor invisible
cursor_left	**cub1**	**le**	Move left one space.
cursor_mem_address	**mrcup**	**CM**	Memory relative cursor addressing
cursor_normal	**cnorm**	**ve**	Make cursor appear normal (undo **vs/vi**)
cursor_right	**cuf1**	**nd**	Non-destructive space (cursor or carriage right)
cursor_to_ll	**ll**	**ll**	Last line, first column (if no **cup**)
cursor_up	**cuu1**	**up**	Upline (cursor up)
cursor_visible	**cvvis**	**vs**	Make cursor very visible
define_bit_image_region	**defbi**	**Yx**	Define rectangular bit-image region (use tparm)
define_char	**defc**	**ZE**	Define a character in a character set
delete_character	**dch1**	**dc**	Delete character
delete_line	**dl1**	**dl**	Delete line
device_type	**devt**	**dv**	Indicate language/codeset support
dis_status_line	**dsl**	**ds**	Disable status line
display_pc_char	**dispc**	**S1**	Display PC character
down_half_line	**hd**	**hd**	Half-line down (forward 1/2 linefeed)
ena_acs	**enacs**	**eA**	Enable alternate character set
end_bit_image_region	**endbi**	**Yy**	End a bit-image region (use tparm)
enter_alt_charset_mode	**smacs**	**as**	Start alternate character set
enter_am_mode	**smam**	**SA**	Turn on automatic margins
enter_blink_mode	**blink**	**mb**	Turn on blinking
enter_bold_mode	**bold**	**md**	Turn on bold (extra bright) mode
enter_ca_mode	**smcup**	**ti**	String to begin programs that use **cup**
enter_delete_mode	**smdc**	**dm**	Delete mode (enter)
enter_dim_mode	**dim**	**mh**	Turn on half-bright mode
enter_doublewide_mode	**swidm**	**ZF**	Enable double wide printing
enter_draft_quality	**sdrfq**	**ZG**	Set draft quality print
enter_horizontal_hl_mode	**ehhlm**	*n/a*	turn on horizontal highlight mode
enter_insert_mode	**smir**	**im**	Insert mode (enter)
enter_italics_mode	**sitm**	**ZH**	Enable italics
enter_left_hl_mode	**elhlm**	*n/a*	Turn on left highlight mode
enter_leftward_mode	**slm**	**ZI**	Enable leftward carriage motion
enter_low_hl_mode	**elohlm**	*n/a*	turn on low highlight mode
enter_micro_mode	**smicm**	**ZJ**	Enable micro motion capabilities
enter_near_letter_quality	**snlq**	**ZK**	Set near-letter quality print
enter_normal_quality	**snrmq**	**ZL**	Set normal quality print
enter_pc_charset_mode	**smpch**	**S2**	Enter PC character display mode
enter_protected_mode	**prot**	**mp**	Turn on protected mode
enter_reverse_mode	**rev**	**mr**	Turn on reverse video mode
enter_right_hl_mode	**erhlm**	*n/a*	turn on right highlight mode
enter_scancode_mode	**smsc**	**S4**	Enter PC scancode mode
enter_secure_mode	**invis**	**mk**	Turn on blank mode (characters invisible)
enter_shadow_mode	**sshm**	**ZM**	Enable shadow printing
enter_standout_mode	**smso**	**so**	Begin standout mode
enter_subscript_mode	**ssubm**	**ZN**	Enable subscript printing
enter_superscript_mode	**ssupm**	**ZO**	Enable superscript printing
enter_top_hl_mode	**ethlm**	*n/a*	Turn on top highlight mode
enter_underline_mode	**smul**	**us**	Start underscore mode
enter_upward_mode	**sum**	**ZP**	Enable upward carriage motion
enter_vertical_hl_mode	**evhlm**	*n/a*	turn on vertical highlight mode
enter_xon_mode	**smxon**	**SX**	Turn on xon/xoff handshaking
erase_chars	**ech**	**ec**	Erase #1 characters
exit_alt_charset_mode	**rmacs**	**ae**	End alternate character set
exit_am_mode	**rmam**	**RA**	Turn off automatic margins

t

exit_attribute_mode	**sgr0**	**me**	Turn off all attributes
exit_ca_mode	**rmcup**	**te**	String to end programs that use **cup**
exit_delete_mode	**rmdc**	**ed**	End delete mode
exit_doublewide_mode	**rwidm**	**ZQ**	Disable double wide printing
exit_insert_mode	**rmir**	**ei**	End insert mode
exit_italics_mode	**ritm**	**ZR**	Disable italics
exit_leftward_mode	**rlm**	**ZS**	Enable rightward (normal) carriage motion
exit_micro_mode	**rmicm**	**ZT**	Disable micro motion capabilities
exit_pc_charset_mode	**rmpch**	**S3**	Disable PC character display mode
exit_scancode_mode	**rmsc**	**S5**	Disable PC scancode mode
exit_shadow_mode	**rshm**	**ZU**	Disable shadow printing
exit_standout_mode	**rmso**	**se**	End standout mode
exit_subscript_mode	**rsubm**	**ZV**	Disable subscript printing
exit_superscript_mode	**rsupm**	**ZW**	Disable superscript printing
exit_underline_mode	**rmul**	**ue**	End underscore mode
exit_upward_mode	**rum**	**ZX**	Enable downward (normal) carriage motion
exit_xon_mode	**rmxon**	**RX**	Turn off xon/xoff handshaking
flash_screen	**flash**	**vb**	Visible bell (may not move cursor)
form_feed	**ff**	**ff**	Hardcopy terminal page eject
from_status_line	**fsl**	**fs**	Return from status line
init_1string	**is1**	**i1**	Terminal or printer initialization string
init_2string	**is2**	**is**	Terminal or printer initialization string
init_3string	**is3**	**i3**	Terminal or printer initialization string
init_file	**if**	**if**	Name of initialization file
init_prog	**iprog**	**iP**	Path name of program for initialization
initialize_color	**initc**	**IC**	Initialize the definition of color
initialize_pair	**initp**	**Ip**	Initialize color-pair
insert_character	**ich1**	**ic**	Insert character
insert_line	**il1**	**al**	Add new blank line
insert_padding	**ip**	**ip**	Insert pad after character inserted

The "**key_**" strings are sent by specific keys. The "**key_**" descriptions include the macro, defined in **<curses.h>**, for the code returned by the CURSES function **getch()** when the key is pressed [see *curs_getch*(3X)].

Variable	**Cap-name**	**Termcap Code**	**Description**
key_a1	**ka1**	**K1**	upper left of keypad
key_a3	**ka3**	**K3**	upper right of keypad
key_b2	**kb2**	**K2**	center of keypad
key_backspace	**kbs**	**kb**	sent by backspace key
key_beg	**kbeg**	**@1**	sent by beg(inning) key
key_btab	**kcbt**	**kB**	sent by back-tab key
key_c1	**kc1**	**K4**	lower left of keypad
key_c3	**kc3**	**K5**	lower right of keypad
key_cancel	**kcan**	**@2**	sent by cancel key
key_catab	**ktbc**	**ka**	sent by clear-all-tabs key
key_clear	**kclr**	**kC**	sent by clear-screen or erase key
key_close	**kclo**	**@3**	sent by close key
key_command	**kcmd**	**@4**	sent by cmd (command) key
key_copy	**kcpy**	**@5**	sent by copy key
key_create	**kcrt**	**@6**	sent by create key
key_ctab	**kctab**	**kt**	sent by clear-tab key
key_dc	**kdch1**	**kD**	sent by delete-character key
key_dl	**kdl1**	**kL**	sent by delete-line key
key_down	**kcud1**	**kd**	sent by terminal down-arrow key
key_eic	**krmir**	**kM**	sent by **rmir** or **smir** in insert mode
key_end	**kend**	**@7**	sent by end key
key_enter	**kent**	**@8**	sent by enter/send key
key_eol	**kel**	**kE**	sent by clear-to-end-of-line key
key_eos	**ked**	**kS**	sent by clear-to-end-of-screen key

t

key_exit	**kext**	**@9**	sent by exit key
key_f0	**kf0**	**k0**	sent by function key f0
key_f1	**kf1**	**k1**	sent by function key f1
key_f2	**kf2**	**k2**	sent by function key f2
key_f3	**kf3**	**k3**	sent by function key f3
key_f4	**kf4**	**k4**	sent by function key f4
key_f5	**kf5**	**k5**	sent by function key f5
key_f6	**kf6**	**k6**	sent by function key f6
key_f7	**kf7**	**k7**	sent by function key f7
key_f8	**kf8**	**k8**	sent by function key f8
key_f9	**kf9**	**k9**	sent by function key f9
key_f10	**kf10**	**k;**	sent by function key f10
key_f11	**kf11**	**F1**	sent by function key f11
key_f12	**kf12**	**F2**	sent by function key f12
key_f13	**kf13**	**F3**	sent by function key f13
key_f14	**kf14**	**F4**	sent by function key f14
key_f15	**kf15**	**F5**	sent by function key f15
key_f16	**kf16**	**F6**	sent by function key f16
key_f17	**kf17**	**F7**	sent by function key f17
key_f18	**kf18**	**F8**	sent by function key f18
key_f19	**kf19**	**F9**	sent by function key f19
key_f20	**kf20**	**FA**	sent by function key f20
key_f21	**kf21**	**FB**	sent by function key f21
key_f22	**kf22**	**FC**	sent by function key f22
key_f23	**kf23**	**FD**	sent by function key f23
key_f24	**kf24**	**FE**	sent by function key f24
key_f25	**kf25**	**FF**	sent by function key f25
key_f26	**kf26**	**FG**	sent by function key f26
key_f27	**kf27**	**FH**	sent by function key f27
key_f28	**kf28**	**FI**	sent by function key f28
key_f29	**kf29**	**FJ**	sent by function key f29
key_f30	**kf30**	**FK**	sent by function key f30
key_f31	**kf31**	**FL**	sent by function key f31
key_f32	**kf32**	**FM**	sent by function key f32
key_f33	**kf33**	**FN**	sent by function key f33
key_f34	**kf34**	**FO**	sent by function key f34
key_f35	**kf35**	**FP**	sent by function key f35
key_f36	**kf36**	**FQ**	sent by function key f36
key_f37	**kf37**	**FR**	sent by function key f37
key_f38	**kf38**	**FS**	sent by function key f38
key_f39	**kf39**	**FT**	sent by function key f39
key_f40	**kf40**	**FU**	sent by function key f40
key_f41	**kf41**	**FV**	sent by function key f41
key_f42	**kf42**	**FW**	sent by function key f42
key_f43	**kf43**	**FX**	sent by function key f43
key_f44	**kf44**	**FY**	sent by function key f44
key_f45	**kf45**	**FZ**	sent by function key f45
key_f46	**kf46**	**Fa**	sent by function key f46
key_f47	**kf47**	**Fb**	sent by function key f47
key_f48	**kf48**	**Fc**	sent by function key f48
key_f49	**kf49**	**Fd**	sent by function key f49
key_f50	**kf50**	**Fe**	sent by function key f50
key_f51	**kf51**	**Ff**	sent by function key f51
key_f52	**kf52**	**Fg**	sent by function key f52
key_f53	**kf53**	**Fh**	sent by function key f53
key_f54	**kf54**	**Fi**	sent by function key f54
key_f55	**kf55**	**Fj**	sent by function key f55
key_f56	**kf56**	**Fk**	sent by function key f56
key_f57	**kf57**	**Fl**	sent by function key f57
key_f58	**kf58**	**Fm**	sent by function key f58
key_f59	**kf59**	**Fn**	sent by function key f59
key_f60	**kf60**	**Fo**	sent by function key f60

key_f61	**kf61**	**Fp**	sent by function key f61
key_f62	**kf62**	**Fq**	sent by function key f62
key_f63	**kf63**	**Fr**	sent by function key f63
key_find	**kfnd**	**@0**	sent by find key
key_help	**khlp**	**%1**	sent by help key
key_home	**khome**	**kh**	sent by home key
key_ic	**kich1**	**kI**	sent by ins-char/enter ins-mode key
key_il	**kil1**	**kA**	sent by insert-line key
key_left	**kcub1**	**kl**	sent by terminal left-arrow key
key_ll	**kll**	**kH**	sent by home-down key
key_mark	**kmrk**	**%2**	sent by mark key
key_message	**kmsg**	**%3**	sent by message key
key_mouse	**kmous**	**Km**	0631, Mouse event has occurred
key_move	**kmov**	**%4**	sent by move key
key_next	**knxt**	**%5**	sent by next-object key
key_npage	**knp**	**kN**	sent by next-page key
key_open	**kopn**	**%6**	sent by open key
key_options	**kopt**	**%7**	sent by options key
key_ppage	**kpp**	**kP**	sent by previous-page key
key_previous	**kprv**	**%8**	sent by previous-object key
key_print	**kprt**	**%9**	sent by print or copy key
key_redo	**krdo**	**%0**	sent by redo key
key_reference	**kref**	**&1**	sent by ref(erence) key
key_refresh	**krfr**	**&2**	sent by refresh key
key_replace	**krpl**	**&3**	sent by replace key
key_restart	**krst**	**&4**	sent by restart key
key_resume	**kres**	**&5**	sent by resume key
key_right	**kcuf1**	**kr**	sent by terminal right-arrow key
key_save	**ksav**	**&6**	sent by save key
key_sbeg	**kBEG**	**&9**	sent by shifted beginning key
key_scancel	**kCAN**	**&0**	sent by shifted cancel key
key_scommand	**kCMD**	***1**	sent by shifted command key
key_scopy	**kCPY**	***2**	sent by shifted copy key
key_screate	**kCRT**	***3**	sent by shifted create key
key_sdc	**kDC**	***4**	sent by shifted delete-char key
key_sdl	**kDL**	***5**	sent by shifted delete-line key
key_select	**kslt**	***6**	sent by select key
key_send	**kEND**	***7**	sent by shifted end key
key_seol	**kEOL**	***8**	sent by shifted clear-line key
key_sexit	**kEXT**	***9**	sent by shifted exit key
key_sf	**kind**	**kF**	sent by scroll-forward/down key
key_sfind	**kFND**	***0**	sent by shifted find key
key_shelp	**kHLP**	**#1**	sent by shifted help key
key_shome	**kHOM**	**#2**	sent by shifted home key
key_sic	**kIC**	**#3**	sent by shifted input key
key_sleft	**kLFT**	**#4**	sent by shifted left-arrow key
key_smessage	**kMSG**	**%a**	sent by shifted message key
key_smove	**kMOV**	**%b**	sent by shifted move key
key_snext	**kNXT**	**%c**	sent by shifted next key
key_soptions	**kOPT**	**%d**	sent by shifted options key
key_sprevious	**kPRV**	**%e**	sent by shifted prev key
key_sprint	**kPRT**	**%f**	sent by shifted print key
key_sr	**kri**	**kR**	sent by scroll-backward/up key
key_sredo	**kRDO**	**%g**	sent by shifted redo key
key_sreplace	**kRPL**	**%h**	sent by shifted replace key
key_sright	**kRIT**	**%i**	sent by shifted right-arrow key
key_srsume	**kRES**	**%j**	sent by shifted resume key
key_ssave	**kSAV**	**!1**	sent by shifted save key
key_ssuspend	**kSPD**	**!2**	sent by shifted suspend key
key_stab	**khts**	**kT**	sent by set-tab key
key_sundo	**kUND**	**!3**	sent by shifted undo key
key_suspend	**kspd**	**&7**	sent by suspend key

key_undo	**kund**	**&8**	sent by undo key
key_up	**kcuu1**	**ku**	sent by terminal up-arrow key
keypad_local	**rmkx**	**ke**	Out of “keypad-transmit” mode
keypad_xmit	**smkx**	**ks**	Put terminal in “keypad-transmit” mode
lab_f0	**lf0**	**l0**	Labels on function key f0 if not f0
lab_f1	**lf1**	**l1**	Labels on function key f1 if not f1
lab_f2	**lf2**	**l2**	Labels on function key f2 if not f2
lab_f3	**lf3**	**l3**	Labels on function key f3 if not f3
lab_f4	**lf4**	**l4**	Labels on function key f4 if not f4
lab_f5	**lf5**	**l5**	Labels on function key f5 if not f5
lab_f6	**lf6**	**l6**	Labels on function key f6 if not f6
lab_f7	**lf7**	**l7**	Labels on function key f7 if not f7
lab_f8	**lf8**	**l8**	Labels on function key f8 if not f8
lab_f9	**lf9**	**l9**	Labels on function key f9 if not f9
lab_f10	**lf10**	**la**	Labels on function key f10 if not f10
label_format	**fln**	**Lf**	Label format
label_off	**rmln**	**LF**	Turn off soft labels
label_on	**smln**	**LO**	Turn on soft labels
meta_off	**rmm**	**mo**	Turn off "meta mode"
meta_on	**smm**	**mm**	Turn on "meta mode" (8th bit)
micro_column_address	**mhpa**	**ZY**	Like **column_address** for micro adjustment
micro_down	**mcud1**	**ZZ**	Like **cursor_down** for micro adjustment
micro_left	**mcub1**	**Za**	Like **cursor_left** for micro adjustment
micro_right	**mcuf1**	**Zb**	Like **cursor_right** for micro adjustment
micro_row_address	**mvpa**	**Zc**	Like **row_address** for micro adjustment
micro_up	**mcuu1**	**Zd**	Like **cursor_up** for micro adjustment
mouse_info	**minfo**	**Mi**	Mouse status information
newline	**nel**	**nw**	Newline (behaves like **cr** followed by **lf**)
order_of_pins	**porder**	**Ze**	Matches software bits to print-head pins
orig_colors	**oc**	**oc**	Set all color(-pair)s to the original ones
orig_pair	**op**	**op**	Set default color-pair to the original one
pad_char	**pad**	**pc**	Pad character (rather than null)
parm_dch	**dch**	**DC**	Delete #1 chars
parm_delete_line	**dl**	**DL**	Delete #1 lines
parm_down_cursor	**cud**	**DO**	Move down #1 lines.
parm_down_micro	**mcud**	**Zf**	Like **parm_down_cursor** for micro adjust.
parm_ich	**ich**	**IC**	Insert #1 blank chars
parm_index	**indn**	**SF**	Scroll forward #1 lines.
parm_insert_line	**il**	**AL**	Add #1 new blank lines
parm_left_cursor	**cub**	**LE**	Move cursor left #1 spaces
parm_left_micro	**mcub**	**Zg**	Like **parm_left_cursor** for micro adjust.
parm_right_cursor	**cuf**	**RI**	Move right #1 spaces.
parm_right_micro	**mcuf**	**Zh**	Like **parm_right_cursor** for micro adjust.
parm_rindex	**rin**	**SR**	Scroll backward #1 lines.
parm_up_cursor	**cuu**	**UP**	Move cursor up #1 lines.
parm_up_micro	**mcuu**	**Zi**	Like **parm_up_cursor** for micro adjust.
pc_term_options	**pctrm**	**S6**	PC terminal options
pkey_key	**pfkey**	**pk**	Prog funct key #1 to type string #2
pkey_local	**pfloc**	**pl**	Prog funct key #1 to execute string #2
pkey_plab	**pfxl**	**xl**	Prog key #1 to xmit string #2 and show string #3
pkey_xmit	**pfx**	**px**	Prog funct key #1 to xmit string #2
plab_norm	**pln**	**pn**	Prog label #1 to show string #2
print_screen	**mc0**	**ps**	Print contents of the screen
prtr_non	**mc5p**	**pO**	Turn on the printer for #1 bytes
prtr_off	**mc4**	**pf**	Turn off the printer
prtr_on	**mc5**	**po**	Turn on the printer
pulse	**pulse**	**PU**	Select pulse dialing
quick_dial	**qdial**	**QD**	Dial phone number #1, without progress detection
remove_clock	**rmclk**	**RC**	Remove time-of-day clock
repeat_char	**rep**	**rp**	Repeat char #1 #2 times
req_for_input	**rfi**	**RF**	Send next input char (for ptys)
req_mouse_pos	**reqmp**	**RQ**	Request mouse position report

