

**559**



HEWLETT  
PACKARD

HP 9000  
Computers

HP-UX Reference  
Release 10.0  
Volume 3 (of 4)

---

## 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.

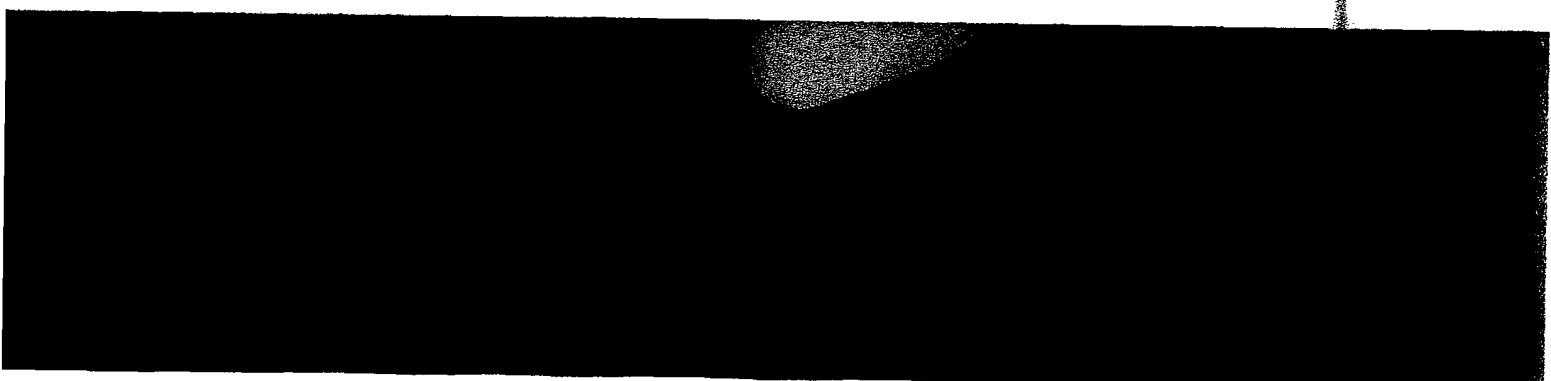
**Restricted Rights Legend.** Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DOD agencies, and subparagraphs (c) (1) and (c) (2) of the Commercial 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.

**Copyright Notices.** ©copyright 1983-95 Hewlett-Packard Company, all rights reserved.

Reproduction, adaptation, or translation of this document without prior written permission is prohibited, except as allowed under the copyright laws.



©copyright 1979, 1980, 1983, 1985-93 Regents of the University of California

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

©copyright 1980, 1984, 1986 Novell, Inc.

©copyright 1986-1992 Sun Microsystems, Inc.

©copyright 1985-86, 1988 Massachusetts Institute of Technology.

©copyright 1989-93 The Open Software Foundation, Inc.

©copyright 1986 Digital Equipment Corporation.

©copyright 1990 Motorola, Inc.

©copyright 1990, 1991, 1992 Cornell University

©copyright 1989-1991 The University of Maryland

©copyright 1988 Carnegie Mellon University

**Trademark Notices** UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

X Window System is a trademark of the Massachusetts Institute of Technology.

MS-DOS and Microsoft are U.S. registered trademarks of Microsoft Corporation.

OSF/Motif is a trademark of the Open Software Foundation, Inc. in the U.S. and other countries.

## 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>*:

- |                  |   |
|------------------|---|
| <b>MS_DATA</b>   | This is ordinarily required. It indicates the presence of the <i>fstype</i> , <i>dataptr</i> , and <i>datalen</i> arguments.<br>(For backward compatibility, if this flag is not set, the <i>fstype</i> is assumed to be that of the root file system, and <i>dataptr</i> and <i>datalen</i> are assumed to be zero.) |
| <b>MS_RDONLY</b> | This is used to control write permission on the mounted file system. If not set, writing is permitted according to individual file accessibility.   |
| <b>MS_NOSUID</b> | This flag disables set-user-ID and set-group-ID behavior on this file system.   |
| <b>MS_QUOTA</b>  | 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>*:

- |                 |  |
|-----------------|--|
| <b>MS_DELAY</b> | 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. |
|-----------------|--|

## RETURN VALUE

mount() r

0 S

-1 F

## ERRORS

If mount()

[EACC

[EBUS

[EBUS

[EBUS

[EFAU

[EINV.

[ELOC

[ENAM

[ENOI

[ENOI

[ENOI

[ENOI

[ENOI

[ENOI

[ENXI

[EPER

[EROF

## WARNINGS

If mount()

the table

/etc/mnt

## SEE ALSO

mount(1M),

- 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**

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\_utimes and tms\_cutimes of the child processes.

tms\_cstime is the sum of the tms\_stimes and tms\_cstimes of the child processes.