t

reset_1string	**rs1**	**r1**	Reset terminal completely to sane modes
reset_2string	**rs2**	**r2**	Reset terminal completely to sane modes
reset_3string	**rs3**	**r3**	Reset terminal completely to sane modes
reset_file	**rf**	**rf**	Name of file containing reset string
restore_cursor	**rc**	**rc**	Restore cursor to position of last sc
row_address	**vpa**	**cv**	Vertical position absolute
save_cursor	**sc**	**sc**	Save cursor position
scancode_escape	**scesc**	**S7**	Escape for scancode emulation
scroll_forward	**ind**	**sf**	Scroll text up
scroll_reverse	**ri**	**sr**	Scroll text down
select_char_set	**scs**	**Zj**	Select character set
set0_des_seq	**s0ds**	**s0**	Shift into codeset 0 (EUC set 0, ASCII)
set1_des_seq	**s1ds**	**s1**	Shift into codeset 1
set2_des_seq	**s2ds**	**s2**	Shift into codeset 2
set3_des_seq	**s3ds**	**s3**	Shift into codeset 3
set_a_background	**setab**	**AB**	Set background color using ANSI escape
set_a_foreground	**setaf**	**AF**	Set foreground color using ANSI escape
set_attributes	**sgr**	**sa**	Define the video attributes #1-#9
set_background	**setb**	**Sb**	Set current background color
set_bottom_margin	**smgb**	**Zk**	Set bottom margin at current line
set_bottom_margin_parm	**smgbp**	**Zl**	Set bottom margin at #1 or #2 lines from bottom
set_clock	**sclk**	**SC**	Set time-of-day clock
set_color_band	**setcolor**	**Yz**	Change to ribbon color #1
set_color_pair	**scp**	**sp**	Set current color-pair
set_foreground	**setf**	**Sf**	Set current foreground color1
set_left_margin	**smgl**	**ML**	Set left margin at current line
set_left_margin_parm	**smglp**	**Zm**	Set left (right) margin at column #1 (#2)
set_lr_margin	**smglr**	**ML**	Sets both left and right margins
set_page_length	**slines**	**YZ**	Set page length to #1 lines (use tparm)
set_right_margin	**smgr**	**MR**	Set right margin at current column
set_right_margin_parm	**smgrp**	**Zn**	Set right margin at column #1
set_tab	**hts**	**st**	Set a tab in all rows, current column
set_tb_margin	**smgtb**	**MT**	Sets both top and bottom margins
set_top_margin	**smgt**	**Zo**	Set top margin at current line
set_top_margin_parm	**smgtp**	**Zp**	Set top (bottom) margin at line #1 (#2)
set_window	**wind**	**wi**	Current window is lines #1-#2 cols #3-#4
start_bit_image	**sbim**	**Zq**	Start printing bit image graphics
start_char_set_def	**scsd**	**Zr**	Start definition of a character set
stop_bit_image	**rbim**	**Zs**	End printing bit image graphics
stop_char_set_def	**rcsd**	**Zt**	End definition of a character set
subscript_characters	**subcs**	**Zu**	List of “subscript-able” characters
superscript_characters	**supcs**	**Zv**	List of “superscript-able” characters
tab	**ht**	**ta**	Tab to next 8-space hardware tab stop
these_cause_cr	**docr**	**Zw**	Printing any of these chars causes **cr**
to_status_line	**tsl**	**ts**	Go to status line, col #1
tone	**tone**	**TO**	Select touch tone dialing
user0	**u0**	**U0**	User string 0
user1	**u1**	**U1**	User string 1
user2	**u2**	**U2**	User string 2
user3	**U3**	**u3**	User string 3
user4	**u4**	**u4**	User string 4
user5	**u5**	**u5**	User string 5
user6	**u6**	**u6**	User string 6
user7	**u7**	**u7**	User string 7
user8	**u8**	**u8**	User string 8
user9	**u9**	**u9**	User string 9
underline_char	**uc**	**uc**	Underscore one char and move past it
up_half_line	**hu**	**hu**	Half-line up (reverse 1/2 linefeed)
wait_tone	**wait**	**WA**	Wait for dial tone
xoff_character	**xoffc**	**XF**	X-off character
xon_character	**xonc**	**XN**	X-on character
zero_motion	**zerom**	**Zx**	No motion for the subsequent character

The following are declared as functions and may be defined as macros:

```
int    tgetent(char *bp, char *name);
int    tgetflag(char id[2]);
int    tgetnum(char id[2]);
char  *tgetstr(char id[2], char **area);
char  *tgoto(char *cap, int col, int row);
int    tputs(char *str, int affcnt, int (*putc)(void));
```

SEE ALSO

curs_termcap(3X), curs_termin(3X), printf(1).

t

NAME

terminfo - printer, terminal, and modem capability database

SYNOPSIS

/usr/lib/terminfo/?/*

List of Section Headings in DESCRIPTION

DESCRIPTION

The requirements in this manpage are in effect only for implementations that claim Enhanced Curses compliance.

Terminfo Source Format

The **terminfo** database contains a description of the capabilities of a variety of devices, such as terminals and printers. Devices are described by specifying a set of capabilities, by quantifying certain aspects of the device, and by specifying character sequences that effect particular results.

This manpage specifies the format of **terminfo** source files.

X/Open-compliant implementations must provide a facility that accepts source files in the format specified in this manpage as a means of entering information into the **terminfo** database. The facility for installing this information into the database is implementation-specific. A valid **terminfo** entry describing a given model of terminal can be added to **terminfo** on any X/Open-compliant implementation to permit use of the same terminal model.

The "Source File Syntax" section describes the syntax of **terminfo** source files. A grammar and lexical conventions appear in the "Formal Grammar" section below. A list of all terminal capabilities defined by X/Open appears in the "Defined Capabilities" section below. An example follows in the "Sample Entry" section below. The "Device Capabilities" section describes the specification of devices in general, such as video terminals. The "Printer Capabilities" section describes the specification of printers.

The **terminfo** database is often used by screen-oriented applications such as **vi** and Curses programs, as well as by some utilities such as **ls** and **more**. This usage allows them to work with a variety of devices without changes to the programs.

Source File Syntax

Source files can use the ISO 8859-1 codeset. The behavior when the source file is in another codeset is unspecified. Traditional practice has been to translate information from other codesets into the source file syntax.

terminfo source files consist of one or more device descriptions. Each description defines a mnemonic name for the terminal model. Each description consists of a header (beginning in column 1) and one or more lines that list the features for that particular device. Every line in a **terminfo** source file must end in a comma. Every line in a **terminfo** source file except the header must be indented with one or more white spaces (either spaces or tabs).

Entries in **terminfo** source files consist of a number of comma-separated fields. White space after each comma is ignored. Embedded commas must be escaped by using a backslash. The following example shows the format of a **terminfo** source file:

alias1 **|** *alias2* **|** ... **|** *aliasn* **|** *longname*,
whitespace **am, lines #24,**
whitespace **home=\Eeh,**

The first line, commonly referred to as the header line, must begin in column one and must contain at least two aliases separated by vertical bars. The last field in the header line must be the long name of the device and it may contain any string.

Alias names must be unique in the **terminfo** database and they must conform to file naming conventions established by implementation-specific **terminfo** compilation utilities. Implementations will recognize alias names consisting only of characters from the portable file name character set except that implementations need not accept a first character of minus (**-**). For example, a typical restriction is that they cannot contain white space or slashes. There may be further constraints imposed on source file values by the implementation-specific **terminfo** compilation utilities.

Each capability in **terminfo** is of one of the following types:

- Boolean capabilities show that a device has or does not have a particular feature.
- Numeric capabilities quantify particular features of a device.
- String capabilities provide sequences that can be used to perform particular operations on devices.

Capability names adhere to an informal length limit of five characters. Whenever possible, capability names are chosen to be the same as or similar to those specified by the ANSI X3.64-1979 standard. Semantics are also intended to match those of the ANSI standard.

All string capabilities may have padding specified, with the exception of those used for input. Input capabilities, listed under the **Strings** section in the following tables, have names beginning with **key_**. These capabilities are defined in **<term.h>**.

Minimum Guaranteed Limits

All X/Open-compliant implementations support at least the following limits for the **terminfo** source file:

Source File Characteristic	Minimum Guaranteed Value
Length of a line	1023 bytes
Length of a terminal alias	14 bytes
Length of a terminal model name	128 bytes
Width of a single field	128 bytes
Length of a string value	1000 bytes
Length of a string representing a numeric value	99 digits
Magnitude of a numeric value	0 up to and including 32767

An implementation may support higher limits than those specified above.

Formal Grammar

The grammar and lexical conventions in this section together describe the syntax for **terminfo** terminal descriptions within a **terminfo** source file. A terminal description that satisfies the requirements of this section will be accepted by all implementations. (The notation "*(n)*" refers to a note following the description.)

```
descriptions : START_OF_HEADER_LINE(1) rest_of_header_line feature_lines
          | descriptions START_OF_HEADER_LINE rest_of_header_line
          | feature_lines
          ;

rest_of_header_line : PIPE LONGNAME COMMA NEWLINE
          | aliases PIPE LONGNAME COMMA NEWLINE
          ;

feature_lines : start_feature_line rest_of_feature_line
          | feature_lines start_feature_line rest_of_feature_line
          ;

start_feature_line : START_FEATURE_LINE_BOOLEAN(2)
          | START_FEATURE_LINE_NUMERIC(3)
          | START_FEATURE_LINE_STRING(4)
          ;
```

t

```
rest_of_feature_line : features COMMA NEWLINE
          | COMMA NEWLINE
          ;
features : COMMA feature
          | features COMMA feature
          ;
aliases : PIPE ALIAS
          | aliases PIPE ALIAS
          ;
feature : BOOLEAN
          | NUMERIC
          | STRING
          ;
```

(1) An ALIAS that begins in column one. This is handled by the lexical analyzer.

(2) A BOOLEAN feature that begins after column one but is the first feature on the feature line. This is handled by the lexical analyzer.

(3) A NUMERIC feature that begins after column one but is the first feature on the feature line. This is handled by the lexical analyzer.

(4) A STRING feature that begins after column one but is the first feature on the feature line. This is handled by the lexical analyzer.

The lexical conventions for **terminfo** descriptions are as follows:

1. White space consists of the <space> and <tab> characters.
2. An ALIAS may contain any graph characters other than comma (**,**), slash (**/**), and bar (**|**). (Graph characters are those characters for which **isgraph()** returns nonzero; see *ctype*(3C).)
3. A LONGNAME may contain any print characters other than comma (**,**) and bar (**|**). (Print characters are those characters for which **isprint()** returns nonzero; see *ctype*(3C).)
4. A BOOLEAN feature may contain any print characters other than comma (**,**), equals (**=**), and pound sign (**#**).
5. A NUMERIC feature consists of:
   a. A name which may contain any print character other than comma (**,**), equals (**=**), and pound sign (**#**).
   b. The pound sign (**#**) character.
   c. A positive integer which conforms to the C language convention for integer constants.
6. A STRING feature consists of:
   a. A name which may contain any print character other than comma (**,**), equals (**=**), and pound sign (**#**).
   b. The equals (**=**) character.
   c. A string which may contain any print characters other than comma (**,**).
7. White space immediately following a comma (**,**) is ignored.
8. Comments consist of the beginning of a line, optional white space, a required pound sign (**#**), and a terminating end of line.
9. A header line must begin in column one.
10. A feature line must not begin in column one.
11. Blank lines are ignored.

Defined Capabilities

X/Open defines the capabilities listed in the following table. All X/Open-compliant implementations must accept each of these capabilities in an entry in a **terminfo** source file. Implementations use this information to determine how properly to operate the current terminal. In addition, implementations return any of the current terminal's capabilities when the application calls the query functions listed in **tgetent()** (in the cases where the following table lists a **Termcap** code) and **tigetflag()** (see *tgetent*(3X) and *tigetflag*(3X)).

The table of capabilities has the following columns:

Variable Names for use by the Curses functions that operate on the **terminfo** database. These names are reserved and the application must not define them.

Capname The short name for a capability specified in the **terminfo** source file. It is used for updating the source file and by the **tput** command (see *tput*(1)).

Termcap Codes provided for compatibility with older applications. These codes are **TO BE WITHDRAWN**. Because of this, not all **Capname**s have **Termcap** codes.

Description A short summary of the capability.

Booleans

Variable	Cap-name	Term-cap	Description
auto_left_margin	**bw**	**bw**	**cub1** wraps from column 0 to last column
auto_right_margin	**am**	**am**	Terminal has automatic margins
back_color_erase	**bce**	**ut**	Screen erased with background color
can_change	**ccc**	**cc**	Terminal can re-define existing color
ceol_standout_glitch	**xhp**	**xs**	Standout not erased by overwriting (hp)
col_addr_glitch	**xhpa**	**YA**	Only positive motion for **hpa**/**mhpa** caps
cpi_changes_res	**cpix**	**YF**	Changing character pitch changes resolution
cr_cancels_micro_mode	**crxm**	**YB**	Using **cr** turns off micro mode
dest_tabs_magic_smso	**xt**	**xt**	Destructive tabs, magic **smso** char (t1061)
eat_newline_glitch	**xenl**	**xn**	Newline ignored after 80 columns (Concept)
erase_overstrike	**eo**	**eo**	Can erase overstrikes with a blank
generic_type	**gn**	**gn**	Generic line type (e.g., dialup, switch)
get_mouse	**getm**	**Gm**	Curses should get button events
hard_copy	**hc**	**hc**	Hardcopy terminal
hard_cursor	**chts**	**HC**	Cursor is hard to see
has_meta_key	**km**	**km**	Has a meta key (shift, sets parity bit)
has_print_wheel	**daisy**	**YC**	Printer needs operator to change character set
has_status_line	**hs**	**hs**	Has extra "status line"
hue_lightness_saturation	**hls**	**hl**	Terminal uses only HLS color notation (Tektronix)
insert_null_glitch	**in**	**in**	Insert mode distinguishes nulls
lpi_changes_res	**lpix**	**YG**	Changing line pitch changes resolution
memory_above	**da**	**da**	Display may be retained above the screen
memory_below	**db**	**db**	Display may be retained below the screen
move_insert_mode	**mir**	**mi**	Safe to move while in insert mode
move_standout_mode	**msgr**	**ms**	Safe to move in standout modes
needs_xon_xoff	**nxon**	**nx**	Padding won't work, XON/XOFF required
no_esc_ctlc	**xsb**	**xb**	Beehive (f1=escape, f2=ctrl C)
no_pad_char	**npc**	**NP**	Pad character doesn't exist
non_dest_scroll_region	**ndscr**	**ND**	Scrolling region is nondestructive
non_rev_rmcup	**nrrmc**	**NR**	**smcup** does not reverse **rmcup**
over_strike	**os**	**os**	Terminal overstrikes on hard-copy terminal
prtr_silent	**mc5i**	**5i**	Printer won't echo on screen
row_addr_glitch	**xvpa**	**YD**	Only positive motion for **vpa**/**mvpa** caps
semi_auto_right_margin	**sam**	**YE**	Printing in last column causes **cr**
status_line_esc_ok	**eslok**	**es**	Escape can be used on the status line
tilde_glitch	**hz**	**hz**	Hazeltine; can't print tilde (˜)
transparent_underline	**ul**	**ul**	Underline character overstrikes
xon_xoff	**xon**	**xo**	Terminal uses XON/XOFF handshaking

t

Numbers

Variable	Cap-name	Term-cap	Description
bit_image_entwining	**bitwin**	**Yo**	Number of passes for each bit-map row
bit_image_type	**bitype**	**Yp**	Type of bit image device
buffer_capacity	**bufsz**	**Ya**	Number of bytes buffered before printing
buttons	**btns**	**BT**	Number of buttons on the mouse
columns	**cols**	**co**	Number of columns in a line
dot_horz_spacing	**spinh**	**Yc**	Spacing of dots horizontally in dots per inch
dot_vert_spacing	**spinv**	**Yb**	Spacing of pins vertically in pins per inch
init_tabs	**it**	**it**	Tabs initially every # spaces
label_height	**lh**	**lh**	Number of rows in each label
label_width	**lw**	**lw**	Number of columns in each label
lines	**lines**	**li**	Number of lines on a screen or a page
lines_of_memory	**lm**	**lm**	Lines of memory if greater than **lines**; 0 means varies
max_attributes	**ma**	**ma**	Maximum combined video attributes terminal can display
magic_cookie_glitch	**xmc**	**sg**	Number of blank characters left by **smso** or **rmso**
max_colors	**colors**	**Co**	Maximum number of colors on the screen
max_micro_address	**maddr**	**Yd**	Maximum value in **micro_..._address**
max_micro_jump	**mjump**	**Ye**	Maximum value in **parm_..._micro**
max_pairs	**pairs**	**pa**	Maximum number of color-pairs on the screen
maximum_windows	**wnum**	**MW**	Maximum number of definable windows
micro_col_size	**mcs**	**Yf**	Character step size when in micro mode
micro_line_size	**mls**	**Yg**	Line step size when in micro mode
no_color_video	**ncv**	**NC**	Video attributes that can't be used with colors
num_labels	**nlab**	**Nl**	Number of labels on screen (start at 1)
number_of_pins	**npins**	**Yh**	Number of pins in print-head
output_res_char	**orc**	**Yi**	Horizontal resolution in units per character
output_res_line	**orl**	**Yj**	Vertical resolution in units per line
output_res_horz_inch	**orhi**	**Yk**	Horizontal resolution in units per inch
output_res_vert_inch	**orvi**	**Yl**	Vertical resolution in units per inch
padding_baud_rate	**pb**	**pb**	Lowest baud rate where padding needed
print_rate	**cps**	**Ym**	Print rate in characters per second
virtual_terminal	**vt**	**vt**	Virtual terminal number
wide_char_size	**widcs**	**Yn**	Character step size when in double-wide mode
width_status_line	**wsl**	**ws**	Number of columns in status line

t

Strings

Variable	Cap-name	Term-cap	Description
acs_chars	acsc	ac	Graphic charset pairs aAbBcC
alt_scancode_esc	scesa	S8	Alternate escape for scancode emulation (default is for VT100)
back_tab	cbt	bt	Back tab
bell	bel	bl	Audible signal (bell)
bit_image_carriage_return	bicr	Yv	Move to beginning of same row
bit_image_newline	binel	Zz	Move to next row of the bit image
bit_image_repeat	birep	Xy	Repeat bit-image cell #1 #2 times
carriage_return	cr	cr	Carriage return
change_char_pitch	cpi	ZA	Change number of characters per inch
change_line_pitch	lpi	ZB	Change number of lines per inch
change_res_horz	chr	ZC	Change horizontal resolution
change_res_vert	cvr	ZD	Change vertical resolution
change_scroll_region	csr	cs	Change to lines #1 through #2 (VT100)
char_padding	rmp	rP	Like **ip** but when in replace mode
char_set_names	csnm	Zy	Returns a list of character set names
clear_all_tabs	tbc	ct	Clear all tab stops
clear_margins	mgc	MC	Clear all margins (top, bottom, and sides)
clear_screen	clear	cl	Clear screen and home cursor
clr_bol	el1	cb	Clear to beginning of line, inclusive
clr_eol	el	ce	Clear to end of line
clr_eos	ed	cd	Clear to end of display
code_set_init	csin	ci	Init sequence for multiple codesets
color_names	colornm	Yw	Give name for color #1
column_address	hpa	ch	Set horizontal position to absolute #1
command_character	cmdch	CC	Terminal settable cmd character in prototype
create_window	cwin	CW	Define win #1 to go from #2,#3 to #4,#5
cursor_address	cup	cm	Move to row #1 col #2
cursor_down	cud1	do	Down one line
cursor_home	home	ho	Home cursor (if no **cup**)
cursor_invisible	civis	vi	Make cursor invisible
cursor_left	cub1	le	Move left one space.
cursor_mem_address	mrcup	CM	Memory relative cursor addressing
cursor_normal	cnorm	ve	Make cursor appear normal (undo **vs/vi**)
cursor_right	cuf1	nd	Non-destructive space (cursor or carriage right)
cursor_to_ll	ll	ll	Last line, first column (if no **cup**)
cursor_up	cuu1	up	Upline (cursor up)
cursor_visible	cvvis	vs	Make cursor very visible
define_bit_image_region	defbi	Yx	Define rectangular bit-image region
define_char	defc	ZE	Define a character in a character set
delete_character	dch1	dc	Delete character
delete_line	dl1	dl	Delete line
device_type	devt	dv	Indicate language/codeset support
dial_phone	dial	DI	Dial phone number #1
dis_status_line	dsl	ds	Disable status line
display_clock	dclk	DK	Display time-of-day clock
display_pc_char	dispc	S1	Display PC character
down_half_line	hd	hd	Half-line down (forward 1/2 linefeed)
ena_acs	enacs	eA	Enable alternate character set
end_bit_image_region	endbi	Yy	End a bit-image region
enter_alt_charset_mode	smacs	as	Start alternate character set
enter_am_mode	smam	SA	Turn on automatic margins
enter_blink_mode	blink	mb	Turn on blinking
enter_bold_mode	bold	md	Turn on bold (extra bright) mode
enter_ca_mode	smcup	ti	String to begin programs that use **cup**

enter_delete_mode	**smdc**	**dm**	Delete mode (enter)
enter_dim_mode	**dim**	**mh**	Turn on half-bright mode
enter_doublewide_mode	**swidm**	**ZF**	Enable double wide printing
enter_draft_quality	**sdrfq**	**ZG**	Set draft quality print
enter_horizontal_hl_mode	**ehhlm**		Turn on horizontal highlight mode
enter_insert_mode	**smir**	**im**	Insert mode (enter)
enter_italics_mode	**sitm**	**ZH**	Enable italics
enter_left_hl_mode	**elhlm**		Turn on left highlight mode
enter_leftward_mode	**slm**	**ZI**	Enable leftward carriage motion
enter_low_hl_mode	**elohlm**		Turn on low highlight mode
enter_micro_mode	**smicm**	**ZJ**	Enable micro motion capabilities
enter_near_letter_quality	**snlq**	**ZK**	Set near-letter quality print
enter_normal_quality	**snrmq**	**ZL**	Set normal quality print
enter_pc_charset_mode	**smpch**	**S2**	Enter PC character display mode
enter_protected_mode	**prot**	**mp**	Turn on protected mode
enter_reverse_mode	**rev**	**mr**	Turn on reverse video mode
enter_right_hl_mode	**erhlm**		Turn on right highlight mode
enter_scancode_mode	**smsc**	**S4**	Enter PC scancode mode
enter_secure_mode	**invis**	**mk**	Turn on blank mode (characters invisible)
enter_shadow_mode	**sshm**	**ZM**	Enable shadow printing
enter_standout_mode	**smso**	**so**	Begin standout mode
enter_subscript_mode	**ssubm**	**ZN**	Enable subscript printing
enter_superscript_mode	**ssupm**	**ZO**	Enable superscript printing
enter_top_hl_mode	**ethlm**		Turn on top highlight mode
enter_underline_mode	**smul**	**us**	Start underscore mode
enter_upward_mode	**sum**	**ZP**	Enable upward carriage motion
enter_vertical_hl_mode	**evhlm**		Turn on vertical highlight mode
enter_xon_mode	**smxon**	**SX**	Turn on XON/XOFF handshaking
erase_chars	**ech**	**ec**	Erase #1 characters
exit_alt_charset_mode	**rmacs**	**ae**	End alternate character set
exit_am_mode	**rmam**	**RA**	Turn off automatic margins
exit_attribute_mode	**sgr0**	**me**	Turn off all attributes
exit_ca_mode	**rmcup**	**te**	String to end programs that use **cup**
exit_delete_mode	**rmdc**	**ed**	End delete mode
exit_doublewide_mode	**rwidm**	**ZQ**	Disable double wide printing
exit_insert_mode	**rmir**	**ei**	End insert mode
exit_italics_mode	**ritm**	**ZR**	Disable italics
exit_leftward_mode	**rlm**	**ZS**	Enable rightward (normal) carriage motion
exit_micro_mode	**rmicm**	**ZT**	Disable micro motion capabilities
exit_pc_charset_mode	**rmpch**	**S3**	Disable PC character display mode
exit_scancode_mode	**rmsc**	**S5**	Disable PC scancode mode
exit_shadow_mode	**rshm**	**ZU**	Disable shadow printing
exit_standout_mode	**rmso**	**se**	End standout mode
exit_subscript_mode	**rsubm**	**ZV**	Disable subscript printing
exit_superscript_mode	**rsupm**	**ZW**	Disable superscript printing
exit_underline_mode	**rmul**	**ue**	End underscore mode
exit_upward_mode	**rum**	**ZX**	Enable downward (normal) carriage motion
exit_xon_mode	**rmxon**	**RX**	Turn off XON/XOFF handshaking
fixed_pause	**pause**	**PA**	Pause for 2–3 seconds
flash_hook	**hook**	**fh**	Flash the switch hook
flash_screen	**flash**	**vb**	Visible bell (may move cursor)
form_feed	**ff**	**ff**	Hardcopy terminal page eject
from_status_line	**fsl**	**fs**	Return from status line
goto_window	**wingo**	**WG**	Go to window #1
hangup	**hup**	**HU**	Hang-up phone
init_1string	**is1**	**i1**	Terminal or printer initialization string
init_2string	**is2**	**is**	Terminal or printer initialization string
init_3string	**is3**	**i3**	Terminal or printer initialization string
init_file	**if**	**if**	Name of initialization file
init_prog	**iprog**	**iP**	Path name of program for initialization

t

initialize_color	**initc**	**IC**	Set color #1 to RGB #2, #3, #4
initialize_pair	**initp**	**Ip**	Set color-pair #1 to fg #2, bg #3
insert_character	**ich1**	**ic**	Insert character
insert_line	**il1**	**al**	Add new blank line
insert_padding	**ip**	**ip**	Insert pad after character inserted

The "**key_**" strings are sent by specific keys. The "**key_**" descriptions include the macro, defined in **<curses.h>**, for the code returned by **getch()** when the key is pressed (see *getch*(3X)).

Variable	Cap-name	Term-cap	Description
key_a1	**ka1**	**K1**	Upper left of keypad
key_a3	**ka3**	**K3**	Upper right of keypad
key_b2	**kb2**	**K2**	Center of keypad
key_backspace	**kbs**	**kb**	Sent by backspace key
key_beg	**kbeg**	**@1**	Sent by beg(inning) key
key_btab	**kcbt**	**kB**	Sent by back-tab key
key_c1	**kc1**	**K4**	Lower left of keypad
key_c3	**kc3**	**K5**	Lower right of keypad
key_cancel	**kcan**	**@2**	Sent by cancel key
key_catab	**ktbc**	**ka**	Sent by clear-all-tabs key
key_clear	**kclr**	**kC**	Sent by clear-screen or erase key
key_close	**kclo**	**@3**	Sent by close key
key_command	**kcmd**	**@4**	Sent by cmd (command) key
key_copy	**kcpy**	**@5**	Sent by copy key
key_create	**kcrt**	**@6**	Sent by create key
key_ctab	**kctab**	**kt**	Sent by clear-tab key
key_dc	**kdch1**	**kD**	Sent by delete-character key
key_dl	**kdl1**	**kL**	Sent by delete-line key
key_down	**kcud1**	**kd**	Sent by terminal down-arrow key
key_eic	**krmir**	**kM**	Sent by **rmir** or **smir** in insert mode
key_end	**kend**	**@7**	Sent by end key
key_enter	**kent**	**@8**	Sent by enter/send key
key_eol	**kel**	**kE**	Sent by clear-to-end-of-line key
key_eos	**ked**	**kS**	Sent by clear-to-end-of-screen key
key_exit	**kext**	**@9**	Sent by exit key
key_f0	**kf0**	**k0**	Sent by function key f0
key_f1	**kf1**	**k1**	Sent by function key f1
.	.	.	.
.	.	.	. Similarly for f2 through f61
.	.	.	.
key_f62	**kf62**	**Fq**	Sent by function key f62
key_f63	**kf63**	**Fr**	Sent by function key f63
key_find	**kfnd**	**@0**	Sent by find key
key_help	**khlp**	**%1**	Sent by help key
key_home	**khome**	**kh**	Sent by home key
key_ic	**kich1**	**kI**	Sent by ins-char/enter ins-mode key
key_il	**kil1**	**kA**	Sent by insert-line key
key_left	**kcub1**	**kl**	Sent by terminal left-arrow key
key_ll	**kll**	**kH**	Sent by home-down key
key_mark	**kmrk**	**%2**	Sent by mark key
key_message	**kmsg**	**%3**	Sent by message key
key_mouse	**kmous**	**Km**	0631, mouse event has occurred
key_move	**kmov**	**%4**	Sent by move key
key_next	**knxt**	**%5**	Sent by next-object key
key_npage	**knp**	**kN**	Sent by next-page key
key_open	**kopn**	**%6**	Sent by open key
key_options	**kopt**	**%7**	Sent by options key
key_ppage	**kpp**	**kP**	Sent by previous-page key
key_previous	**kprv**	**%8**	Sent by previous-object key
key_print	**kprt**	**%9**	Sent by print or copy key
key_redo	**krdo**	**%0**	Sent by redo key
key_reference	**kref**	**&1**	Sent by ref(erence) key

key_refresh	krfr	&2	Sent by refresh key
key_replace	krpl	&3	Sent by replace key
key_restart	krst	&4	Sent by restart key
key_resume	kres	&5	Sent by resume key
key_right	kcuf1	kr	Sent by terminal right-arrow key
key_save	ksav	&6	Sent by save key
key_sbeg	kBEG	&9	Sent by shifted beginning key
key_scancel	kCAN	&0	Sent by shifted cancel key
key_scommand	kCMD	*1	Sent by shifted command key
key_scopy	kCPY	*2	Sent by shifted copy key
key_screate	kCRT	*3	Sent by shifted create key
key_sdc	kDC	*4	Sent by shifted delete-char key
key_sdl	kDL	*5	Sent by shifted delete-line key
key_select	kslt	*6	Sent by select key
key_send	kEND	*7	Sent by shifted end key
key_seol	kEOL	*8	Sent by shifted clear-line key
key_sexit	kEXT	*9	Sent by shifted exit key
key_sf	kind	kF	Sent by scroll-forward/down key
key_sfind	kFND	*0	Sent by shifted find key
key_shelp	kHLP	#1	Sent by shifted help key
key_shome	kHOM	#2	Sent by shifted home key
key_sic	kIC	#3	Sent by shifted input key
key_sleft	kLFT	#4	Sent by shifted left-arrow key
key_smessage	kMSG	%a	Sent by shifted message key
key_smove	kMOV	%b	Sent by shifted move key
key_snext	kNXT	%c	Sent by shifted next key
key_soptions	kOPT	%d	Sent by shifted options key
key_sprevious	kPRV	%e	Sent by shifted prev key
key_sprint	kPRT	%f	Sent by shifted print key
key_sr	kri	kR	Sent by scroll-backward/up key
key_sredo	kRDO	%g	Sent by shifted redo key
key_sreplace	kRPL	%h	Sent by shifted replace key
key_sright	kRIT	%i	Sent by shifted right-arrow key
key_srsume	kRES	%j	Sent by shifted resume key
key_ssave	kSAV	!1	Sent by shifted save key
key_ssuspend	kSPD	!2	Sent by shifted suspend key
key_stab	khts	kT	Sent by set-tab key
key_sundo	kUND	!3	Sent by shifted undo key
key_suspend	kspd	&7	Sent by suspend key
key_undo	kund	&8	Sent by undo key
key_up	kcuu1	ku	Sent by terminal up-arrow key
keypad_local	rmkx	ke	Out of "keypad-transmit" mode
keypad_xmit	smkx	ks	Put terminal in "keypad-transmit" mode
lab_f0	lf0	l0	Labels on function key f0 if not f0
lab_f1	lf1	l1	Labels on function key f1 if not f1
lab_f2	lf2	l2	Labels on function key f2 if not f2
lab_f3	lf3	l3	Labels on function key f3 if not f3
lab_f4	lf4	l4	Labels on function key f4 if not f4
lab_f5	lf5	l5	Labels on function key f5 if not f5
lab_f6	lf6	l6	Labels on function key f6 if not f6
lab_f7	lf7	l7	Labels on function key f7 if not f7
lab_f8	lf8	l8	Labels on function key f8 if not f8
lab_f9	lf9	l9	Labels on function key f9 if not f9
lab_f10	lf10	la	Labels on function key f10 if not f10
label_format	fln	Lf	Label format
label_off	rmln	LF	Turn off soft labels
label_on	smln	LO	Turn on soft labels
memory_lock	meml	ml	Lock memory above cursor
memory_unlock	memu	mu	Turn memory lock off
meta_off	rmm	mo	Turn off "meta mode"
meta_on	smm	mm	Turn on "meta mode" (8th bit)
micro_column_address	mhpa	ZY	Like **column_address** for micro adjustment

t

micro_down	**mcud1**	**ZZ**	Like **cursor_down** for micro adjustment
micro_left	**mcub1**	**Za**	Like **cursor_left** for micro adjustment
micro_right	**mcuf1**	**Zb**	Like **cursor_right** for micro adjustment
micro_row_address	**mvpa**	**Zc**	Like **row_address** for micro adjustment
micro_up	**mcuu1**	**Zd**	Like **cursor_up** for micro adjustment
mouse_info	**minfo**	**Mi**	Mouse status information
newline	**nel**	**nw**	Newline (behaves like **cr** followed by **lf**)
order_of_pins	**porder**	**Ze**	Matches software bits to print-head pins
orig_colors	**oc**	**oc**	Set all color(-pair)s to the original ones
orig_pair	**op**	**op**	Set default color-pair to the original one
pad_char	**pad**	**pc**	Pad character (rather than null)
parm_dch	**dch**	**DC**	Delete #1 chars
parm_delete_line	**dl**	**DL**	Delete #1 lines
parm_down_cursor	**cud**	**DO**	Move down #1 lines.
parm_down_micro	**mcud**	**Zf**	Like **parm_down_cursor** for micro adjust.
parm_ich	**ich**	**IC**	Insert #1 blank chars
parm_index	**indn**	**SF**	Scroll forward #1 lines.
parm_insert_line	**il**	**AL**	Add #1 new blank lines
parm_left_cursor	**cub**	**LE**	Move cursor left #1 spaces
parm_left_micro	**mcub**	**Zg**	Like **parm_left_cursor** for micro adjust.
parm_right_cursor	**cuf**	**RI**	Move right #1 spaces.
parm_right_micro	**mcuf**	**Zh**	Like **parm_right_cursor** for micro adjust.
parm_rindex	**rin**	**SR**	Scroll backward #1 lines.
parm_up_cursor	**cuu**	**UP**	Move cursor up #1 lines.
parm_up_micro	**mcuu**	**Zi**	Like **parm_up_cursor** for micro adjust.
pc_term_options	**pctrm**	**S6**	PC terminal options
pkey_key	**pfkey**	**pk**	Prog funct key #1 to type string #2
pkey_local	**pfloc**	**pl**	Prog funct key #1 to execute string #2
pkey_plab	**pfxl**	**xl**	Prog key #1 to xmit string #2 and show string #3
pkey_xmit	**pfx**	**px**	Prog funct key #1 to xmit string #2
plab_norm	**pln**	**pn**	Prog label #1 to show string #2
print_screen	**mc0**	**ps**	Print contents of the screen
prtr_non	**mc5p**	**pO**	Turn on the printer for #1 bytes
prtr_off	**mc4**	**pf**	Turn off the printer
prtr_on	**mc5**	**po**	Turn on the printer
pulse	**pulse**	**PU**	Select pulse dialing
quick_dial	**qdial**	**QD**	Dial phone number #1, without progress detection
remove_clock	**rmclk**	**RC**	Remove time-of-day clock
repeat_char	**rep**	**rp**	Repeat char #1 #2 times
req_for_input	**rfi**	**RF**	Send next input char (for ptys)
req_mouse_pos	**reqmp**	**RQ**	Request mouse position report
reset_1string	**rs1**	**r1**	Reset terminal completely to sane modes
reset_2string	**rs2**	**r2**	Reset terminal completely to sane modes
reset_3string	**rs3**	**r3**	Reset terminal completely to sane modes
reset_file	**rf**	**rf**	Name of file containing reset string
restore_cursor	**rc**	**rc**	Restore cursor to position of last sc
row_address	**vpa**	**cv**	Set vertical position to absolute #1
save_cursor	**sc**	**sc**	Save cursor position
scancode_escape	**scesc**	**S7**	Escape for scancode emulation
scroll_forward	**ind**	**sf**	Scroll text up
scroll_reverse	**ri**	**sr**	Scroll text down
select_char_set	**scs**	**Zj**	Select character set
set0_des_seq	**s0ds**	**s0**	Shift into codeset 0 (EUC set 0, ASCII)
set1_des_seq	**s1ds**	**s1**	Shift into codeset 1
set2_des_seq	**s2ds**	**s2**	Shift into codeset 2
set3_des_seq	**s3ds**	**s3**	Shift into codeset 3
set_a_attributes	**sgr1**		Define second set of video attributes #1–#6
set_a_background	**setab**	**AB**	Set background color to #1 using ANSI escape
set_a_foreground	**setaf**	**AF**	Set foreground color to #1 using ANSI escape
set_attributes	**sgr**	**sa**	Define first set of video attributes #1–#9
set_background	**setb**	**Sb**	Set background color to #1