**RETURN 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.

**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

truncate, ftruncate

SYNOPSIS

```
#include <u
```

```
int truncate
```

```
int ftruncate
```

DESCRIPTION

The truncate() function shall have a size of was previously sh the file must be o

RETURN VALUES

truncate() a

0 Success

-1 Failure

ERRORS

If truncate()

[EACCES]

[EACCES]

[EDQUOT]

[EFAULT]

[EINVAL]

[EISDIR]

[ELOOP]

[ENAMETOO

[ENOENT]

[ENOTDIR]

[EROFS]

[ETXTBSY]

If ftruncate()

[EBADF]

[EDQUOT]

[EINVAL]

AUTHOR

truncate() w

SEE ALSO

open(2).

STANDARDS CONFC

truncate(): A

ftruncate():.

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.

## NETWORKING FEATURES

### NFS

An additional value for the *type* argument is supported.

#### MOUNT\_NFS

Mount an NFS file system. *data* points to a structure of the following format:

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

struct nfs_args {
    struct sockaddr_in *addr;
    fh_t *fh;
    int flags;
    int wsize;
    int rsize;
    int timeo;
    int retrans;
    char *hostname;
    int acregmin;
    int acregmax;
    int accirmin;
    int accirmax;
};
```

Elements in the structure as follows:

<b>addr</b>	Points to a local socket address structure (see <i>inet(7)</i> ), which is used by the system to communicate with the remote file server.
<b>fh</b>	Points to a structure containing a file handle, an abstract data type that is used by the remote file server when serving an NFS request.
<b>flags</b>	Bit map that sets options and indicates which of the following fields contain valid information. The following values of the bits are defined in <i>&lt;nfs/nfs.h&gt;</i> :
<b>NFSMNT_SOFT</b>	Specify whether the mount is a soft mount or a hard mount. If set, the mount is soft and will cause requests to be retried <i>retrans</i> number of times. Otherwise, the mount is hard and requests will be tried forever.
<b>NFSMNT_WSIZE</b>	Set the write size.
<b>NFSMNT_RSIZE</b>	Set the read size.
<b>NFSMNT_TIMEO</b>	Set the initial timeout value.
<b>NFSMNT_RETRANS</b>	Set the number of request retries.
<b>NFSMNT_HOSTNAME</b>	Set a host name.
<b>NFSMNT_INT</b>	Set the option to have interruptible I/O to the mounted file system.



vfsmount(2)

vfsmount(2)

vfsmount(2)

exclusive.

1 metadata is to be

1 metadata is to be  
for certain applica  
ata loss in case of a

mutually exclusive.

h is used by the sys

ata type that is used

g fields contain valid  
<nfs/nfs.h>:

t mount or a hard  
will cause requests to  
ies. Otherwise, the  
d forever.

) to the mounted file

**NFSMNT\_NODEVS** Set the option to deny access to local devices via NFS device files. By default, access to local devices via NFS device files is allowed.

**NFSMNT\_IGNORE** Mark the file system type as ignore in /etc/mnttab.

**NFSMNT\_NOAC** Turn off attribute caching. By default, NFS caches attributes of files and directories to speed up operations on NFS files by not always getting the attributes from the server. Names are also cached to speed up path name lookup. However it does allow modifications to files on the server to not be immediately detectable on the clients. Setting **NFSMNT\_NOAC** turns off attribute caching and name lookup caching. NFS caches attributes for a length of time proportional to how much time has elapsed since the last modification. The time length is subject to **acregmin**, **acregmax**, **acdirmin**, and **acdirmax**, described below.

**NFSMNT\_NOCTO** Cached attributes are flushed when a NFS file is opened unless this option is specified. This option is useful where it is known that the files will not be changing as is the case for a CD-ROM drive.

**NFSMNT\_ACREGMIN**  
Use the **acregmin** value. See **acregmin** below.

**NFSMNT\_ACDIRMIN**  
Use the **acdirmin** value. See **acdirmin** below.

**NFSMNT\_ACREGMAX**  
Use the **acregmax** value. See **acregmax** below.

**NFSMNT\_ACDIRMAX**  
Use the **acdirmax** value. See **acdirmax** below.

**wsiz** Can be used to advise the system about the maximum number of data bytes to use for a single outgoing protocol (such as UDP) message. This value must be greater than 0. The default is 8192.

**rsiz** Can be used to advise the system about the maximum number of data bytes to use for a single incoming protocol (such as UDP) message. This value must be greater than 0. The default is 8192.

**timeo** Can be used to advise the system on the time to wait between NFS request retries. This is in units of 0.1 seconds. This value must be greater than 0. The default is 7.

**retrans** Can be used to advise the system about the number of times the system will resend a request. This value must be 0 or greater. The default is 4.

**hostname** A name for the file server that can be used when any messages are given concerning the server. The string can contain 0 to 32 characters.

**acregmin** Can be used to advise the system of the minimum number of seconds to cache attributes for a nondirectory file. If this number is less than 0, it means to use the system-defined maximum of 3600 seconds. The number specified can not be 0. If the number is greater than 3600, 3600 will be used. The default is 3. **acregmin** is ignored if **NFSMNT\_NOAC** is specified.

**acdirmin** Can be used to advise the system of the minimum number of seconds to cache attributes for a directory. If this number is less than 0, it means to use the system-defined maximum of 3600 seconds. The number specified can not be 0. If the number is greater than 3600, 3600 will be used. The default is 30. **acdirmin** is ignored if **NFSMNT\_NOAC** is specified.

**acregmax** Can be used to advise the system of the maximum number of seconds to cache attributes for a nondirectory file. If this number is less than 0, it means to use

the system defined maximum of 36000 seconds. The number specified cannot be 0. If the number is greater than 36000, 36000 is used. The default is 60. `acregmax` is ignored if `NFSMNT_NOAC` is specified.

`acdirmax` can be used to advise the system of the maximum number of seconds to cache attributes for a directory. If this number is less than 0, it means to use the system defined maximum of 36000 seconds. The number specified cannot be 0. If the number is greater than 36000, 36000 is used. The default is 60. `acdirmax` is ignored if `NFSMNT_NOAC` is specified.

## 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`, `MOUNT_NFS`, 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.
- [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.
- [EREMOTE] An attempt was made to remotely mount a file system that was already mounted from another remote node.

See *mountd(8)*.

## WARNINGS

The mount mounting operation is performed in `/etc/mnttab`.

## AUTHOR

`vfsmount()`

## SEE ALSO

*mount(1M)*, *nfs*

ount(2)

vfsmount(2)

vfsmount(2)

l cannot be  
ult is 60.

is to cache  
use the sys-  
ot be 0. If  
. acdir-

ck size.  
he file sys-

system as  
file system

ATH\_MAX  
ytes while

r only dur.

ot a direc-  
(indicating

s space  
nted from

June 1995

See *mountd(1M)*, *getfh(2)*, and *inet(7)* 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)*.



waits until a terminator character is seen, or until a time interval specified by the system has passed that is longer than necessary for the number of characters specified.

The data-block-terminator character is included in the data returned to the user, and is included in the byte count. If the number of bytes transferred by the terminal in a block-mode transfer exceeds the number of bytes requested by the user, the read returns the requested number of bytes and the remaining bytes are discarded. The user can determine if data was discarded by checking the last character of the returned data. If the last character is not the terminator character, then more data was received than was requested and data was discarded.

The EIO error can be caused by several events, including errors in transmission, framing, parity, break, and overrun, or if the internal timer expires. The internal timer starts when the second trigger character is sent by the computer, and ends when the terminating character is received by the computer. The length of this timer is determined by the number of bytes requested in the read and the current baud rate, plus an additional ten seconds.

### User Control of Handshaking

If desired, the application program can provide its own handshake mechanism in response to the *alert* character by selecting the **OWNTERM** mode (see **CB\_OWNTERM** below). With this mode selected, the driver completes a read request when the *alert* character is received. No data is discarded before the *alert*, and the *alert* is returned in the data read. The *alert* character may be escaped with a backslash (\) character. The second *trigger* is sent when the application issues the next read.

### blmode Control Calls

First, the standard **open()** call to a tty device must be made to obtain a file descriptor for the subsequent block-mode control calls (an **open()** is done automatically by the system for **stdin** on the terminal).