set_bottom_margin	**smgb**	**Zk**	Set bottom margin at current line
set_bottom_margin_parm	**smgbp**	**Zl**	Set bottom margin at line #1 or #2 lines from bottom
set_clock	**sclk**	**SC**	Set clock to hours (#1), minutes (#2), seconds (#3)
set_color_band	**setcolor**	**Yz**	Change to ribbon color #1
set_color_pair	**scp**	**sp**	Set current color pair to #1
set_foreground	**setf**	**Sf**	Set foreground color to #1
set_left_margin	**smgl**	**ML**	Set left margin at current column
set_left_margin_parm	**smglp**	**Zm**	Set left (right) margin at column #1 (#2)
set_lr_margin	**smglr**	**ML**	Sets both left and right margins
set_page_length	**slines**	**YZ**	Set page length to #1 lines
set_pglen_inch	**slength**	**YI**	Set page length to #1 hundredth of an inch
set_right_margin	**smgr**	**MR**	Set right margin at current column
set_right_margin_parm	**smgrp**	**Zn**	Set right margin at column #1
set_tab	**hts**	**st**	Set a tab in all rows, current column
set_tb_margin	**smgtb**	**MT**	Sets both top and bottom margins
set_top_margin	**smgt**	**Zo**	Set top margin at current line
set_top_margin_parm	**smgtp**	**Zp**	Set top (bottom) margin at line #1 (#2)
set_window	**wind**	**wi**	Current window is lines #1–#2 cols #3–#4
start_bit_image	**sbim**	**Zq**	Start printing bit image graphics
start_char_set_def	**scsd**	**Zr**	Start definition of a character set
stop_bit_image	**rbim**	**Zs**	End printing bit image graphics
stop_char_set_def	**rcsd**	**Zt**	End definition of a character set
subscript_characters	**subcs**	**Zu**	List of "subscript-able" characters
superscript_characters	**supcs**	**Zv**	List of "superscript-able" characters
tab	**ht**	**ta**	Tab to next 8-space hardware tab stop
these_cause_cr	**docr**	**Zw**	Printing any of these chars causes **cr**
to_status_line	**tsl**	**ts**	Go to status line, col #1
tone	**tone**	**TO**	Select touch tone dialing
user0	**u0**	**u0**	User string 0
user1	**u1**	**u1**	User string 1
user2	**u2**	**u2**	User string 2
user3	**u3**	**u3**	User string 3
user4	**u4**	**u4**	User string 4
user5	**u5**	**u5**	User string 5
user6	**u6**	**u6**	User string 6
user7	**u7**	**u7**	User string 7
user8	**u8**	**u8**	User string 8
user9	**u9**	**u9**	User string 9
underline_char	**uc**	**uc**	Underscore one char and move past it
up_half_line	**hu**	**hu**	Half-line up (reverse 1/2 linefeed)
wait_tone	**wait**	**WA**	Wait for dial tone
xoff_character	**xoffc**	**XF**	XOFF character
xon_character	**xonc**	**XN**	XON character
zero_motion	**zerom**	**Zx**	No motion for the subsequent character

t

Sample Entry

The following entry describes the AT&T 610 terminal. (The **pfxl** and **sgr** values have been split for printing; they would actually be entered as single lines.)

```
610|610bct|ATT610|att610|AT&T610;80column;98key keyboard,
        am, eslok, hs, mir, msgr, xenl, xon,
        cols#80, it#8, lh#2, lines#24, lw#8, nlab#8, wsl#80,
        acsc=‘‘aaffggjjkkllmmnnooppqqrrssttuuvvwwxxyyzz{{||}}~~,
        bel=^G, blink=\E[5m, bold=\E[1m, cbt=\E[Z,
        civis=\E[?25l, clear=\E[H\E[J, cnorm=\E[?25h\E[?12l,
        cr=\r, csr=\E[%i%p1%d;%p2%dr, cub=\E[%p1%dD, cub1=\b,
        cud=\E[%p1%dB, cud1=\E[B, cuf=\E[%p1%dC, cuf1=\E[C,
        cup=\E[%i%p1%d;%p2%dH, cuu=\E[%p1%dA, cuu1=\E[A,
        cvvis=\E[?12;25h, dch=\E[%p1%dP, dch1=\E[P, dim=\E[2m,
        dl=\E[%p1%dM, dl1=\E[M, ed=\E[J, el=\E[K, el1=\E[1K,
        flash=\E[?5h$<200>\E[?5l, fsl=\E8, home=\E[H, ht=\t,
```

```
        ich=\E[%p1%d@, il=\E[%p1%dL, il1=\E[L, ind=\ED, .ind=\ED$<9>,
        invis=\E[8m,
        is1=\E[8;0 | \E[?3;4;5;13;15l\E[13;20l\E[?7h\E[12h\E(B\E)0,
        is2=\E[0m^O, is3=\E(B\E)0, kLFT=\E[\s@, kRIT=\E[\sA,
        kbs=^H, kcbt=\E[Z, kclr=\E[2J, kcub1=\E[D, kcud1=\E[B,
        kcuf1=\E[C, kcuu1=\E[A, kfP=\EOc, kfP0=\ENp,
        kfP1=\ENq, kfP2=\ENr, kfP3=\ENs, kfP4=\ENt, kfI=\EOd,
        kfB=\EOe, kf4=\EOf, kf(CW=\EOg, kf6=\EOh, kf7=\EOi,
        kf8=\EOj, kf9=\ENo, khome=\E[H, kind=\E[S, kri=\E[T,
        ll=\E[24H, mc4=\E[?4i, mc5=\E[?5i, nel=\EE,
        pfxl=\E[%p1%d;%p2%l%02dq%?%p1%{9}%<%t\s\s\sF%p1%1d
\s\s\s\s\s\s\s\s\s\s\s%;%p2%s,
        pln=\E[%p1%d;0;0;0q%p2%:-16.16s, rc=\E8, rev=\E[7m,
        ri=\EM, rmacs=^O, rmir=\E[4l, rmln=\E[2p, rmso=\E[m,
        rmul=\E[m, rs2=\Ec\E[?3l, sc=\E7,
        sgr=\E[0%?%p6%t;1%;%?%p5%t;2%;%?%p2%t;4%;%?%p4%t;5%;%?%p3%p1%
|%t;7%;%?%p7%t;8%;m%?%p9%t^N%e^O%;,
        sgr0=\E[m^O, smacs=^N, smir=\E[4h, smln=\E[p,
        smso=\E[7m, smul=\E[4m, tsl=\E7\E[25;%i%p1%dx,
```

Types of Capabilities in the Sample Entry

The sample entry shows the formats for the three types of **terminfo** capabilities: boolean, numeric, and string. All capabilities specified in the **terminfo** source file must be followed by commas, including the last capability in the source file. In **terminfo** source files, capabilities are referenced by their capability names (as shown in the **Capname** column of the previous tables).

Boolean Capabilities

A boolean capability is true if its **Capname** is present in the entry, and false if its **Capname** is not present in the entry.

The "**@**" character following a **Capname** is used to explicitly declare that a boolean capability is false, in situations described in the "Similar Terminals" subsection of the "Insert/Delete Line" section below.

Numeric Capabilities

Numeric capabilities are followed by the character "**#**" and then a positive integer value. The example assigns the value 80 to the **cols** numeric capability by coding:

```
cols#80
```

Values for numeric capabilities may be specified in decimal, octal or hexadecimal, using normal C-language conventions.

String Capabilities

String-valued capabilities such as **el** (clear to end of line sequence) are listed by the **Capname**, an "**=**", and a string ended by the next occurrence of a comma.

A delay in milliseconds may appear anywhere in such a capability, preceded by "**$**" and enclosed in angle brackets, as in **el=\EK$<3>**. The Curses implementation achieves delays by outputting to the terminal an appropriate number of system-defined padding characters. The **tputs()** function provides delays when used to send such a capability to the terminal.

The delay can be any of the following: a number; a number followed by an asterisk, such as **5***; a number followed by a slash, such as **5/**; or a number followed by both, such as **5*/**.

- ***** Shows that the required delay is proportional to the number of lines affected by the operation, and the amount given is the delay required per affected unit. (In the case of insert characters, the factor is still the number of lines affected. This is always 1 unless the device has **in** and the software uses it.) When a "*****" is specified, it is sometimes useful to give a delay of the form **3.5** to specify a delay per unit to tenths of milliseconds. (Only one decimal place is allowed.)
- **/** Indicates that the delay is mandatory and padding characters are transmitted regardless of the setting of **xon**. If "**/**" is not specified or if a device has **xon** defined, the delay information is advisory and is only used for cost estimates or when the device is in raw mode. However, any delay specified for **bel** or **flash** is treated as mandatory.

The following notation is valid in **terminfo** source files for specifying special characters:

Notation	Represents Character
^*x*	Control-*x* (for any appropriate *x*)
\a	Alert
\b	Backspace
\E or **\e**	An ESCAPE character
\f	Form feed
\l	Linefeed
\n	Newline
\r	Carriage return
\s	Space
\t	Tab
\ˆ	Caret (ˆ)
****	Backslash (\\)
\\,	Comma (**,**)
\\:	Colon (**:**)
\0	Null
*****nnn*	Any character, specified as three octal digits

(See the "X/Open System Interface Definitions, Issue 4, Version 2" specification, "General Terminal Interface".)

Commented-Out Capabilities

Sometimes individual capabilities must be commented out. To do this, put a period before the capability name. For example, see the second **ind** in the example in the "Sample Entry" section above. Note that capabilities are defined in a left-to-right order and, therefore, a prior definition will override a later definition.

Device Capabilities

Basic Capabilities

The number of columns on each line for the device is given by the **cols** numeric capability. If the device has a screen, then the number of lines on the screen is given by the **lines** capability. If the device wraps around to the beginning of the next line when it reaches the right margin, then it should have the **am** capability. If the terminal can clear its screen, leaving the cursor in the home position, then this is given by the **clear** string capability. If the terminal overstrikes (rather than clearing a position when a character is struck over) then it should have the **os** capability. If the device is a printing terminal, with no soft copy unit, specify both **hc** and **os**. If there is a way to move the cursor to the left edge of the current row, specify this as **cr**. (Normally this will be carriage return, control-M.) If there is a way to produce an audible signal (such as a bell or a beep), specify it as **bel**. If, like most devices, the device uses the XON/XOFF flow-control protocol, specify **xon**.

If there is a way to move the cursor one position to the left (such as backspace), that capability should be given as **cub1**. Similarly, sequences to move to the right, up, and down should be given as **cuf1**, **cuu1**, and **cud1**, respectively. These local cursor motions must not alter the text they pass over; for example, you would not normally use "**cuf1=\s**" because the space would erase the character moved over.

t

A very important point here is that the local cursor motions encoded in **terminfo** are undefined at the left and top edges of a screen terminal. Programs should never attempt to backspace around the left edge, unless **bw** is specified, and should never attempt to go up locally off the top. To scroll text up, a program goes to the bottom left corner of the screen and sends the **ind** (index) string. To scroll text down, a program goes to the top left corner of the screen and sends the **ri** (reverse index) string. The strings **ind** and **ri** are undefined when not on their respective corners of the screen.

Parameterized versions of the scrolling sequences are **indn** and **rin**. These versions have the same semantics as **ind** and **ri**, except that they take one argument and scroll the number of lines specified by that argument. They are also undefined except at the appropriate edge of the screen.

The **am** capability tells whether the cursor sticks at the right edge of the screen when text is output, but this does not necessarily apply to a **cuf1** from the last column. Backward motion from the left edge of the screen is possible only when **bw** is specified. In this case, **cub1** will move to the right edge of the previous row. If **bw** is not given, the effect is undefined. This is useful for drawing a box around the edge of the screen, for example. If the device has switch-selectable automatic margins, **am** should be specified in the **terminfo** source file. In this case, initialization strings should turn on this option, if possible. If the device has a command that moves to the first column of the next line, that command can be given as **nel** (newline). It does not matter if the command clears the remainder of the current line, so if the device has no **cr** and **lf** it may still be possible to craft a working **nel** out of one or both of them.

These capabilities suffice to describe hardcopy and screen terminals. Thus the AT&T 5320 hardcopy terminal is described as follows:

```
5320|att5320|AT&T 5320 hardcopy terminal,
        am, hc, os,
        cols#132,
        bel=^G, cr=\r, cub1=\b, cnd1=\n,
        dch1=\E[P, dl1=\E[M,
        ind=\n,
```

while the Lear Siegler ADM-3 is described as

```
adm3|lsi adm3,
        am, bel=^G, clear=^Z, cols#80, cr=^M, cub1=^H,
        cud1=^J, ind=^J, lines#24,
```

Parameterized Strings

Cursor addressing and other strings requiring arguments are described by a argumentized string capability with escapes in a form (**%***x*) comparable to **printf()** (see *printf*(1)). For example, to address the cursor, the **cup** capability is given, using two arguments: the row and column to address to. (Rows and columns are numbered from zero and refer to the physical screen visible to the user, not to any unseen memory.) If the terminal has memory relative cursor addressing, that can be indicated by **mrcup**.

The argument mechanism uses a stack and special "**%**" codes to manipulate the stack in the manner of Reverse Polish Notation (postfix). Typically a sequence pushes one of the arguments onto the stack and then prints it in some format. Often more complex operations are necessary. Operations are in postfix form with the operands in the usual order. That is, to subtract 5 from the first argument, one would use **%p1%{5}%-**.

The "**%**" encodings have the following meanings:

%% Outputs "**%**".

%[[**:**]*flags*][*width*[*.precision*]][**doxXs**]
As in **printf()**; flags are **[-+#]** and space.

%c Print **pop()** gives **%c**.

%p[1-9] Push the *i*th argument.

%P[a-z] Set dynamic variable [a-z] to **pop()**.

%g[a-z] Get dynamic variable [a-z] and push it.

%P[A-Z] Set static variable [a-z] to **pop()**.

%g[A-Z] Get static variable [a-z] and push it.

%'*c***'** Push **char** constant *c*.

%{*nn***}** Push decimal constant *nn*.

%l Push *strlen*(*pop*()).

%+ %- %* %/ %m
Arithmetic (**%m** is mod): push(pop integer2 op pop integer1) where integer1 represents the top of the stack

%& %| %^ Bit operations: push(pop integer2 op pop integer1)

%= %> %< Logical operations: push(pop integer2 op pop integer1)

%A %O Logical operations: and, or

%! %~ Unary operations: *push*(op *pop*())

%i (For ANSI terminals) add 1 to the first argument (if one argument present), or first two arguments (if more than one argument present).

%? *expr* **%t** *thenpart* **%e** *elsepart* **%;**
If-then-else; **%e** *elsepart* is optional; else-if's are possible as in Algol 68:

t

%? c_1 **%t** b_1 **%e** c_2 **%t** b_2 **%e** c_3 **%t** b_3 **%e** c_4 **%t** b_4 **%e** b_5 **%;**

c_i are conditions; b_i are bodies.

If the "**-**" flag is used with "**%[doxXs]**", then a colon must be placed between the "**%**" and the "**-**" to differentiate the flag from the binary "**%-**" operator. For example: "**%:-16.16s**".

Consider the Hewlett-Packard 2645, which, to get to row 3 and column 12, needs to be sent **\E&a12c03Y** padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are zero-padded as two digits. Thus, its **cup** capability is:

```
cup=\E&a%p2%2.2dc%p1%2.2dY$<6>
```

The Micro-Term ACT-IV needs the current row and column sent preceded by a **^T**, with the row and column simply encoded in binary:

```
cup=^T%p1%c%p2%c
```

Devices that use "**%c**" need to be able to backspace the cursor (**cub1**), and to move the cursor up one line on the screen (**cuu1**). This is necessary because it is not always safe to transmit **\n**, **^D**, and **\r**, as the system may change or discard them. (The library functions dealing with **terminfo** set tty modes so that tabs are never expanded, so **\t** is safe to send. This turns out to be essential for the Ann Arbor 4080.)

A final example is the LSI ADM-3a, which uses row and column offset by a blank character, thus:

```
cup=\E=%p1%'\s'%+%c%p2%'\s'%+%c
```

After sending "**\E=**", this pushes the first argument, pushes the ASCII decimal value for a space (32), adds them (pushing the sum on the stack in place of the two previous values), and outputs that value as a character. Then the same is done for the second argument. More complex arithmetic is possible using the stack.

Cursor Motions

If the terminal has a fast way to home the cursor (to very upper left corner of screen) then this can be given as **home**; similarly a fast way of getting to the lower left-hand corner can be given as **ll**; this may involve going up with **cuu1** from the home position, but a program should never do this itself (unless **ll** does) because it can make no assumption about the effect of moving up from the home position. Note that the home position is the same as addressing to (0,0): to the top left corner of the screen, not of memory. (Thus, the **\EH** sequence on Hewlett-Packard terminals cannot be used for **home** without losing some of the other features on the terminal.)

If the device has row or column absolute-cursor addressing, these can be given as single argument capabilities **hpa** (horizontal position absolute) and **vpa** (vertical position absolute). Sometimes these are shorter than the more general two-argument sequence (as with the Hewlett-Packard 2645) and can be used in preference to **cup**. If there are argumentised local motions (such as "move *n* spaces to the right"), these can be given as **cud**, **cub**, **cuf**, and **cuu** with a single argument indicating how many spaces to move. These are primarily useful if the device does not have **cup**, such as the Tektronix 4025.

If the device needs to be in a special mode when running a program that uses these capabilities, the codes to enter and exit this mode can be given as **smcup** and **rmcup**. This arises, for example, from terminals, such as the Concept, with more than one page of memory. If the device has only memory relative cursor addressing and not screen relative cursor addressing, a one screen-sized window must be fixed into the device for cursor addressing to work properly. This is also used for the Tektronix 4025, where **smcup** sets the command character to be the one used by **terminfo**. If the **rmcup** sequence will not restore the screen after an **smcup** sequence is output (to the state prior to outputting **smcup**), specify **nrrmc**.

Area Clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as **el**. If the terminal can clear from the beginning of the line to the current position inclusive, leaving the cursor where it is, this should be given as **el1**. If the terminal can clear from the current position to the end of the display, then this should be given as **ed**. **ed** is only defined from the first column of a line. (Thus, it can be simulated by a request to delete a large number of lines, if a true **ed** is not available.)

Insert/Delete Line

If the terminal can open a new blank line before the line where the cursor is, this should be given as **il1**; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line which the cursor is on, then this should be given as **dl1**; this is done only from the first position on the line to be deleted. Versions of **il1** and **dl1** which take a single argument and insert or delete that many lines can be given as **il** and **dl**.

If the terminal has a settable destructive scrolling region (like the VT100) the command to set this can be described with the **csr** capability, which takes two arguments: the top and bottom lines of the scrolling region. The cursor position is, alas, undefined after using this command. It is possible to get the effect of insert or delete line using this command — the **sc** and **rc** (save and restore cursor) commands are also useful. Inserting lines at the top or bottom of the screen can also be done using **ri** or **ind** on many terminals without a true insert/delete line, and is often faster even on terminals with those features.

To determine whether a terminal has destructive scrolling regions or nondestructive scrolling regions, create a scrolling region in the middle of the screen, place data on the bottom line of the scrolling region, move the cursor to the top line of the scrolling region, and do a reverse index (**ri**) followed by a delete line (**dl1**) or index (**ind**). If the data that was originally on the bottom line of the scrolling region was restored into the scrolling region by the **dl1** or **ind**, then the terminal has nondestructive scrolling regions. Otherwise, it has destructive scrolling regions. Do not specify **csr** if the terminal has nondestructive scrolling regions, unless **ind**, **ri**, **indn**, **rin**, **dl**, and **dl1** all simulate destructive scrolling.

If the terminal has the ability to define a window as part of memory, which all commands affect, it should be given as the argumentized string **wind**. The four arguments are the starting and ending lines in memory and the starting and ending columns in memory, in that order.

If the terminal can retain display memory above, then the **da** capability should be given; if display memory can be retained below, then **db** should be given. These indicate that deleting a line or scrolling a full screen may bring nonblank lines up from below or that scrolling back with **ri** may bring down nonblank lines.

Insert/Delete Character

There are two basic kinds of intelligent terminals with respect to insert/delete character operations which can be described using **terminfo.** The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept 100 and the Perkin-Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated, or expanded to two untyped blanks. You can determine the kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type "**abc def**" using local cursor motions (not spaces) between the **abc** and the **def**. Then position the cursor before the **abc** and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the **abc** shifts over to the **def** which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability **in**, which stands for "insert null". While these are two logically separate attributes (one line versus multiline insert mode, and special treatment of untyped spaces) we have seen no terminals whose insert mode cannot be described with the single attribute.

terminfo can describe both terminals that have an insert mode and terminals which send a simple sequence to open a blank position on the current line. Give as **smir** the sequence to get into insert mode. Give as **rmir** the sequence to leave insert mode. Now give as **ich1** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give **ich1**; terminals that send a sequence to open a screen position should give it here. (If your terminal has both, insert mode is usually preferable to **ich1**. Do not give both unless the terminal requires both to be used in combination.) If post-insert padding is needed, give this as a number of milliseconds padding in **ip** (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in **ip**. If your terminal needs both to be placed into an "insert mode" and a special code to precede each inserted character, then both **smir**/**rmir** and **ich1** can be given, and both will be used. The **ich** capability, with one argument, *n*, will insert *n* blanks.

If padding is necessary between characters typed while not in insert mode, give this as a number of milliseconds padding in **rmp**.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (for example, if there is a tab after the insertion position). If your terminal allows motion while in insert mode you can give the capability **mir** to speed up inserting in this case. Omitting **mir** will affect only speed. Some terminals (notably Datamedia) must not have **mir** because of the way their insert mode works.

Finally, you can specify **dch1** to delete a single character, **dch** with one argument, *n*, to delete *n* characters, and delete mode by giving **smdc** and **rmdc** to enter and exit delete mode (any mode the terminal needs to be placed in for **dch1** to work).

A command to erase *n* characters (equivalent to outputting *n* blanks without moving the cursor) can be given as **ech** with one argument.

Highlighting, Underlining, and Visible Bells

Your device may have one or more kinds of display attributes that allow you to highlight selected characters when they appear on the screen. The following display modes (shown with the names by which they are set) may be available:

- A blinking screen (**blink**)
- Bold or extra-bright characters (**bold**)
- Dim or half-bright characters (**dim**)
- Blanking or invisible text (**invis**)
- Protected text (**prot**)
- A reverse-video screen (**rev**)
- An alternate character set (**smacs** to enter this mode and **rmacs** to exit it). (If a command is necessary before you can enter alternate character set mode, give the sequence in **enacs** or "enable alternate-character-set" mode.) Turning on any of these modes singly may turn off other modes.

sgr0 should be used to turn off all video enhancement capabilities. It should always be specified because it represents the only way to turn off some capabilities, such as **dim** or **blink**.

Choose one display method as **standout mode** and use it to highlight error messages and other text to which you want to draw attention. Choose a form of display that provides strong contrast but that is easy on the eyes. (We recommend reverse-video plus half-bright or reverse-video alone.) The sequences to enter and exit standout mode are given as **smso** and **rmso**, respectively. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TVI 912 and Teleray 1061 do, then **xmc** should be given to tell how many spaces are left.

Sequences to begin underlining and end underlining can be specified as **smul** and **rmul**, respectively. If the device has a sequence to underline the current character and to move the cursor one space to the right (such as the Micro-Term MIME), this sequence can be specified as **uc**.

Terminals with the "magic cookie" glitch (**xmc**) deposit special "cookies" when they receive mode-setting sequences, which affect the display algorithm rather than having extra bits for each character. Some terminals, such as the Hewlett-Packard 2621, automatically leave standout mode when they move to a new line or the cursor is addressed. Programs using standout mode should exit standout mode before moving the cursor or sending a newline, unless the **msgr** capability, asserting that it is safe to move in standout mode, is present.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement), then this can be given as **flash**; it must not move the cursor. A good flash can be done by changing the screen into reverse video, pad for 200 ms, then return the screen to normal video.

If the cursor needs to be made more visible than normal when it is not on the bottom line (to make, for example, a nonblinking underline into an easier to find block or blinking underline) give this sequence as **cvvis**. The boolean **chts** should also be given. If there is a way to make the cursor completely invisible, give that as **civis**. The capability **cnorm** should be given, which undoes the effects of either of these modes.

If your terminal generates underlined characters by using the underline character (with no special sequences needed) even though it does not otherwise overstrike characters, then specify the capability **ul**. For devices on which a character overstriking another leaves both characters on the screen, specify the capability **os**. If overstrikes are erasable with a blank, then this should be indicated by specifying **eo**.

If there is a sequence to set arbitrary combinations of modes, this should be given as **sgr** (set attributes), taking nine arguments. Each argument is either 0 or nonzero, as the corresponding attribute is on or off. The nine arguments are, in order: standout, underline, reverse, blink, dim, bold, blank, protect, alternate character set. Not all modes need to be supported by **sgr**; only those for which corresponding separate attribute commands exist should be supported. For example, let's assume that the terminal in question needs the following escape sequences to turn on various modes.

tparm() Argument	**Attribute**	**Escape Sequence**
	none	`\E[0m`
p1	standout	`\E[0;4;7m`
p2	underline	`\E[0;3m`
p3	reverse	`\E[0;4m`
p4	blink	`\E[0;5m`
p5	dim	`\E[0;7m`
p6	bold	`\E[0;3;4m`
p7	invis	`\E[0;8m`
p8	protect	not available
p9	altcharset	`^O` (off) `^N` (on)

Note that each escape sequence requires a 0 to turn off other modes before turning on its own mode. Also note that, as suggested above, *standout* is set up to be the combination of *reverse* and *dim*. Also, because this terminal has no *bold* mode, *bold* is set up as the combination of *reverse* and *underline*. In addition, to allow combinations, such as *underline+blink*, the sequence to use would be **\E[0;3;5m**. The terminal doesn't have *protect* mode, either, but that cannot be simulated in any way, so **p8** is ignored. The *altcharset* mode is different in that it is either **^O** or **^N**, depending on whether it is off or on. If all modes were to be turned on, the sequence would be:

```
\E[0;3;4;5;7;8m^N
```

Now look at when different sequences are output. For example, **;3** is output when either **p2** or **p6** is true, that is, if either *underline* or *bold* modes are turned on. Writing out the above sequences, along with their dependencies, gives the following:

Sequence	**When to Output**	**terminfo** Translation
`\E[0`	always	`\E[0`
`;3`	if **p2** or **p6**	`%?%p2%p6%|%t;3%;`
`;4`	if **p1** or **p3** or **p6**	`%?%p1%p3%|%p6%|%t;4%;`
`;5`	if **p4**	`%?%p4%t;5%;`
`;7`	if **p1** or **p5**	`%?%p1%p5%|%t;7%;`
`;8`	if **p7**	`%?%p7%t;8%;`
`m`	always	`m`
`^N` or `^O`	if **p9**, **^N**; else **^O**	`%?%p9%t^N%e^O%;`

Putting this all together into the **sgr** sequence gives:

```
sgr=\E[0%?%p2%p6%|%t;3%;%?%p1%p3%|%p6%
        |%t;4%;%?%p5%t;5%;%?%p1%p5%
        |%t;7%;%?%p7%t;8%;m%?%p9%t^N%e^O%;,
```

Remember that **sgr** and **sgr0** must always be specified.

Keypad

If the device has a keypad that transmits sequences when the keys are pressed, this information can also be specified. Note that it is not possible to handle devices where the keypad only works in local (this applies, for example, to the unshifted Hewlett-Packard 2621 keys). If the keypad can be set to transmit or not transmit, specify these sequences as **smkx** and **rmkx**. Otherwise the keypad is assumed to always transmit.

The sequences sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as **kcub1**, **kcuf1**, **kcuu1**, **kcud1** and **khome**, respectively. If there are function keys such as f0, f1, ..., f63, the sequences they send can be specified as **kf0**, **kf1**, ..., **kf63**. If the first 11 keys have labels other than the default f0 through f10, the labels can be given as **lf0**, **lf1**, ..., **lf10**.

The codes transmitted by certain other special keys can be given: **kll** (home down), **kbs** (backspace), **ktbc** (clear all tabs), **kctab** (clear the tab stop in this column), **kclr** (clear screen or erase key), **kdch1** (delete character), **kdl1** (delete line), **krmir** (exit insert mode), **kel** (clear to end of line), **ked** (clear to end of screen), **kich1** (insert character or enter insert mode), **kil1** (insert line), **knp** (next page), **kpp** (previous page), **kind** (scroll forward/down), **kri** (scroll backward/up), **khts** (set a tab stop in this column). In addition, if the keypad has a 3 by 3 array of keys including the four arrow keys, the other five keys can be given as **ka1**, **ka3**, **kb2**, **kc1**, and **kc3**. These keys are useful when the effects of a 3 by 3 directional pad are needed. Further keys are defined above in the capabilities list.

t

Strings to program function keys can be specified as **pfkey**, **pfloc**, and **pfx**. A string to program screen labels should be specified as **pln**. Each of these strings takes two arguments: a function key identifier and a string to program it with. **pfkey** causes pressing the given key to be the same as the user typing the given string; **pfloc** causes the string to be executed by the terminal in local mode; and **pfx** causes the string to be transmitted to the computer. The capabilities **nlab**, **lw** and **lh** define the number of programmable screen labels and their width and height. If there are commands to turn the labels on and off, give them in **smln** and **rmln**. **smln** is normally output after one or more **pln** sequences to make sure that the change becomes visible.

Tabs and Initialization

If the device has hardware tabs, the command to advance to the next tab stop can be given as **ht** (usually control-I). A "backtab" command that moves leftward to the next tab stop can be given as **cbt**. By convention, if tty modes show that tabs are being expanded by the computer rather than being sent to the device, programs should not use **ht** or **cbt** (even if they are present) because the user might not have the tab stops properly set. If the device has hardware tabs that are initially set every *n* spaces when the device is powered up, the numeric argument **it** is given, showing the number of spaces the tabs are set to. This is normally used by **tput init** to determine whether to set the mode for hardware tab expansion and whether to set the tab stops. If the device has tab stops that can be saved in nonvolatile memory, the **terminfo** description can assume that they are properly set. If there are commands to set and clear tab stops, they can be given as **tbc** (clear all tab stops) and **hts** (set a tab stop in the current column of every row).

Other capabilities include: **is1**, **is2**, and **is3**, initialization strings for the device; **iprog**, the path name of a program to be run to initialize the device; and **if**, the name of a file containing long initialization strings. These strings are expected to set the device into modes consistent with the rest of the **terminfo** description. They must be sent to the device each time the user logs in and be output in the following order: run the program **iprog**; output **is1**; output **is2**; set the margins using **mgc**, **smgl** and **smgr**; set the tabs using **tbc** and **hts**; print the file **if**; and finally output **is3**. This is usually done using the **init** option of **tput**.

Most initialization is done with **is2**. Special device modes can be set up without duplicating strings by putting the common sequences in **is2** and special cases in **is1** and **is3**. Sequences that do a reset from a totally unknown state can be given as **rs1**, **rs2**, **rf**, and **rs3**, analogous to **is1**, **is2**, **is3**, and **if**. (The method using files, **if** and **rf**, is used for a few terminals; however, the recommended method is to use the initialization and reset strings.) These strings are output by **tput reset**, which is used when the terminal gets into a wedged state. Commands are normally placed in **rs1**, **rs2**, **rs3**, and **rf** only if they produce annoying effects on the screen and are not necessary when logging in. For example, the command to set a terminal into 80-column mode would normally be part of **is2**, but on some terminals it causes an annoying glitch on the screen and is not normally needed because the terminal is usually already in 80-column mode.

If a more complex sequence is needed to set the tabs than can be described by using **tbc** and **hts**, the sequence can be placed in **is2** or **if**.

Any margin can be cleared with **mgc**. (For instructions on how to specify commands to set and clear margins, see the "Margins" subsection of the "Capabilities That Cause Movement" section below.

t

Delays

Certain capabilities control padding in the **tty** driver. These are primarily needed by hard-copy terminals, and are used by **tput init** to set tty modes appropriately (see *tput*(1)). Delays embedded in the capabilities **cr**, **ind**, **cub1**, **ff**, and **tab** can be used to set the appropriate delay bits to be set in the tty driver. If **pb** (padding baud rate) is given, these values can be ignored at baud rates below the value of **pb**.

Status Lines

If the terminal has an extra "status line" that is not normally used by software, this fact can be indicated. If the status line is viewed as an extra line below the bottom line, into which one can cursor-address normally (such as the Heathkit H19's 25th line, or the 24th line of a VT100 which is set to a 23-line scrolling region), the capability **hs** should be given. Special strings that go to a given column of the status line and return from the status line can be given as **tsl** and **fsl**. (**fsl** must leave the cursor position in the same place it was before **tsl**. If necessary, the **sc** and **rc** strings can be included in **tsl** and **fsl** to get this effect.) The capability **tsl** takes one argument, which is the column number of the status line the cursor is to be moved to.

If escape sequences and other special commands, such as tab, work while in the status line, the flag **eslok** can be given. A string which turns off the status line (or otherwise erases its contents) should be given as **dsl**. If the terminal has commands to save and restore the position of the cursor, give them as **sc** and

rc. The status line is normally assumed to be the same width as the rest of the screen (that is, **cols**). If the status line is a different width (possibly because the terminal does not allow an entire line to be loaded) the width, in columns, can be indicated with the numeric argument **wsl**.

Line Graphics

If the device has a line drawing alternate character set, the mapping of glyph to character would be given in **acsc**. The definition of this string is based on the alternate character set used in the Digital VT100 terminal, extended slightly with some characters from the AT&T 4410v1 terminal.

Glyph Name	VT100+ Character
arrow pointing right	**+**
arrow pointing left	**,**
arrow pointing down	**.**
solid square block	**0**
lantern symbol	**I**
arrow pointing up	**-**
diamond	**`**
checker board (stipple)	**a**
degree symbol	**f**
plus/minus	**g**
board of squares	**h**
lower right corner	**j**
upper right corner	**k**
upper left corner	**l**
lower left corner	**m**
plus	**n**
scan line 1	**o**
horizontal line	**q**
scan line 9	**s**
left tee (├)	**t**
right tee (┤)	**u**
bottom tee (┴)	**v**
top tee (┬)	**w**
vertical line	**x**
bullet	**~**

The best way to describe a new device's line graphics set is to add a third column to the above table with the characters for the new device that produce the appropriate glyph when the device is in alternate-character-set mode. For example:

Glyph Name	VT100+ Character	Character Used on New Device
upper left corner	**l**	**R**
lower left corner	**m**	**F**
upper right corner	**k**	**T**
lower right corner	**j**	**G**
horizontal line	**q**	**,**
vertical line	**x**	**.**

Now write down the characters left to right; for example:

```
acsc=lRmFkTjGq\,x.
```

In addition, **terminfo** lets you define multiple character sets (see the "Alternate Character Sets" section below.

Color Manipulation

Most color terminals belong to one of two classes of terminal:

- **Tektronix-style**

  The Tektronix method uses a set of N predefined colors (usually 8) from which an application can select "current" foreground and background colors. Thus a terminal can support up to N colors mixed into N*N color-pairs to be displayed on the screen at the same time.

t

- **Hewlett-Packard-style**

  In the HP method, the application cannot define the foreground independently of the background, or vice-versa. Instead, the application must define an entire color-pair at once. Up to M color-pairs, made from 2*M different colors, can be defined this way.

The numeric variables **colors** and **pairs** define the number of colors and color-pairs that can be displayed on the screen at the same time. If a terminal can change the definition of a color (for example, the Tektronix 4100 and 4200 series terminals), this should be specified with **ccc** (can change color). To change the definition of a color (Tektronix 4200 method), use **initc** (initialize color). It requires four arguments: color number (ranging from 0 to **colors**–1) and three RGB (red, green, and blue) values or three HLS colors (Hue, Lightness, Saturation). Ranges of RGB and HLS values are terminal-dependent.

Tektronix 4100 series terminals only use HLS color notation. For such terminals (or dual-mode terminals to be operated in HLS mode) one must define a boolean variable **hls**; that would instruct the **init_color()** function (see *can_change_color*(3X)) to convert its RGB arguments to HLS before sending them to the terminal. The last three arguments to the **initc** string would then be HLS values.

If a terminal can change the definitions of colors, but uses a color notation different from RGB and HLS, a mapping to either RGB or HLS must be developed.

If the terminal supports ANSI escape sequences to set background and foreground, they should be coded as **setab** and **setaf**, respectively. If the terminal supports other escape sequences to set background and foreground, they should be coded as **setb** and **setf**, respectively. The **vidputs()** function (see *vidattr*(3X)) and the refresh functions use **setab** and **setaf** if they are defined. Each of these capabilities requires one argument: the number of the color. By convention, the first eight colors (0–7) map to, in order: black, red, green, yellow, blue, magenta, cyan, white. However, color re-mapping may occur or the underlying hardware may not support these colors. Mappings for any additional colors supported by the device (that is, to numbers greater than 7) are at the discretion of the **terminfo** entry writer.

To initialize a color-pair (HP method), use **initp** (initialize pair). It requires seven arguments: the number of a color-pair (range=0 to **pairs**–1), and six RGB values: three for the foreground followed by three for the background. (Each of these groups of three should be in the order RGB.) When **initc** or **initp** are used, RGB or HLS arguments should be in the order "red, green, blue" or "hue, lightness, saturation"), respectively. To make a color-pair current, use **scp** (set color-pair). It takes one argument, the number of a color-pair.

Some terminals (for example, most color terminal emulators for PCs) erase areas of the screen with current background color. In such cases, **bce** (background color erase) should be defined. The variable **op** (original pair) contains a sequence for setting the foreground and the background colors to what they were at the terminal start-up time. Similarly, **oc** (original colors) contains a control sequence for setting all colors (for the Tektronix method) or color-pairs (for the HP method) to the values they had at the terminal start-up time.

Some color terminals substitute color for video attributes. Such video attributes should not be combined with colors. Information about these video attributes should be packed into the **ncv** (no color video) variable. There is a one-to-one correspondence between the nine least significant bits of that variable and the video attributes. The following table depicts this correspondence.

Attribute	Bit Position	Decimal Value	Characteristic That Sets
WA_STANDOUT	0	1	**sgr**, parameter 1
WA_UNDERLINE	1	2	**sgr**, parameter 2
WA_REVERSE	2	4	**sgr**, parameter 3
WA_BLINK	3	8	**sgr**, parameter 4
WA_DIM	4	16	**sgr**, parameter 5
WA_BOLD	5	32	**sgr**, parameter 6
WA_INVIS	6	64	**sgr**, parameter 7
WA_PROTECT	7	128	**sgr**, parameter 8
WA_ALTCHARSET	8	256	**sgr**, parameter 9
WA_HORIZONTAL	9	512	**sgr1**, parameter 1
WA_LEFT	10	1024	**sgr1**, parameter 2
WA_LOW	11	2048	**sgr1**, parameter 3
WA_RIGHT	12	4096	**sgr1**, parameter 4
WA_TOP	13	8192	**sgr1**, parameter 5
WA_VERTICAL	14	16384	**sgr1**, parameter 6

When a particular video attribute should not be used with colors, set the corresponding **ncv** bit to 1; otherwise set it to 0. To determine the information to pack into the **ncv** variable, add the decimal values corresponding to those attributes that cannot coexist with colors. For example, if the terminal uses colors to simulate reverse video (bit number 2 and decimal value 4) and bold (bit number 5 and decimal value 32), the resulting value for **ncv** will be 36 (4 + 32).

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as **pad**. Only the first character of the **pad** string is used. If the terminal does not have a pad character, specify **npc**.

If the terminal can move up or down half a line, this can be indicated with **hu** (half-line up) and **hd** (half-line down). This is primarily useful for superscripts and subscripts on hardcopy terminals. If a hardcopy terminal can eject to the next page (form feed), give this as **ff** (usually control-L).