```
int bfdes;
```

```
bfdes = blopen (int fildes)
```

A call to **blopen()** must be made before any block-mode access is allowed on the specified file descriptor. **blopen()** initializes the block-mode parameters as described below. The return value from **blopen()** is a block-mode file descriptor that must be passed to all subsequent block-mode control calls.

```
int blclose (int bfdes)
```

A call to **blclose()** must be issued before the standard **close()** to ensure proper closure of the device (see **close(2)**). Otherwise unpredictable results can occur. The argument *bfdes* is the file descriptor returned from a previous **blopen()** system call.

```
int blread (int bfdes, char *buf, size_t nbyte)
```

The **blread()** routine has the same parameters as the **read()** system call (see **read(2)**). At the beginning of a read, the **cb\_trig1c** character (if defined) is sent to the device. If **CB\_BMTRANS** is not set, and no **cb\_alertc** character is received, the read data is processed according to **termio(7)**. If **CB\_BMTRANS** is set, or if a non-escaped **cb\_alertc** character is received, echo is turned off for the duration of the transfer, and no further special character processing is done other than that required for the termination character. The argument *bfdes* is the file descriptor returned from a previous **blopen()** system call.

```
int blget (int bfdes, struct blmodeio *arg)
```

A call to **blget()** returns the current values of the **blmodeio** structure (see below). The argument *bfdes* is the file descriptor returned from a previous **blopen()** system call.