If there is a command to repeat a given character a given number of times (to save time transmitting a large number of identical characters) this can be indicated with the argumentized string **rep**. The first argument is the character to be repeated and the second is the number of times to repeat it. Thus, **tparm(repeat_char, 'x', 10)** is the same as **xxxxxxxxxx**.

If the terminal has a settable command character, such as the Tektronix 4025, this can be indicated with **cmdch**. A prototype command character is chosen which is used in all capabilities. This character is given in the **cmdch** capability to identify it. The following convention is supported on some systems: If the environment variable **CC** exists, all occurrences of the prototype character are replaced with the character in **CC**.

Terminal descriptions that do not represent a specific kind of known terminal, such as *switch*, *dialup*, *patch*, and *network*, should include the **gn** (generic) capability so that programs can complain that they do not know how to talk to the terminal. (This capability does not apply to *virtual* terminal descriptions for which the escape sequences are known.) If the terminal is one of those supported by the virtual terminal protocol, the terminal number can be given as **vt**. A line-turn-around sequence to be transmitted before doing reads should be specified in **rfi**.

If the device uses XON/XOFF handshaking for flow control, give **xon**. Padding information should still be included so that functions can make better decisions about costs, but actual pad characters will not be transmitted. Sequences to turn on and off XON/XOFF handshaking may be given in **smxon** and **rmxon**. If the characters used for handshaking are not **^S** and **^Q**, they may be specified with **xonc** and **xoffc**.

If the terminal has a "meta key" which acts as a shift key, setting the 8th bit of any character transmitted, this fact can be indicated with **km**. Otherwise, software will assume that the 8th bit is parity and it will usually be cleared. If strings exist to turn this "meta mode" on and off, they can be given as **smm** and **rmm**.

If the terminal has more lines of memory than will fit on the screen at once, the number of lines of memory can be indicated with **lm**. A value of **lm#0** indicates that the number of lines is not fixed, but that there is still more memory than fits on the screen.

Media copy strings which control an auxiliary printer connected to the terminal can be given as:

mc0 Print the contents of the screen.
mc4 Turn off the printer.
mc5 Turn on the printer.

When the printer is on, all text sent to the terminal will be sent to the printer. A variation, **mc5p**, takes one argument, and leaves the printer on for as many characters as the value of the argument, then turns the printer off. The argument should not exceed 255. If the text is not displayed on the terminal screen when the printer is on, specify **mc5i** (silent printer). All text, including **mc4**, is transparently passed to the printer while an **mc5p** is in effect.

Special Cases

The working model used by **terminfo** fits most terminals reasonably well. However, some terminals do not completely match that model, requiring special support by **terminfo**. These are not meant to be construed as deficiencies in the terminals; they are just differences between the working model and the actual hardware. They may be unusual devices or, for some reason, do not have all the features of the **terminfo** model implemented.

Terminals that cannot display tilde (˜) characters, such as certain Hazeltine terminals, should indicate **hz**.

Terminals that ignore a linefeed immediately after an **am** wrap, such as the Concept 100, should indicate **xenl**. Those terminals whose cursor remains on the right-most column until another character has been received, rather than wrapping immediately upon receiving the right-most character, such as the VT100, should also indicate **xenl**.

If **el** is required to get rid of standout (instead of writing normal text on top of it), **xhp** should be given.

Those Teleray terminals whose tabs turn all characters moved over to blanks, should indicate **xt** (destructive tabs). This capability is also taken to mean that it is not possible to position the cursor on top of a "magic cookie". Therefore, to erase standout mode, it is necessary, instead, to use delete and insert line.

For Beehive Superbee terminals that do not transmit the escape or control-C characters, specify **xsb**, indicating that the f1 key is to be used for escape and the f2 key for control-C.

Similar Terminals

If there are two similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **use** can be given with the name of the similar terminal. The capabilities given before **use** override those in the terminal type invoked by **use**. A capability can be canceled by placing *capability-name*@ prior to the appearance of the string capability **use**. For example, the entry:

```
att4424-2|Teletype 4424 in display function group ii,
        rev@, sgr@, smul@, use=att4424,
```

defines an AT&T 04424 terminal that does not have the **rev**, **sgr**, and **smul** capabilities, and hence cannot do highlighting. This is useful for different modes for a terminal, or for different user preferences. More than one **use** capability may be given.

t

Printer Capabilities

The **terminfo** database lets you define capabilities of printers as well as terminals. Capabilities available for printers are included in the lists in the "Defined Capabilities" section above.

Rounding Values

Because argumentized string capabilities work only with integer values, **terminfo** designers should create strings that expect numeric values that have been rounded. Application designers should note this and should always round values to the nearest integer before using them with a argumentized string capability.

Printer Resolution

A printer's resolution is defined to be the smallest spacing of characters it can achieve. In general, the horizontal and vertical resolutions are independent. Thus the vertical resolution of a printer can be determined by measuring the smallest achievable distance between consecutive printing baselines, while the horizontal resolution can be determined by measuring the smallest achievable distance between the leftmost edges of consecutive printed, identical, characters.

All printers are assumed to be capable of printing with a uniform horizontal and vertical resolution. The view of printing that **terminfo** currently presents is one of printing inside a uniform matrix: All characters are printed at fixed positions relative to each "cell" in the matrix; furthermore, each cell has the same size given by the smallest horizontal and vertical step sizes dictated by the resolution. (The cell size can be changed as will be seen later.)

Many printers are capable of "proportional printing", where the horizontal spacing depends on the size of the character last printed. **terminfo** does not make use of this capability, although it does provide enough capability definitions to allow an application to simulate proportional printing.

A printer must not only be able to print characters as close together as the horizontal and vertical resolutions suggest, but also of "moving" to a position an integral multiple of the smallest distance away from a previous position. Thus printed characters can be spaced apart a distance that is an integral multiple of the smallest distance, up to the length or width of a single page.

Some printers can have different resolutions depending on different "modes". In "normal mode", the existing **terminfo** capabilities are assumed to work on columns and lines, just like a video terminal. Thus the old **lines** capability would give the length of a page in lines, and the **cols** capability would give the width of a page in columns. In "micro mode," many **terminfo** capabilities work on increments of lines and columns. With some printers the micro mode may be concomitant with normal mode, so that all the capabilities work at the same time.

Specifying Printer Resolution

The printing resolution of a printer is given in several ways. Each specifies the resolution as the number of smallest steps per distance:

Characteristic Number of Smallest Steps	
orhi	Steps per inch horizontally
orvi	Steps per inch vertically
orc	Steps per column
orl	Steps per line

When printing in normal mode, each character printed causes movement to the next column, except in special cases described later; the distance moved is the same as the per-column resolution. Some printers cause an automatic movement to the next line when a character is printed in the rightmost position; the distance moved vertically is the same as the per-line resolution. When printing in micro mode, these distances can be different, and may be zero for some printers.

Automatic Motion after Printing	
Normal Mode:	
orc	Steps moved horizontally
orl	Steps moved vertically
Micro Mode:	
mcs	Steps moved horizontally
mls	Steps moved vertically

Some printers are capable of printing wide characters. The distance moved when a wide character is printed in normal mode may be different from when a regular width character is printed. The distance moved when a wide character is printed in micro mode may also be different from when a regular character is printed in micro mode, but the differences are assumed to be related: If the distance moved for a regular character is the same whether in normal mode or micro mode (**mcs**=**orc**), then the distance moved for a wide character is also the same whether in normal mode or micro mode. This doesn't mean the normal character distance is necessarily the same as the wide character distance, just that the distances don't change with a change in normal to micro mode. However, if the distance moved for a regular character is different in micro mode from the distance moved in normal mode (**mcs**<**orc**), the micro mode distance is assumed to be the same for a wide character printed in micro mode, as the table below shows.

Automatic Motion after Printing Wide Character	
Normal Mode or *Micro Mode* (**mcs** = **orc**):	
widcs	Steps moved horizontally
Micro Mode (**mcs** < **orc**):	
mcs	Steps moved horizontally

There may be control sequences to change the number of columns per inch (the character pitch) and to change the number of lines per inch (the line pitch). If these are used, the resolution of the printer changes, but the type of change depends on the printer:

t

Changing the Character/Line Pitches

cpi	Change character pitch
cpix	If set, **cpi** changes **orhi**; otherwise, changes **orc**
lpi	Change line pitch
lpix	If set, **lpi** changes **orvi**; otherwise, changes **orl**
chr	Change steps per column
cvr	Change steps per line

The **cpi** and **lpi** string capabilities are each used with a single argument, the pitch in columns (or characters) and lines per inch, respectively. The **chr** and **cvr** string capabilities are each used with a single argument, the number of steps per column and line, respectively.

Using any of the control sequences in these strings will imply a change in some of the values of **orc**, **orhi**, **orl**, and **orvi**. Also, the distance moved when a wide character is printed, **widcs**, changes in relation to **orc**. The distance moved when a character is printed in micro mode, **mcs**, changes similarly, with one exception: if the distance is 0 or 1, then no change is assumed.

Programs that use **cpi**, **lpi**, **chr**, or **cvr** should recalculate the printer resolution (and should recalculate other values. See the "Effect of Changing Printing Resolution" section below.

Effects of Changing the Character/Line Pitches

Before	*After*
Using **cpi** with **cpix** clear:	
orhi′	**orhi**
orc′	**orc = orhi /** *Vcpi*
Using **cpi** with **cpix** set:	
orhi′	**orhi = orc *** *Vcpi*
orc′	**orc**
Using **lpi** with **lpix** clear:	
orvi′	**orvi**
orl′	**orl = orvi /** *Vlpi*
Using **lpi** with **lpix** set:	
orvi′	**orvi = orl *** *Vlp*
orl′	**orl**
Using **chr:**	
orhi′	**orhi**
orc′	*Vchr*
Using **cvr:**	
orvi′	**orvi**
orl′	*Vcvr*
Using **cpi** or **chr:**	
widcs′	**widcs = widcs′ * orc / orc′**
mcs′	**mcs = mcs′ * orc / orc′**

Vchr, *Vcpi*, *Vcvr*, and *Vlpi* are the arguments used with **chr**, **cpi**, **cvr**, and **lpi**, respectively. The prime marks (**′**) indicate the old values.

Capabilities That Cause Movement

In the following descriptions, "movement" refers to the motion of the "current position". With video terminals this would be the cursor; with some printers, this is the carriage position. Other printers have different equivalents. In general, the current position is where a character would be displayed if printed.

terminfo has string capabilities for control sequences that cause movement a number of full columns or lines. It also has equivalent string capabilities for control sequences that cause movement a number of smallest steps.

t

String Capabilities for Motion	
mcub1	Move 1 step left
mcuf1	Move 1 step right
mcuu1	Move 1 step up
mcud1	Move 1 step down
mcub	Move *N* steps left
mcuf	Move *N* steps right
mcuu	Move *N* steps up
mcud	Move *N* steps down
mhpa	Move *N* steps from the left
mvpa	Move *N* steps from the top

The latter six strings are each used with a single argument, *N*.

Sometimes the motion is limited to less than the width or length of a page. Also, some printers don't accept absolute motion to the left of the current position. **terminfo** has capabilities for specifying these limits.

Limits to Motion	
mjump	Limit on use of **mcub1**, **mcuf1**, **mcuu1**, **mcud1**
maddr	Limit on use of **mhpa**, **mvpa**
xhpa	If set, **hpa** and **mhpa** can't move left
xvpa	If set, **vpa** and **mvpa** can't move up

If a printer needs to be in a "micro mode" for the motion capabilities described above to work, there are string capabilities defined to contain the control sequence to enter and exit this mode. A boolean is available for those printers where using a carriage return causes an automatic return to normal mode.

Entering/Exiting Micro Mode	
smicm	Enter micro mode
rmicm	Exit micro mode
crxm	Using **cr** exits micro mode

The movement made when a character is printed in the rightmost position varies among printers. Some make no movement, some move to the beginning of the next line, others move to the beginning of the same line. **terminfo** has boolean capabilities for describing all three cases.

What Happens After Character Printed in Rightmost Position	
sam	Automatic move to beginning of same line

Some printers can be put in a mode where the normal direction of motion is reversed. This mode can be especially useful when there are no capabilities for leftward or upward motion, because those capabilities can be built from the motion reversal capability and the rightward or downward motion capabilities. It is best to leave it up to an application to build the leftward or upward capabilities, though, and not enter them in the **terminfo** database. This allows several reverse motions to be strung together without intervening wasted steps that leave and reenter reverse mode.

Entering/Exiting Reverse Modes	
slm	Reverse sense of horizontal motions
rlm	Restore sense of horizontal motions
sum	Reverse sense of vertical motions
rum	Restore sense of vertical motions
While sense of horizontal motion is reversed:	
mcub1	Move 1 step right
mcuf1	Move 1 step left
mcub	Move *N* steps right
mcuf	Move *N* steps left
cub1	Move 1 column right
cuf1	Move 1 column left
cub	Move *N* columns right
cuf	Move *N* columns left
While sense of vertical motion is reversed:	
mcuu1	Move 1 step down
mcud1	Move 1 step up
mcuu	Move *N* steps down
mcud	Move *N* steps up
cuu1	Move 1 line down
cud1	Move 1 line up
cuu	Move *N* lines down
cud	Move *N* lines up

The reverse motion modes should not affect the **mvpa** and **mhpa** absolute motion capabilities. The reverse vertical motion mode should, however, also reverse the action of the line "wrapping" that occurs when a character is printed in the right-most position. Thus printers that have the standard **terminfo** capability **am** defined should experience motion to the beginning of the previous line when a character is printed in the rightmost position in reverse vertical motion mode.

The action when any other motion capabilities are used in reverse motion modes is not defined; thus, programs must exit reverse motion modes before using other motion capabilities.

Two miscellaneous capabilities complete the list of motion capabilities. One of these is needed for printers that move the current position to the beginning of a line when certain control characters, such as linefeed or formfeed, are used. The other is used for the capability of suspending the motion that normally occurs after printing a character.

Miscellaneous Motion Strings	
docr	List of control characters causing **cr**
zerom	Prevent auto motion after printing next single character

Margins

terminfo provides two strings for setting margins on terminals: one for the left and one for the right margin. Printers, however, have two additional margins, for the top and bottom margins of each page. Furthermore, some printers require not using motion strings to move the current position to a margin and then fixing the margin there, but require the specification of where a margin should be regardless of the current position. Therefore **terminfo** offers six additional strings for defining margins with printers.

Setting Margins	
smgl	Set left margin at current column
smgr	Set right margin at current column
smgb	Set bottom margin at current line
smgt	Set top margin at current line
smgbp	Set bottom margin at line *N*
smglp	Set left margin at column *N*
smgrp	Set right margin at column *N*
smgtp	Set top margin at line *N*

The last four strings are used with one or more arguments that give the position of the margin or margins to set. If both of **smglp** and **smgrp** are set, each is used with a single argument, *N*, that gives the column number of the left and right margin, respectively. If both of **smgtp** and **smgbp** are set, each is used to set the top and bottom margin, respectively: **smgtp** is used with a single argument, *N*, the line number of the top margin; however, **smgbp** is used with two arguments, *N* and *M*, that give the line number of the bottom margin, the first counting from the top of the page and the second counting from the bottom. This accommodates the two styles of specifying the bottom margin in different manufacturers' printers. When coding a **terminfo** entry for a printer that has a settable bottom margin, only the first or second argument should be used, depending on the printer. When writing an application that uses **smgbp** to set the bottom margin, both arguments must be given.

If only one of **smglp** and **smgrp** is set, then it is used with two arguments, the column number of the left and right margins, in that order. Likewise, if only one of **smgtp** and **smgbp** is set, then it is used with two arguments that give the top and bottom margins, in that order, counting from the top of the page. Thus when coding a **terminfo** entry for a printer that requires setting both left and right or top and bottom margins simultaneously, only one of **smglp** and **smgrp** or **smgtp** and **smgbp** should be defined; the other should be left blank. When writing an application that uses these string capabilities, the pairs should be first checked to see if each in the pair is set or only one is set, and should then be used accordingly.

In counting lines or columns, line zero is the top line and column zero is the left-most column. A zero value for the second argument with **smgbp** means the bottom line of the page.

All margins can be cleared with **mgc**.

Shadows, Italics, Wide Characters, Superscripts, Subscripts

Five sets of strings describe the capabilities printers have of enhancing printed text.

Enhanced Printing	
sshm	Enter shadow-printing mode
rshm	Exit shadow-printing mode
sitm	Enter italicizing mode
ritm	Exit italicizing mode
swidm	Enter wide character mode
rwidm	Exit wide character mode
ssupm	Enter superscript mode
rsupm	Exit superscript mode
supcs	List of characters available as superscripts
ssubm	Enter subscript mode
rsubm	Exit subscript mode
subcs	List of characters available as subscripts

If a printer requires the **sshm** control sequence before every character to be shadow-printed, the **rshm** string is left blank. Thus programs that find a control sequence in **sshm** but none in **rshm** should use the **sshm** control sequence before every character to be shadow-printed; otherwise, the **sshm** control sequence should be used once before the set of characters to be shadow-printed, followed by **rshm**. The same is also true of each of the **sitm**-**ritm**, **swidm**-**rwidm**, **ssupm**-**rsupm**, and **ssubm**-**rsubm** pairs.

terminfo also has a capability for printing emboldened text (**bold**). While shadow printing and emboldened printing are similar in that they "darken" the text, many printers produce these two types of print in slightly different ways. Generally, emboldened printing is done by overstriking the same character one or more times. Shadow printing likewise usually involves overstriking, but with a slight movement up and/or to the side so that the character is "fatter".

It is assumed that enhanced printing modes are independent modes, so that it would be possible, for instance, to shadow print italicized subscripts.

As mentioned earlier, the amount of motion automatically made after printing a wide character should be given in **widcs**.

If only a subset of the printable ASCII characters can be printed as superscripts or subscripts, they should be listed in **supcs** or **subcs** strings, respectively. If the **ssupm** or **ssubm** strings contain control sequences, but the corresponding **supcs** or **subcs** strings are empty, it is assumed that all printable ASCII characters are available as superscripts or subscripts.

Automatic motion made after printing a superscript or subscript is assumed to be the same as for regular characters. Thus, for example, printing any of the following three examples results in equivalent motion:

Bi $\mathbf{B}_{\mathbf{i}}$ $\mathbf{B}^{\mathbf{i}}$

Note that the existing **msgr** boolean capability describes whether motion control sequences can be used while in "standout mode". This capability is extended to cover the enhanced printing modes added here. **msgr** should be set for those printers that accept any motion control sequences without affecting shadow, italicized, widened, superscript, or subscript printing. Conversely, if **msgr** is not set, a program should end these modes before attempting any motion.

Alternate Character Sets

In addition to allowing you to define line graphics (described in the "Line Graphics" subsection of the "Insert/Delete Character" section above), **terminfo** lets you define alternate character sets. The following capabilities cover printers and terminals with multiple selectable or definable character sets:

Alternate Character Sets	
scs	Select character set *N*
scsd	Start definition of character set *N*, *M* characters
defc	Define character *A*, *B* dots wide, descender *D*
rcsd	End definition of character set *N*
csnm	List of character set names
daisy	Printer has manually changed print-wheels

The **scs**, **rcsd**, and **csnm** strings are used with a single argument, *N*, a number from 0 to 63 that identifies the character set. The **scsd** string is also used with the argument *N* and another, *M*, that gives the number of characters in the set. The **defc** string is used with three arguments: *A* gives the ASCII code representation for the character, *B* gives the width of the character in dots, and *D* is zero or one depending on whether the character is a "descender" or not. The **defc** string is also followed by a string of "image-data" bytes that describe how the character looks (see below).