```
int blset (int bfdes, const struct blmodeio *arg)
```

A call to **blset()** sets the block-mode values from the structure whose address is *arg*. The argument *bfdes* is the file descriptor returned from a previous **blopen()** system call.

### blmode Structure

The two block-mode control calls, **blget()** and **blset()**, use the following structure, defined in **<sys/blmodeio.h>**:

```
#define NBREPLY 64
struct blmodeio {
    unsigned long    cb_flags;           /* Modes */
    unsigned char    cb_trig1c;         /* First trigger */
    unsigned char    cb_trig2c;         /* Second trigger */
};
```

```
};
```

```
The cb_f
```

```
CB_F
CB_O
```

```
If CB_BM
handshake
handshake
issued with
```

```
If CB_BM
stream, the
cb_trig2
escaped by
```

```
If CB_OW
buffer flush
code to per
normal blo
```

```
The initial
```

```
There are s
ters and t
undefined b
```

```
cb_trig1
```

```
cb_trig2
```

```
cb_alert
```

```
cb_termc
```

```
The cb_r
cb_reply
```

```
The cb_r
second trig
number of
cb_reply
```

### RETURNS

```
If an error
detected, E
pletion.
```

```
During a re
in the user'
library calls
```

```
blopen()
```

```
[ENOT
```

```
blclose()
```

```
[ENOT
```

## mode(3C)

passed that is

d in the byte  
e number of  
ng bytes are  
he returned  
d than was

, break, and  
character is  
he length of  
ate, plus an

e *alert* char-  
i, the driver  
ie *alert*, and  
) character.

e subsequent  
terminal).

specified file  
The return  
l subsequent

oper closure  
nent *bfdes* is

*read(2)*). At  
e device. If  
is processed  
character is  
aracter pro-  
nent *bfdes* is

below). The  
all.

is *arg*. The  
all.

, defined in

/

.0: June 1995

## blmode(3C)

```

    unsigned char    cb_alertc;        /* Alert character */
    unsigned char    cb_termc;        /* Terminating char */
    unsigned char    cb_replen;       /* cb_reply length */
    char             cb_reply[NBREPLY]; /* optional reply */
};

```

The **cb\_flags** field controls the basic block-mode protocol:

```

CB_BMTRANS    0000001    Enable mandatory block-mode transmission.
CB_OWNTerm    0000002    Enable user control of handshake.

```

If **CB\_BMTRANS** is set, all transmissions are processed as block-mode transmissions. The block-mode handshake is not required and data read is processed as block-mode transfer data. The block-mode handshake can still be invoked by receipt of an *alert* character as the first character seen. A *blread()* issued with the **CB\_BMTRANS** bit set causes any existing input buffer data to be flushed.

If **CB\_BMTRANS** is not set, and if the *alert* character is defined and is detected anywhere in the input stream, the input buffer is flushed and the block-mode handshake is invoked. The system then sends the *cb\_trig2c* character to the terminal, and a block-mode transfer follows. The *alert* character can be escaped by preceding it with a backslash (\).

If **CB\_OWNTerm** is set, reads are terminated upon receipt of a non-escaped *alert* character. No input buffer flushing is performed, and the *alert* character is returned in the data read. This allows application code to perform its own block-mode handshaking. If the bit is clear, a non-escaped *alert* character causes normal block-mode handshaking to be used.

The initial **cb\_flags** value is all-bits-cleared.

There are several special characters (both input and output) that are used with block mode. These characters and the initial values for these characters are described below. Any of these characters can be undefined by setting its value to 0377.

**cb\_trig1c** (default DC1) is the initial *trigger* character sent to the terminal at the beginning of a read request.

**cb\_trig2c** (default DC1) is the secondary *trigger* character sent to the terminal after the *alert* character has been seen.

**cb\_alertc** (default DC2) is the *alert* character sent by the terminal in response to the first *trigger* character. It signifies that the terminal is ready to send the data block. The *alert* character can be escaped by preceding it with a backslash ("\).

**cb\_termc** (default RS) is sent by the terminal after the block-mode transfer has completed. It signifies the end of the data block to the computer.

The **cb\_replen** field specifies the length in bytes of the **cb\_reply** field. If set to zero, the **cb\_reply** string is not used. The **cb\_replen** field is initially set to zero.

The **cb\_reply** array contains a string to be sent out after receipt of the *alert* character, but before the second *trigger* character is sent by the computer. Any character can be included in the reply string. The number of characters sent is specified by **cb\_replen**. The initial value of all characters in the **cb\_reply** array is NULL.

### RETURNS

If an error occurs, all calls return a value of -1 and **errno** is set to indicate the error. If no error is detected, *blread()* returns the number of characters read. All other calls return 0 upon successful completion.

During a read, it is possible for the user's buffer to be altered, even if an error value is returned. The data in the user's buffer should be ignored as it is not complete. The following errors can be returned by the library calls indicated:

**blopen()**  
[ENOTTY] The file descriptor specified is not related to a terminal device.

**blclose()**  
[ENOTTY] No previous **blopen** has been issued for the specified file descriptor.

**b**

**hread()**

[EDEADLK] A resource deadlock would occur as a result of this operation (see *lockf(2)*).

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

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

[EIO] An I/O error occurred during block-mode data transmissions.

[ENOTTY] No previous **blopen** has been issued for the specified file descriptor.

**lget()**

[ENOTTY] No previous **blopen** has been issued for the specified file descriptor.

**lset()**

[EINVAL] An illegal value was specified in the structure passed to the system.

[ENOTTY] No previous **blopen** has been issued for the specified file descriptor.

**WARNINGS**

Once **blopen** has been called with a file descriptor and returned successfully, that file descriptor should not subsequently be used as a parameter to the following system calls: **close()**, **dup()**, **dup2()**, **fcntl()**, **ioctl()**, **read()**, or **select()** until a **blclose** is called with the same file descriptor as its parameter. Additionally, **scanf()**, **fscanf()**, **getc()**, **getchar()**, **fgetc()**, and **fgetw()** should not be called for a stream associated with a file descriptor that has been used in a **blopen()** call but has not been used in a **blclose()** call. These functions call **read()**, and calling these routines results in unpredictable behavior.

**AUTHOR**

**blopen()**, **blclose()**, **hread()**, **lget()**, and **lset()** were developed by HP.

**SEE ALSO**

**termio(7)**.

**NAME**

**bsearch()**

**SYNOPSIS**