Character set 0 is the default character set present after the printer has been initialized. Not every printer has 64 character sets, of course; using **scs** with an argument that doesn't select an available character set should cause a null pointer to be returned by **tparm()** (see *tigetflag*(3X)).

If a character set has to be defined before it can be used, the **scsd** control sequence is to be used before defining the character set, and the **rcsd** is to be used after. They should also cause a NULL pointer to be returned by **tparm()** when used with an argument *N* that doesn't apply. If a character set still has to be selected after being defined, the **scs** control sequence should follow the **rcsd** control sequence. By examining the results of using each of the **scs**, **scsd**, and **rcsd** strings with a character set number in a call to **tparm()**, a program can determine which of the three are needed.

Between use of the **scsd** and **rcsd** strings, the **defc** string should be used to define each character. To print any character on printers covered by **terminfo**, the ASCII code is sent to the printer. This is true for characters in an alternate set as well as "normal" characters. Thus the definition of a character includes the ASCII code that represents it. In addition, the width of the character in dots is given, along with an indication of whether the character should descend below the print line (such as the lower case letter "**g**" in most character sets). The width of the character in dots also indicates the number of image-data bytes that will follow the **defc** string. These image-data bytes indicate where in a dot-matrix pattern ink should be applied to "draw" the character; the number of these bytes and their form are defined in the "Dot-Matrix Graphics" section below.

t

It's easiest for the creator of **terminfo** entries to refer to each character set by number; however, these numbers will be meaningless to the application developer. The **csnm** string alleviates this problem by providing names for each number.

When used with a character set number in a call to **tparm()**, the **csnm** string will produce the equivalent name. These names should be used as a reference only. No naming convention is implied, although anyone who creates a **terminfo** entry for a printer should use names consistent with the names found in user documents for the printer. Application developers should allow a user to specify a character set by number (leaving it up to the user to examine the **csnm** string to determine the correct number), or by name, where the application examines the **csnm** string to determine the corresponding character set number.

These capabilities are likely to be used only with dot-matrix printers. If they are not available, the strings should not be defined. For printers that have manually changed print-wheels or font cartridges, the boolean **daisy** is set.

Dot-Matrix Graphics

Dot-matrix printers typically have the capability of reproducing raster graphics images. Three numeric capabilities and three string capabilities help a program draw raster-graphics images independent of the type of dot-matrix printer or the number of pins or dots the printer can handle at one time.

Dot-Matrix Graphics	
npins	Number of pins, *N*, in print-head
spinv	Spacing of pins vertically in pins per inch
spinh	Spacing of dots horizontally in dots per inch
porder	Matches software bits to print-head pins
sbim	Start printing bit image graphics, *B* bits wide
rbim	End printing bit image graphics

The **sbim** string is used with a single argument, *B*, the width of the image in dots.

The model of dot-matrix or raster-graphics that **terminfo** presents is similar to the technique used for most dot-matrix printers: each pass of the printer's print-head is assumed to produce a dot-matrix that is *N* dots high and *B* dots wide. This is typically a wide, squat, rectangle of dots. The height of this rectangle in dots will vary from one printer to the next; this is given in the **npins** numeric capability. The size of the rectangle in fractions of an inch will also vary; it can be deduced from the **spinv** and **spinh** numeric capabilities. With these three values an application can divide a complete raster-graphics image into several horizontal strips, perhaps interpolating to account for different dot spacing vertically and horizontally.

The **sbim** and **rbim** strings start and end a dot-matrix image, respectively. The **sbim** string is used with a single argument that gives the width of the dot-matrix in dots. A sequence of "image-data bytes" are sent to the printer after the **sbim** string and before the **rbim** string. The number of bytes is a integral multiple of the width of the dot-matrix; the multiple and the form of each byte is determined by the **porder** string as described below.

The **porder** string is a comma separated list of pin numbers optionally followed by an numerical offset. The offset, if given, is separated from the list with a semicolon. The position of each pin number in the list corresponds to a bit in an 8-bit data byte. The pins are numbered consecutively from 1 to **npins**, with 1 being the top pin. Note that the term "pin" is used loosely here; "ink-jet" dot-matrix printers don't have pins, but can be considered to have an equivalent method of applying a single dot of ink to paper. The bit positions in **porder** are in groups of 8, with the first position in each group the most significant bit and the last position the least significant bit. An application produces 8-bit bytes in the order of the groups in **porder**.

An application computes the "image-data bytes" from the internal image, mapping vertical dot positions in each print-head pass into 8-bit bytes, using a 1 bit where ink should be applied and 0 where no ink should be applied. This can be reversed (0 bit for ink, 1 bit for no ink) by giving a negative pin number. If a position is skipped in **porder**, a 0 bit is used. If a position has a lower case "**x**" instead of a pin number, a 1 bit is used in the skipped position. For consistency, a lower case "**o**" can be used to represent a 0 filled, skipped bit. There must be a multiple of 8 bit positions used or skipped in **porder**; if not, low-order bits of the last byte are set to 0. The offset, if given, is added to each data byte; the offset can be negative.

Some examples may help clarify the use of the **porder** string. The AT&T 470, AT&T 475 and C.Itoh 8510 printers provide eight pins for graphics. The pins are identified top to bottom by the 8 bits in a byte, from least significant to most. The **porder** strings for these printers would be **8,7,6,5,4,3,2,1**. The AT&T 478 and AT&T 479 printers also provide eight pins for graphics. However, the pins are identified in the reverse order. The **porder** strings for these printers would be **1,2,3,4,5,6,7,8**. The AT&T 5310, AT&T 5320, Digital LA100, and Digital LN03 printers provide six pins for graphics. The pins are identified top to bottom by the decimal values 1, 2, 4, 8, 16 and 32. These correspond to the low six bits in an 8-bit byte, although the decimal values are further offset by the value 63. The **porder** string for these printers would be **,,6,5,4,3,2,1;63**, equivalent to **o,o,6,5,4,3,2,1;63**.

Effect of Changing Printing Resolution

If the control sequences to change the character pitch or the line pitch are used, the pin or dot spacing may change:

Changing the Character/Line Pitches	
cpi	Change character pitch
cpix	If set, **cpi** changes **spinh**
lpi	Change line pitch
lpix	If set, **lpi** changes **spinv**

Programs that use **cpi** or **lpi** should recalculate the dot spacing:

t

Effects of Changing the Character/Line Pitches

Before	*After*
Using **cpi** with **cpix** clear: **spinh'**	**spinh**
Using **cpi** with **cpix** set: **spinh'**	**spinh = spinh' * orhi / orhi'**
Using **lpi** with **lpix** clear: **spinv'**	**spinv**
Using **lpi** with **lpix** set: **spinv'**	**spinv = spinv' * orhi / orhi'**
Using **chr**: **spinh'**	**spinh**
Using **cvr**: **spinv'**	**spinv**

orhi' and **orhi** are the values of the horizontal resolution in steps per inch, before using **cpi** and after using **cpi**, respectively. Likewise, **orvi'** and **orvi** are the values of the vertical resolution in steps per inch, before using **lpi** and after using **lpi**, respectively. Thus, the changes in the dots per inch for dot-matrix graphics follow the changes in steps per inch for printer resolution.

Print Quality

Many dot-matrix printers can alter the dot spacing of printed text to produce **near-letter-quality** printing or **draft-quality** printing. It is important to be able to choose one or the other because the rate of printing generally decreases as the quality improves. Three strings describe these capabilities:

Print Quality	
snlq	Set near-letter quality print
snrmq	Set normal quality print
sdrfq	Set draft quality print

The capabilities are listed in decreasing levels of quality. If a printer doesn't have all three levels, the respective strings should be left blank.

Printing Rate and Buffer Size

Because there is no standard protocol that can be used to keep a program synchronized with a printer, and because modern printers can buffer data before printing it, a program generally cannot determine at any time what has been printed. Two numeric capabilities can help a program estimate what has been printed.

Print Rate/Buffer Size	
cps	Nominal print rate in characters per second
bufsz	Buffer capacity in characters

cps is the nominal or average rate at which the printer prints characters; if this value is not given, the rate should be estimated at one-tenth the prevailing baud rate. **bufsz** is the maximum number of subsequent characters buffered before the guaranteed printing of an earlier character, assuming proper flow control has been used. If this value is not given it is assumed that the printer does not buffer characters, but prints them as they are received.

t

As an example, if a printer has a 1000-character buffer, then sending the letter "**a**" followed by 1000 additional characters is guaranteed to cause the letter "**a**" to print. If the same printer prints at the rate of 100 characters per second, then it should take 10 seconds to print all the characters in the buffer, less if the buffer is not full. By keeping track of the characters sent to a printer, and knowing the print rate and buffer size, a program can synchronize itself with the printer.

Note that most printer manufacturers advertise the maximum print rate, not the nominal print rate. A good way to get a value to put in for **cps** is to generate a few pages of text, count the number of printable characters, and then see how long it takes to print the text.

Applications that use these values should recognize the variability in the print rate. Straight text, in short lines, with no embedded control sequences will probably print at close to the advertised print rate and probably faster than the rate in **cps**. Graphics data with a lot of control sequences, or very long lines of text, will print at well below the advertised rate and below the rate in **cps**. If the application is using **cps** to decide how long it should take a printer to print a block of text, the application should pad the estimate. If the application is using **cps** to decide how much text has already been printed, it should shrink the estimate. The application will thus err in favor of the user, who wants, above all, to see all the output in its correct place.

Selecting a Terminal

If the environment variable **TERMINFO** is defined, any program using Curses checks for a local terminal definition before checking in the standard place. For example, if **TERM** is set to **att4424**, then the compiled terminal definition is found by default in the path

a/att4424

within an implementation-specific directory.

(The "**a**" is copied from the first letter of **att4424** to avoid creation of huge directories.) However, if **TERMINFO** is set to **$HOME/myterms**, Curses first checks

$HOME/myterms/a/att4424

If that fails, it then checks the default path name.

This is useful for developing experimental definitions or when write permission in the implementation-defined default database is not available.

If the **LINES** and **COLUMNS** environment variables are set, or if the program is executing in a window environment, line and column information in the environment will override information read by **terminfo**.

Application Usage

The most effective way to prepare a terminal description is by imitating the description of a similar terminal in **terminfo** and to build up a description gradually, using partial descriptions with a screen-oriented editor, to check that they are correct. To easily test a new terminal description, the environment variable **TERMINFO** can be set to the path name of a directory containing the compiled description, and programs will look there rather than in the **terminfo** database.

Conventions for Device Aliases

Every device must be assigned a name, such as **vt100**. Device names (except the long name) should be chosen using the following conventions. The name should not contain hyphens because hyphens are reserved for use when adding suffixes that indicate special modes.

These special modes may be modes that the hardware can be in, or user preferences. To assign a special mode to a particular device, append a suffix consisting of a hyphen and an indicator of the mode to the device name. For example, the **-w** suffix means **wide mode**; when specified, it allows for a width of 132 columns instead of the standard 80 columns. Therefore, if you want to use a VT100 device set to wide mode, name the device **vt100-w**. Use the following suffixes where possible:

Suffix	Meaning	Example
-w	Wide mode (more than 80 columns)	**5410-w**
-am	With automatic margins (usually default)	**vt100-am**
-nam	Without automatic margins	**vt100-nam**
-*n*	Number of lines on the screen	**2300-40**
-na	No arrow keys (leave them in local)	**c100-na**
-*n***p**	Number of pages of memory	**c100-4p**
-rv	Reverse video	**4415-rv**

Variations of Terminal Definitions

It is implementation-defined how the entries in **terminfo** may be created.

There is more than one way to write a **terminfo** entry. A minimal entry may permit applications to use Curses to operate the terminal. If the entry is enhanced to describe more of the terminal's capabilities, applications can use Curses to invoke those features, and can take advantages of optimizations within Curses and thus operate more efficiently. For most terminals, an optimal **terminfo** entry has already been written.

EXTERNAL INFLUENCES

Environment Variables

CC Specifies a substitute character for a prototype command character. See **cmdch** in the "Miscellaneous" subsection of the "Insert/Delete Line" section.

t

COLUMNS Specifies column information that can override the column information in **terminfo**. See the "Selecting a Terminal" section.

LINES Specifies lines information that can override the lines information in **terminfo**. See the "Selecting a Terminal" section.

TERM Specifies the name of the current terminal. See the "Selecting a Terminal" section.

TERMINFO
Specifies an alternate location for a local terminal definition. If the value in **TERM** is not found in **$TERMINFO/?/*** or if **TERMINFO** is not set, the value is sought in the default location, **/usr/lib/terminfo/?/***. See the "Selecting a Terminal" section.

SEE ALSO

tic(1), untic(1), curses(3X), tgetent(3X), tigetflag(3X), term(4), term(5).

ANSI Standard X3.64-1979.

X/Open System Interface Definitions, Issue 4, Version 2.

NAME

ttys - terminal control database file, for trusted systems

SYNOPSIS

/tcb/files/ttys

DESCRIPTION

The system supports a single terminal control database containing entries for each terminal that can log into the system. Authentication programs use information contained in the terminal control database to determine if login to the terminal is permitted. Additional fields are maintained for informational purposes.

The format of the terminal control database file is identical to other system authentication database files. For more information on the file format, see *authcap*(4). The file consists of keyword field identifiers and values for those fields. The keyword identifiers supported and their use include:

t_devname This field defines the terminal device name for the entry. The terminal device is expected to be contained in the **/dev** directory, therefore this prefix should not be specified. If the terminal entry describes the **/dev/tty1** device, the **t_devname** field should contain **tty1**.

t_uid This field contains the user id of the last user to successfully login using the terminal device.

t_logtime This **time_t** field records the last successful login time to the terminal device.

t_unsuctime This **time_t** field records the last unsuccessful login time to the terminal device.

t_failures This field records the number of consecutive unsuccessful login attempts to the terminal device.

t_maxtries This field specifies the maximum number of consecutive unsuccessful login attempts permitted using the terminal before the terminal is locked. Once the terminal is locked, it must be unlocked by an authorized administrator.

t_lock This flag field indicates whether the terminal device has been administratively locked or not. This field is manipulated by authorized administrators only.

EXAMPLES

The following is an example of a terminal control database entry:

```
console:t_devname=console:\
       :t_uid=reese:t_logtime#675430072:\
       :t_unsuctime#673610809:\
       :t_maxtries#777:\
       :chkent:
```

This entry is for the system console device, **/dev/console**. The most recent successful login session was for the user **reese**. The entry records the system time for the current successful login and the time of the most recent unsuccessful login attempt.

AUTHOR

SecureWare Inc.

FILES

/tcb/files/ttys Terminal control database file

SEE ALSO

login(1), getprtcent(3), authcap(4), default(4)

NAME

ttytype - data base of terminal types by port

SYNOPSIS

`/etc/ttytype`

DESCRIPTION

`ttytype` is a database that identifies the kind of terminal that is attached to each tty port on the system. The file contains one line per port, and each line contains the terminal type (as a name listed in *terminfo*(4)), a space, and the name of the tty device file, less the initial **`/dev/`**. For example, for an HP 2622 terminal on tty02:

```
2622 tty02
```

This information is read by **`tset`** and by **`login`** (for remote logins) to initialize the **`TERM`** variable at login time (see *tset*(1) and *login*(1)).

AUTHOR

`ttytype` was developed by the University of California, Berkeley.

SEE ALSO

login(1), tset(1).

WARNINGS

Some lines are identified simply as **`dialup`** or **`plugboard`**.

NAME

tun - IP network tunnel driver

SYNOPSIS

#include <net/tun.h>

open("/dev/tun*n***", mode);**

DESCRIPTION

When IP packets are written to **/dev/tun***n* or **/dev/tun***n+M*, they will be received by the kernel's IP layer on the network interface **du***n*. When the kernel's IP layer sends packets to the IP interface **du***n*, they will be available for reading on **/dev/tun***n* or **/dev/tun***n+M*.

Instead of having hardware and an associated kernel interface that support network functions, the **tun** driver allows a network interface to be implemented as a user-space process. While talking to the same set of tunnel drivers on the same system, different network interface processes can implement different IP encapsulation methods, such as RFC 877 for use over CCITT X.25-based public data networks, or RFC 1055 SLIP or RFC 1548/1332 PPP for use over dedicated lines and dialup modems.

The **tun** driver provides support for a pair of devices collectively known as an *IP tunnel*. The two devices comprising a tunnel are known as the *inbound* and *outbound* sides, similar to the pairing between **/dev/tty***n* (the inbound terminal) and **/dev/cu***n* (the outbound 'auto-call unit' available on many systems). The outbound side's minor device number is that of the inbound side plus *M*, though they together appear to IP as one interface. If both the inbound and outbound sides of a tunnel device are open, packets received from IP are delivered to only the inbound side. On HP-UX systems, *M* is 64.

If a TCP packet received from IP is part of a telnet, rlogin, or FTP command stream, it will be put in a *fast queue*. All packets in the fast queue are delivered to the user before any packets in the normal queue.

Config

pseudo-device tun[*n*]

Ioctls

A few special **ioctl**s are provided for use on the **/dev/tun*** devices to supply the functionality needed by applications programs to emulate real hardware interfaces. The complete list of supported **ioctl**s is:

TUIOSPTPT Set or clear the **IFF_POINTOPOINT** in the associated network interface.

TUIOSADRMD Set or clear 'address mode', in which packets read are prefaced with four octets containing the destination IP address in network byte order. The third argument is a pointer to an integer containing either a zero or a one, indicating whether 'address mode' should be cleared or set, respectively. If both 'address mode' and 'packed buffer mode' are set, each packet's length will come first, followed by the packet's destination address, followed by the packet itself.

TUIOGADRMD Get the current status of 'address mode'.

TUIOSPKBMD Set or clear 'packed buffer mode' where multiple packets are encoded in single read/write buffers. The third argument is a pointer to an integer containing either a zero or a one, indicating whether 'packed buffer mode' should be cleared or set, respectively. If set (1), each packet is preceded by four octets representing the next packet's length in octets. The following packet will then be aligned to the next multiple of four octets. If cleared (0), packets will be delivered one per read(3) from the tunnel device. If both 'address mode' and 'packed buffer mode' are set, each packet's length will come first, followed by the packet's address, followed by the packet itself.

TUIOGPKBMD Get the current status of 'packed buffer mode'.

TUIOSPKMAX Set the max number of IP frames to send back in a packet buffer read.

TUIOGPKMAX Get the **PKMAX** value.

TUIOSPKPAD Set the number of long word zeroes to put on the front of each packet read in packed buffer mode.

TUIOGPKPAD Get the number of pad words.

TUIOSNAME Set the interface name (may only be invoked by the superuser).

TUIOGNAME Get the interface name.

FIONBIO Set or clear non-blocking mode for I/O operations.

EXAMPLES

```
#include <net/tun.h>
int tun_fd = -1, len;
char *packet;

tun_fd = open("/dev/tun0", O_RDWR);
ioctl(tun_fd, TUIOSNAME, "du");
len = read(tun_fd, packet, size);
write(tun_fd, packet, len);
```

ERRORS

If a packet is delivered to the interface for an address family other than **AF_INET**, **EAFNOSUPPORT** will be returned.

FILES

/dev/tun0 through **/dev/tun***M-1* ‘inbound’ tunnel devices
/dev/tun*M* through **/dev/tun***2*M-1* ‘outbound’ tunnel devices

AUTHOR

tun was developed by the Progressive Systems.

SEE ALSO

ppp.Auth(4), ppp.Devices(4), ppp.Dialers(4), ppp.Filter(4), ppp.Keys(4), ppp.Systems(4), pppd(1), RFC 1548, RFC 1332, RFC 1144, RFC 1055, RFC 877, and (for philosophical comparison only) RFC 1241.

t

NAME

tztab - time zone adjustment table for date(1) and ctime(3C)

DESCRIPTION

The **tztab** file describes the differences between Coordinated Universal Time (UTC) and local time. Several local areas can be represented simultaneously with historical detail.

The file **tztab** consists of one or more time zone adjustment entries. The first line of the entry contains a unique string that may match the value of the TZ string in the user's environment. The format is **tzname***diff***dstzname** where **tzname** is the time zone name or abbreviation, *diff* is the difference in hours from UTC, and **dstzname** is the name or abbreviation of the "Daylight Savings" time zone. Fractional values of *diff* are expressed in minutes preceded by a colon. Each such string will start with an alphabetic character.

The second and subsequent lines of each entry details the time zone adjustments for that time zone. The lines contain seven fields each. The first six fields specify the first minute in which the time zone adjustment, specified in the seventh field, applies. The fields are separated by spaces or tabs. The first six are integer patterns that specify the minute (0-59), hour (0-23), day of the month (1-31), month of the year (1-12), year (1970-2038), and day of the week (0-6, with 0=Sunday). The minute, hour, and month of the year must contain a number in the (respective) range indicated above. The day of the month, year, and day of the week can contain a number as above or two numbers separated by a minus (indicating an inclusive range). Either the day of the month or the day of the week field must be a range, the other must be simple number.

The seventh field is a string that describes the time zone adjustment in its simplest form: **tzname***diff* where **tzname** is an alphabetic string giving the time zone name or abbreviation, and *diff* is the difference in hours from UTC. **tzname** must match either the **tzname** field or the **dstzname** field in the first line of the time zone adjustment entry. Any fractional *diff* is shown in minutes.

Comments begin with a **#** in the *first* column, and include all characters up to a new-line. Comments are ignored.

If the value of the **TZ** string does not match any line in the table, it is interpreted according to the current U.S. pattern.

EXTERNAL INFLUENCES

International Code Set Support

Single-byte character code sets are supported.

EXAMPLES

The time zone adjustment table for the Eastern Time Zone in the United States is:

```
EST5EDT
0 3 6     1  1974      0-6 EDT4
0 3 22-28 2  1975      0   EDT4
0 3 24-30 4  1976-1986 0   EDT4
0 3 1-7   4  1987-2038 0   EDT4
0 1 24-30 11 1974      0   EST5
0 1 25-31 10 1975-2038 0   EST5
```

Normally (as indicated in the first line) Eastern Standard Time is five hours earlier than UTC. During Daylight Savings time, it changes to a 4 hour difference. The first time Daylight Savings Time took effect (second line) was on January 6, 1974 at 3:00 a.m., EDT. Note that the minute before was 1:59 a.m., EST. The change back to standard time took effect (sixth line) on the last Sunday in November of the same year. At that point, the time went from 1:59 a.m. EDT to 1:00 a.m. EST. The transition to Daylight Savings Time since then has gone from the last Sunday in February (third line) to the last Sunday in April (fourth line) to the first Sunday in April (fifth line). The return to standard time for the same period has remained at the last Sunday in October (seventh line).

AUTHOR

tztab was developed by HP.

FILES

/usr/lib/tztab

t

SEE ALSO

date(1), ctime(3C), environ(5).

NAME

ups_conf - HP PowerTrust Uninterruptible Power System (UPS) monitor configuration file

DESCRIPTION

The default configuration file for the HP PowerTrust UPS monitor daemon (**ups_mond**). Another file can be used as long as it complies with the specified format, and the monitor daemon is configured to use the alternate file (see the description of the **-f** option in *ups_mond*(1M)).

Lines in the configuration file can contain a maximum of 256 characters, and the full pathname of a UPS-tty device file can contain a maximum of 100 characters.

Only one entry per line is allowed in the configuration file. Each line begins with a keyword. All fields in configuration file entries are delimited by colons (:). Entries in the configuration file end with the first space encountered (as specified by the library function, **isspace()**; see *isspace*(3C)). Characters beyond the first space on each line are treated as comments.

The shutdown delay and timeout values should be the first lines in the file.

Entries in this file begin with keywords that must appear exactly as shown below. The terminating colon separates the keyword from the value of its parameter. **ups_mond** recognizes the following keywords:

shutdown_delay_mins
: The number of minutes following notification by the first **upstty** that its corresponding UPS is operating on internal (battery) power, before **ups_mond** initiates **shutdown -h**. The default is one minute. This value should be set to account for transient power interruptions, if they are common to the site.

shutdown_timeout_mins
: The number of minutes to monitor the **shutdown -h** operation before initiating **reboot** with the halt option (**RB_HALT**; see *reboot*(2)). The default is five minutes. This value should be longer than the longest time the system requires to execute *shutdown*(1M). Note that after the value of **shutdown_timeout_mins** has elapsed, a UPS lacking AC line voltage will power off. When AC line voltage is restored, the UPS will restore power to its output. This timeout value should not be so much longer than **shutdown** that an observer may become impatient. It is important to note that this value is the period for which the UPS will delay its power-cycling, even if AC power is restored sooner.

upstty
: The full pathname of the **tty** device special file through which the UPS is configured. Include one entry for each UPS. **upstty** entries are handled in the order they appear in **/etc/ups_conf**. Therefore, it is important to list your uninterruptible power systems in order of priority (for example, UPS protecting the SPU listed first).

  upstty entries can contain following optional parameters, which can appear in any order following the **upstty** device special file name:

MSG_ONLY
: This **upstty** will not cause **shutdown** or **reboot** to be initiated.

SOLA
: The type of uninterruptible power supply. It defaults to SOLA.

EXAMPLES

The following is a sample **/etc/ups_conf** file:

```
shutdown_delay_mins:1
shutdown_timeout_mins:5
upstty:/dev/tty0p1
upstty:/dev/tty0p2:MSG_ONLY
upstty:/dev/tty0p3:SOLA:MSG_ONLY
upstty:/dev/tty0p5:SOLA
```

FILES

/dev/tty*
/etc/ups_conf

SEE ALSO

ups_mond(1M)

u

NAME

utmp, wtmp, btmp - utmp, wtmp, btmp entry format

SYNOPSIS

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

DESCRIPTION

These files, which hold user and accounting information for such commands as **last**, **who**, **write**, and **login** (see *last*(1), *who*(1), *write*(1), and *login*(1)), have the following structure as defined by <**utmp.h**>:

```
#define UTMP_FILE   "/etc/utmp"
#define WTMP_FILE   "/var/adm/wtmp"
#define BTMP_FILE   "/var/adm/btmp"
#define ut_name     ut_user

struct  utmp {
  char   ut_user[8];             /* User login name */
  char   ut_id[4];               /* /etc/inittab id(usually line#)*/
  char   ut_line[12]             /* device name (console, lnxx) */
  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*/
  unsigned long ut_addr;         /* host Internet addr, if remote*/
};

/*  Definitions for ut_type  */
#define EMPTY           0
#define RUN_LVL         1
#define BOOT_TIME       2
#define OLD_TIME        3
#define NEW_TIME        4
#define INIT_PROCESS    5      /* Process spawned by "init" */
#define LOGIN_PROCESS   6      /* getty process awaiting login */
#define USER_PROCESS    7      /* A user process */
#define DEAD_PROCESS    8
#define ACCOUNTING      9
#define UTMAXTYPE       ACCOUNTING  /* Max. legal value of ut_type */

/*  Special strings or formats used in the "ut_line" field */
/*  when  accounting for something other than a process  */
/*  No string for the ut_line field can be more than */
/*  11 chars + a NULL in length  */
#define RUNLVL_MSG      "run-level %c"
#define BOOT_MSG        "system boot"
#define OTIME_MSG       "old time"
#define NTIME_MSG       "new time"
```


File **utmp** contains a record of all users logged onto the system. File **btmp** contains bad login entries for each invalid logon attempt. File **wtmp** contains a record of all logins and logouts.

Note that **wtmp** and **btmp** tend to grow without bound, and should be checked regularly. Information that is no longer useful should be removed periodically to prevent it from becoming too large. Also note that **wtmp** and **btmp** are not created by the programs that maintain them. Thus, if these files are removed, record-keeping is turned off.

FILES

/etc/utmp
/var/adm/wtmp
/var/adm/btmp

AUTHOR

utmp, **wtmp**, and **btmp** were developed by HP and the University of California, Berkeley.

SEE ALSO

last(1), login(1), who(1), write(1), acctcon(1M), fwtmp(1M), getut(3C).

STANDARDS CONFORMANCE

<**utmp.h**>: XPG2

NAME

utmpx - utmpx database storage file

SYNOPSIS

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

DESCRIPTION

File **utmpx** contains user accounting information for all users logged onto the system. This file will be used instead of the **utmp** file, which is being depreciated. The following information is stored in the **utmpx** file:

- User login name (up to 24 characters)
- /etc/lines id
- device name (console, lnxx)
- process id
- type of entry
- exit status of a process marked as DEAD_PROCESS
- the time the entry was made
- Internet address of host, if remote

The current version of HP-UX updates both **utmp** and **utmpx** files and formats. Direct use of **utmpx** file is not recommended, because **utmp** and **utmpx** files must be updated simultaneously. This functionality is provided by libc APIs **pututline** and **pututxline**

FILES

/etc/utmpx

AUTHOR

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

SEE ALSO

last(1), login(1), who(1), write(1), acctcon(1M), fwtmp(1M), utmp(4), getut(3C).

STANDARDS CONFORMANCE

<**utmp.h**>: X/OPEN 4.2

u

NAME

uuencode - format of an encoded uuencode file

DESCRIPTION

Files output by **uuencode** consist of a header line followed by a number of body lines, and a trailer line. The **uudecode** command ignores any lines preceding the header or following the trailer (see *uuencode*(1)). Lines preceding a header must not look like a header.

The header line consists of the word **begin** followed by a space, a mode (in octal), another space, and a string which specifies the name of the remote file.

The body consists of a number of lines, each containing 62 or fewer characters (including trailing new-line). These lines consist of a character count, followed by encoded characters, followed by a newline.

The character count is a single printing character, which represents an integer. This integer is the number of bytes in the rest of the line, and always ranges from 0 to 63. The byte count can be determined by subtracting the equivalent octal value of an ASCII space character (octal 40) from the character.

Groups of 3 bytes are stored in 4 characters, 6 bits per character. All are offset by a space to make the characters printable. The last line may be shorter than the normal 45 bytes. If the size is not a multiple of 3, this fact can be determined by the value of the count on the last line. Extra meaningless data will be included, if necessary, to make the character count a multiple of 4. The body is terminated by a line with a count of zero. This line consists of one ASCII space.

The trailer line consists of the word **end** on a line by itself.

SEE ALSO

mail(1), uuencode(1), uucp(1).

NAME

vhe_list - information file for the Virtual Home Environment

DESCRIPTION

/etc/vhe_list is an ASCII file that contains the information needed to configure a group of machines together with the Virtual Home Environment (VHE). These machines are connected using the Network File System (NFS). The information from **vhe_list** is used by the script **vhe_mounter**.

An entry in **vhe_list** contains the following information:

- Host name of a machine exporting a file system.
- Name of the file system to be mounted by NFS.
- Name of the directory that acts as the mount point.
- Mount options for the NFS mount (this is optional).

For every file system that is to be available (exported) for NFS mounting for VHE, there is an entry in the **vhe_list** file. Blank lines, lines of white spaces, or lines beginning with the **#** character are ignored.

EXAMPLES

Consider two machines named **high** and **low**, each to be connected with VHE. Machine **high** is exporting the file system **/** to be mounted on directory **/vhe/high**. Machine **low** is exporting the file system **/** to be mounted on directory **/vhe/low** and the file system **/home** to be mounted on directory **/vhe/low/home** using the NFS mount options of **timeo=10,wsize=4096**. For this situation, the contents of the **vhe_list** file would resemble the following:

```
high     /          /vhe/high
low      /          /vhe/low             timeo=10,wsize=4096
low      /home      /vhe/low/home        timeo=10,wsize=4096
# A comment line
```

Mount options must be separated by commas, and must contain no spaces. Mount options are the same as those used in the **mount** command (see *mount*(1M)).

AUTHOR

vhe_list was developed by HP.

FILES

/etc/vhe_list

SEE ALSO

vhe_altlog(1M), vhe_mounter(1M), vhe_u_mnt(1M).

V

NAME

ypfiles - Network Information Service database and directory structure

DESCRIPTION

Remarks

The Network Information Service (NIS) was formerly known as Yellow Pages (yp). Although the name has changed, the functionality of the service remains the same.

The Network Information Service (NIS) network lookup service uses databases in the directory hierarchy under **/var/yp**. These databases exist only on machines that act as NIS servers. A database consists of two files created by **makedbm** (see *makedbm*(1M)). One has the filename extension **.pag** and the other has the filename extension **.dir**. For example, the database named **netgroup** is implemented by the pair of files **netgroup.pag** and **netgroup.dir**. A database served by the NIS is called an NIS *map*.

An NIS *domain* is a named set of Network Information Service maps. Each NIS domain is implemented as a subdirectory of **/var/yp** (whose name is the domain name) and contains the maps for that domain. Any number of NIS domains can exist, and each can contain any number of maps.

Besides the databases contained in **/var/yp/***domain*, master NIS servers have files named *general_NIS_mapname***.time** that reside there, too. These files are merely empty files whose times of last modification are compared with those of the ASCII files from which the maps are built. The **ypmake** script performs these comparisons to determine whether the maps are current (see *ypmake*(1M)). The *general_NIS_mapname* designation is described further in the FILES section below.

The NIS lookup service does not require maps, although maps may be required for the normal operation of other parts of the system. The list of maps an NIS server provides access to is neither restricted nor must it be all-inclusive. If a map exists in a given domain and a client asks about it, the NIS serves it. For a map to be consistently accessible, it must exist on all NIS servers that serve the domain. To provide data uniformity between the replicated maps, make an entry to run **ypxfr** periodically in root's **crontab** file on each server (see *ypxfr*(1M) and *crontab*(1M)). More information on this topic is in *yppush*(1M) and *ypxfr*(1M).

NIS maps contain two special key-value pairs. The first key, *NIS_LAST_MODIFIED*, has a 10-character (ASCII) order number as a value. The order number is the **time()** in seconds when the map was built (see *time*(2)). The second key is *NIS_MASTER_NAME*, whose value is the host name of the map's master NIS server. The **makedbm** command generates both key-value pairs automatically. The **ypxfr** command uses these values when it transfers a map from one NIS server to another.

Generate and modify NIS maps only on the master server. They are copied to the slaves using **ypxfr** to avoid potential byte-ordering problems among NIS servers running on machines with different architectures, and to minimize the disk space required for the databases (see *ypxfr*(1M)). NIS databases can be created initially for both masters and slaves by using **ypinit** (see *ypinit*(1M)).

After servers' databases are created, the contents of some maps will change. Generally, an ASCII source version of each database exists on the master, and is changed with a text editor. The NIS map is rebuilt to include the changes, and propagated from the master to the slaves by running the **ypmake** shell script (see *ypmake*(1M)).

All standard NIS maps are built by commands contained in the **ypmake** script or the NIS **Makefile.** If you add a non-standard NIS map, edit the **ypmake** script or **Makefile** to support the new map (standard NIS maps are discussed under FILES below). **ypmake** and **Makefile** use **makedbm** to generate the NIS maps on the master and may run **yppush** to copy the rebuilt maps to the slaves (see *yppush*(1M)). The **yppush** command reads the map named **ypservers** that contains the host names of all NIS servers for the specific domain. For more information, see *ypmake*(1M), *yppush*(1M), and *ypxfr*(1M).

DEPENDENCIES

If **/var/yp** is in a file system that does not allow file names longer than 14 characters and you want to create a new non-standard map for the Network Information Service, its name must not exceed 10 characters in length. This rule exists because **makedbm** adds the 4-character suffixes **.dir** and **.pag** to any mapname.

The following table describes the translation of standard NIS mapnames to shorter names for storage on a 14-character filename file system. The standard mapnames should be used by NIS clients on HP machines when making requests, regardless of which machine is the NIS server.

Standard NIS Mapname	Abbreviated Mapname
mail.aliases	mail.alias
mail.byaddr	mail.byad
ethers.byaddr	ether.byad
ethers.byname	ether.byna
group.bygid	group.bygi
group.byname	group.byna
hosts.byaddr	hosts.byad
hosts.byname	hosts.byna
netgroup	netgroup
netgroup.byhost	netgr.byho
netgroup.byuser	netgr.byus
netid.byname	netid.byn
networks.byaddr	netwk.byad
networks.byname	netwk.byna
passwd.byname	passw.byna
passwd.byuid	passw.byui
protocols.byname	proto.byna
protocols.bynumber	proto.bynu
publickey.byname	pbkey.byna
rpc.byname	rpc.byna
rpc.bynumber	rpc.bynu
services.byname	servi.byna
vhe_list	vhe_list
auto.master	auto.mast
ypservers	ypservers

AUTHOR

ypfiles was developed by Sun Microsystems, Inc.

FILES

The following table presents information about the standard Network Information Service maps.

The *General NIS Mapname* column lists names for sets of NIS maps; the sets include adjacent entries from the *Standard NIS Mapname* column.

The *ASCII Source* column lists the ASCII files from which the maps are usually built on HP master NIS servers. The **ypmake** script permits the source directory, or file in the case of the passwd maps, to vary.

The *Standard NIS Mapname* column lists names by which maps are stored on NIS servers and referred to by NIS clients.

General NIS Mapname	ASCII Source	Standard NIS Mapname
aliases	/etc/mail/aliase	mail.aliases mail.byaddr
ethers	*	ethers.byaddr ethers.byname
group	/etc/group	group.byname group.bygid
hosts	/etc/hosts	hosts.byname hosts.byaddr
netgroup	/etc/netgroup	netgroup netgroup.byhost netgroup.byuser
netid	/etc/netid	netid.byname
networks	/etc/networks	network.byaddr network.byname
passwd	/etc/passwd	passwd.byname passwd.byuid
protocols	/etc/protocols	protocols.byname protocols.bynumber
publickey	/etc/publickey	publickey.byname
rpc	/etc/rpc	rpc.byname rcp.bynumber
services	/etc/services	servi.bynp services.byname
automounter	/etc/auto_master	auto.master
vhe_list	/etc/vhe_list	vhe_list **
ypservers	***	ypservers

* These databases are not built on HP master Network Information Service servers. However, if an HP machine is a slave to a master NIS server that creates and distributes these databases, the HP slave NIS server will store these databases. It is suggested that if you have a non-HP machine that requires these maps, make that machine the master NIS server. By doing this, the maps should be built as needed.

** The **vhe_list** map is a map generated only by HP master NIS servers.

*** No ASCII source exists for the **ypservers** database. It is created from responses provided by the user of **ypinit** on the master NIS server, and it has no matching **ypservers.time** file.

SEE ALSO

domainname(1M), makedbm(1M), rpcinfo(1M), vhe_altlog(1M), vhe_mounter(1M), vhe_u_mnt(1M), ypinit(1M), ypmake(1M), yppoll(1M), yppush(1M), ypserv(1M), ypxfr(1M), vhe_list(4).

y