```
#include <stdlib.h>

void *bsearch(const void *, const void *, rsize_t, rsize_t, int);
```

**DESCRIPTION**

**bsearch** searches into a table of order *acc* table. *base* is the size with two than, equal to, or greater

**NOTES**

The pointer cast to type

The comparison in addition

Although

**RETURN VALUE**

A NULL pointer

**EXAMPLES**

The example table is on

This code its length,

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

```
#define
```

```
struct
```

```
};
```

```
struct
```

```
{
```

fopen(3S)

fpclassify(3M)

fpclassify(3M)

#### NAME

fpclassify(), fpclassifyf() - floating-point operand classification functions

#### SYNOPSIS

```
#include <math.h>
int fpclassify(double x);
int fpclassifyf(float x);
```

#### DESCRIPTION

fpclassify() and fpclassifyf() return a non-negative integer value that specifies the IEEE operand class to which the argument *x* belongs. The value returned is one of the following macros, which are defined in `<math.h>`:

```
#define FP_PLUS_NORM      0 /* Positive normalized */
#define FP_MINUS_NORM     1 /* Negative normalized */
#define FP_PLUS_ZERO      2 /* Positive zero */
#define FP_MINUS_ZERO     3 /* Negative zero */
#define FP_PLUS_INF       4 /* Positive infinity */
#define FP_MINUS_INF      5 /* Negative infinity */
#define FP_PLUS_DENORM    6 /* Positive denormalized */
#define FP_MINUS_DENORM   7 /* Negative denormalized */
#define FP_SNAN           8 /* Signalling NaN */
#define FP_QNAN           9 /* Quiet NaN */
```

Every possible argument value falls into one of these ten categories, so these functions never result in an error.

fpclassifyf() is a float version of fpclassify(); it takes a float argument. To use this function, compile either with the `-Ae` option or with the `-Aa` and `-D_HPUX_SOURCE` options. Otherwise, the compiler promotes the float argument to double, and the function returns incorrect results.

These functions are not specified by any standard. However, they implement the `class()` function suggested in the "Recommended Functions and Predicates" appendix of the IEEE-754 floating-point standard. Also, fpclassifyf() is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

To use these functions, link in the math library by specifying `-lm` or `-LM` on the compiler or linker command line.

#### ERRORS

No errors are defined.

#### SEE ALSO

finite(3M), isinf(3M), isnan(3M).

f

## NAME

getbootpent(), putbootpent(), setbootpent(), endbootpent(), parse\_bp\_hatype(), parse\_bp\_haddr(),  
parse\_bp\_iaddr() - get or put bootptab entry

## SYNOPSIS

```
#include <bootpent.h>

int getbootpent (struct bootpent **bootpent);
int setbootpent (const char *path);
int endbootpent (void);
void putbootpent (
    const struct bootpent *bootpent,
    const int numfields,
    FILE * bootpfile
);
int parse_bp_hatype (const char *source);
int parse_bp_haddr (
    char **source,
    int htype,
    unsigned char *result,
    unsigned int *bytes
);
int parse_bp_iaddr (
    char **source,
    unsigned long *result
);
```

## Remarks

These functions reside in libdc.a, and are linked using the -ldc option to the ld or cc command.

## DESCRIPTION

These functions help a program read or modify a bootptab (bootpd control) file one entry at a time. getbootpent() locates an entry in the /etc/bootptab file, or an alternate file specified to setbootpent(), and returns a pointer to an array of objects of type struct bootpent that breaks the entry into separate data fields with preceding, or embedded, comment (text) lines.

The bootpent structure is defined in <bootpent.h> and includes the following members:

```
int bp_type; /* BP_DATA, BP_COMMENT, BP_BLANK */
char *bp_text; /* one field or one comment line */
```

The file also defines the following data type and constants:

```
typedef struct bootpent *bpp_t;
#define BP_NULLP ((bpp_t) 0)
#define BP_SIZE (sizeof (struct bootpent))
#define MAXHADDRLEN 6
#define HTYPE_UNKNOWN 0 /* 0 bytes */
#define HTYPE_ETHERNET 1 /* 6 bytes */
#define HTYPE_EXP_ETHERNET 2 /* 1 byte */
#define HTYPE_AX25 3 /* 0 bytes */
#define HTYPE_PRONET 4 /* 1 byte */
#define HTYPE_CHAOS 5 /* 0 bytes */
#define HTYPE_IZEE802 6 /* 6 bytes */
#define HTYPE_ARCNET 7 /* 0 bytes */
#define MAXHTYPES 7
```

The fields are described in the "Field Definitions" section below. The purpose of each function is as follows.

getbootpen

setbootpen

endbootpen

putbootpen

parse\_bp\_h

parse\_bp\_h

parse\_bp\_i

**Field Definition**  
If bootpent  
name field or o  
If bootpent  
line from the fi  
tinued with a b



## atbootpent(3X)

## getbootpent(3X)

## getbootpent(3X)

parse\_bp\_haddr(),

getbootpent()

When first called, `getbootpent()` returns a pointer to, and the number of elements in, an array of `bootpent` structures. The array holds the first entry in the `/etc/bootptab` file (or from an alternate file specified by a call to `setbootpent()`), including leading, or embedded, comment lines. Each subsequent call returns a pointer to the next entry in the file so that successive calls can be used to search the entire file.

If no file is currently in memory, `getbootpent()` reads the `/etc/bootptab` file prior to doing its work.

The returned array exists in static space (malloc'd memory) overwritten by the next call (so previously returned pointers become invalid). However, each array element's `bp_text` pointer points to text in an in-memory copy of the file. This text is not altered by the next call (nor by changes to the file itself). Hence, it is possible to copy an entry's array in order to save it, as illustrated in EXAMPLES below. The data remains valid until the next call of `setbootpent()` or `endbootpent()`.

If there are comments after the last entry, they are returned as a separate entry with no data parts.

setbootpent()

Opens the specified file for reading by `getbootpent()`, reads a copy into memory, and closes the file (which as a side-effect releases any locks on the file; see *lockf(2)*). If the given *path* is a null pointer or a null string, `setbootpent()` opens and reads `/etc/bootptab`.

If the last file opened by `setbootpent()` (or implicitly by `getbootpent()`) was `/etc/bootptab`, a subsequent call to `setbootpent()` for the same file rewinds the file to the beginning, making visible any recent changes to the file, without first requiring a call to `endbootpent()`.

endbootpent()

Frees the in-memory copy of the last file opened by `setbootpent()`, or `getbootpent()`.

putbootpent()

Writes (to the current location in the stream specified by *bootpfile*) the ASCII equivalent of the specified array of `bootpent` structures containing one file entry, and its leading, or embedded, comments (a total of *numfields* array elements). Entries are written in canonical form, meaning the entry name and each data field are on separate lines, data fields are preceded by one tab each, and each line except the last ends with `":"`. If *numfields* is less than or equal to zero, nothing is written.

parse\_bp\_htype()

Converts a host address type from string to numeric format (`HTYPE_*`) in the same manner as `bootpd`.

parse\_bp\_haddr()

Converts a host (hardware, link level) address from string to binary format in the same manner as `bootpd` given a host address type (`HTYPE_*`). The calling program's *result*, which must be an array containing at least `MAXHADDRLEN` elements, is modified to hold the host address binary value, and *bytes* is modified to indicate the length in bytes of the resulting address. This can be used to compare two host addresses, independent of string representations. *source* is modified to point to the first char after the parsed address.

parse\_bp\_iaddr()

Converts an internet address from string to binary format in the same manner as `bootpd`. This can be used to compare two internet addresses, independent of string representations. The calling program's *result* is modified to hold the internet address binary value. *source* is modified to point to the first char after the parsed address.

### Field Definitions

If `bootpent.bp_type` is `BP_DATA`, the associated text is one field from the current entry, either the name field or one of the tag fields (two colons in a row) are ignored, not returned.

If `bootpent.bp_type` is `BP_COMMENT` or `BP_BLANK`, the associated text is one comment line or blank line from the file, either preceding the current entry or embedded in it following a data line that was continued with a backslash. The text is exactly as it appears in the file, including any whitespace appearing on

: command.

e entry at a time.  
specified to set-  
at that breaks the

nbars:

function is as fol-

a blank line, and there is no trailing newline.

The returned array elements are in the same order as data fields and comment lines appear in the file.

Entry field strings are of the form:

```
tag[@]="value"
```

with surrounding whitespace, if any, removed (see *bootpd*(1M) for the full description). Double quotes, and backslashes, can appear anywhere in the field strings.

Template entries (those referred to by other entries using *tc* fields) are not special. They can be managed like other entries. It is the calling program's responsibility to correctly manage the order of fields, *tc* fields, and "@" fields that override earlier field values.

## RETURN VALUE

**getbootpent()** returns the number of valid array elements (one or more) upon successful completion. At the end of the input file it returns zero. If it cannot open or close the file it returns -1. If it encounters a memory allocation or map error, or a read error, it returns -2.

**setbootpent()** returns zero if successful opening and reading the specified or default file. If it cannot open or close the file it returns -1. If it encounters a memory allocation or map error or a read error it returns -2.

**endbootpent()** returns zero if successful freeing the memory for the current open file. If there is no current file it returns -1. If it cannot free the memory for the current file it returns -2.

**putbootpent()** returns zero if successful writing an entry to the specified file, with the **ferror()** indication clear (see *ferror*(3S)). Otherwise it returns non-zero with **ferror()** set.

In all cases above, if a failure is due to a failed system call, the **errno** value from the system is valid on return from the called function.

**parse\_bp\_htype()** returns *HTYPE\_UNKNOWN* if the hardware type string is unrecognized.

**parse\_bp\_haddr()** returns zero if successful, otherwise non-zero in case of parsing error, invalid *htype*, or a host address type for which the address length is unknown; *source* is modified to point to the first illegal char (a NUL if the string is too short). The caller's *bytes* value is unmodified, but *result* might be changed.

**parse\_bp\_laddr()** returns zero if successful, otherwise non-zero, and *source* is modified to point to the first illegal char (a NUL if the string is null).

## EXAMPLES

The following code fragment copies all data and comments from */etc/bootptab* to a temporary copy of the file. It converts data entries to canonical form as a side effect, and prints to standard output the first field of each entry copied (should be the field name, assuming the entry doesn't start with a continuation line).

```
#include <bootpent.h>

FILE *newfile; /* to write temp file */
bpp_t bp;      /* read from file */
int field;     /* current field number */
int fields;    /* total in array for one entry */

if ((newfile = fopen ("/tmp/bootptab", "w")) == (FILE *) NULL)
{
    (handle error)
}

while ((fields = getbootpent (&bp)) > 0)
{
    for (field = 0; field < fields; ++field)
    {
        if ((bp[field].bp_type) == BP_DATA)
        {
            (void) puts (bp[field].bp_text);
        }
    }
}
```

```
}
if (
{
}
if (
{
}
if (
{
}
The follow
#inc
#inc
#inc
bpp_
uns:
size
if (
{
}
(voi
```

**WARNINGS**  
These fun  
Calling: s  
**AUTHOR**  
These fun  
**FILES**  
/etc/bc  
**SEE ALSO**  
bootpd(1M)

otpent(3X)

in the file.

ble quotes, and

an be managed  
r of fields, tc

ful completion.  
it encounters a

le. If it cannot  
a read error it

If there is no

e ferror()

em is valid on

d.

error, invalid  
o point to the  
esult might be

to point to the

porary copy of  
utput the first  
a continuation

getbootpent(3X)

```
        break;
    }

    if (putbootpent (bp, fields, newfilep))
    {
        (handle error)
    }

    if (fields < 0)    /* error reading file */
    {
        (handle error)
    }

    if (endbootpent())
    {
        (handle error)
    }

    if (fclose (newfilep))
    {
        (handle error)
    }
}
```

The following code fragment saves a copy of a bootptab entry returned by getbootpent ().

```
#include <malloc.h>
#include <string.h>
#include <bootpent.h>

bpp_t bpnew;
unsigned size;

size = fields *BP_SIZE;

if ((bpnew = (bpp_t) malloc (size)) == BP_NULLP)
{
    (handle error)
}

(void) memcpy ((void *)bpnew, (void *)bp, size);
```

WARNINGS

These functions are unsafe in multi-thread applications.

Calling setbootpent () releases any locks on the file it opens.

AUTHOR

These functions were developed by HP.

FILES

/etc/bootptab control file for bootpd

SEE ALSO

bootpd(1M), errno(2), lockf(2), ferror(3S), fopen(3S), malloc(3C).

getbootpent(3X)

g

getdate(3C)

getdiskbyname(3C)

getdiskbyname(3C)

#### NAME

getdiskbyname(), getdiskbyname\_r() - get disk description by its name

#### SYNOPSIS

```
#include <disktab.h>

struct disktab *getdiskbyname(const char *name);

int getdiskbyname_r(
    const char *name,
    struct disktab *result,
    char *buffer,
    int buflen);
```

#### DESCRIPTION

getdiskbyname() takes a disk name (such as hp7959B) and returns a pointer to a structure that describes its geometry information and the standard disk partition tables. All information is obtained from the disktab database file (see *disktab(4)*).

The contents of the structure `disktab` include the following members. Note that there is not necessarily any correlation between the placement in this list and the order in the structure.

```
char    *d_name;           /* drive name */
char    *d_type;           /* drive type */
int     d_sectsize;        /* sector size in bytes */
int     d_ntracks;        /* # tracks/cylinder */
int     d_nsectors;        /* # sectors/track */
int     d_ncylinders;      /* # cylinders */
int     d_rpm;             /* revolutions/minute */
struct  partition {
    int    p_size;          /* #sectors in partition */
    short  p_bsize;         /* block size in bytes */
    short  p_fsize;         /* frag size in bytes */
} d_partitions[NSECTIONS];
```

The constant `NSECTIONS` is defined in `<disktab.h>`.

#### Reentrant Interfaces

getdiskbyname\_r() expects to be passed three extra parameters:

1. The address of a `struct disktab` where the result will be stored.
2. A buffer to store character strings to which fields in the `struct disktab` will point.
3. The length of the user-supplied buffer.

A buffer length of 100 is recommended. The `struct disktab` is defined in the file `<disktab.h>`. A -1 will be returned if the end-of-file or an error is encountered, or if the supplied buffer is of insufficient length. If the operation is successful, 0 is returned.

#### DIAGNOSTICS

A NULL pointer is returned in case of an error, or if *name* is not found in the disktab database file.

#### WARNINGS

The return value for `getdiskbyname()` points to static data whose content is overwritten by each call. Thus, `getdiskbyname()` is unsafe in multi-thread applications. `getdiskbyname_r()` is MT-Safe and should be used instead.

#### AUTHOR

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

#### SEE ALSO

disktab(4).

g

te the error.

ach call. Thus,  
should be used

## getenv(3C)

## getfsent(3X)

## getfsent(3X)

### NAME

getfsent(), getfsspec(), getfsfile(), getfstype(), setfsent(), endfsent() - get file system descriptor file entry

### SYNOPSIS

```
#include <checklist.h>

struct checklist *getfsent(void);
struct checklist *getfsspec(const char *spec);
struct checklist *getfsfile(const char *file);
struct checklist *getfstype(const char *type);
int setfsent(void);
int endfsent(void);
```

### Remarks:

These routines are included only for compatibility with 4.2 BSD. For maximum portability and improved functionality, new applications should use the *getmntent(3X)* library routines.

### DESCRIPTION

getfsent(), getfsspec(), getfsfile(), and getfstype() each returns a pointer to an object with the following structure containing the broken-out fields of a line in the */etc/fstab* file. The structure is declared in the *<checklist.h>* header file:

```
struct checklist {
    char    *fs_spec;      /* special file name */
    char    *fs_bspec;     /* block special file name */
    char    *fs_dir;       /* file sys directory name */
    char    *fs_type;      /* type: ro, rw, sw, xx */
    int     fs_passno;     /* fsck pass number */
    int     fs_freq;       /* backup frequency */
};
```

The fields have meanings described in *fstab(4)*. If the block special file name, the file system directory name, and the type are not all defined on the associated line in */etc/fstab*, these routines return pointers to NULL in the *fs\_bspec*, *fs\_dir*, and *fs\_type* fields. If the pass number or the backup frequency field are not present on the line, these routines return -1 in the corresponding structure member. *fs\_freq* is reserved for future use.

getfsent()	Reads the next line of the file, opening the file if necessary.
setfsent()	Opens and rewinds the file.
endfsent()	Closes the file.
getfsspec()	Sequentially searches from beginning of file until a matching special file name is found, or until EOF is encountered.
getfsfile()	Sequentially searches from the beginning of the file until a matching file system file name is found, or until EOF is encountered. getfstype() Sequentially searches from the beginning of the file until a matching file system type field is found, or until EOF is encountered.

### DIAGNOSTICS

A null pointer is returned on EOF, invalid entry, or error.

### WARNINGS

Since all information is contained in a static area, it must be copied to be saved.

### AUTHOR

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

### FILES

*/etc/fstab*

### SEE ALSO

*fstab(4)*.

value, and returns  
wise a NULL pointer.  
value, in which case

lls.

le- and/or multi-byte

, ANSI C

# NAME

getmntent(), getmntent\_r(), setmntent(), addmntent(), endmntent(), hasmntopt() - get file system descriptor file entry

# SYNOPSIS

```
#include <mntent.h>

FILE *setmntent(const char *path, char *type);
struct mntent *getmntent(FILE *stream);
int getmntent_r(
    FILE *stream,
    struct mntent *result,
    char *buffer,
    int buflen);

int addmntent(FILE *stream, struct mntent *mnt);
char *hasmntopt(struct mntent *mnt, const char *opt);
int endmntent(FILE *stream);
```

# DESCRIPTION

These routines replace the obsolete `getfsent()` routines (see `getfsent(3X)`) for accessing the file system description file `/etc/fstab`. They are also used to access the mounted file system description file `/etc/mnttab`.

`setmntent()` Opens a file system description file and returns a file pointer which can then be used with `getmntent()`, `addmntent()`, or `endmntent()`. The `type` argument is the same as in `fopen(3C)`.

`getmntent()` Reads the next line from `stream` and returns a pointer to an object with the following structure containing the broken-out fields of a line in the file-system description file, `<mntent.h>`. The fields have meanings described in `fstab(4)`.

```
struct mntent {
    char    *mnt_fstype; /* file system name */
    char    *mnt_dir;    /* file system path prefix */
    char    *mnt_type;   /* hfs, nfs, swap, or xx */
    char    *mnt_opts;   /* ro, suid, etc. */
    int     mnt_freq;    /* dump frequency, in days */
    int     mnt_passno;  /* pass number on parallel fsck */
    long    mnt_time;    /* When file system was mounted; */
                          /* see mnttab(4). */
                          /* (0 for NFS) */
};
```

`getmntent_r()` Uses three extra parameters to provide results equivalent to those produced by `getmntent()`. The extra parameters are:

1. The address of a `struct mntent` where the result will be stored.
2. A buffer to store character strings to which fields in the `struct mntent` will point.
3. The length of the user-supplied buffer. A buffer length of 1025 is recommended.

`addmntent()` Adds the `mntent` structure `mnt` to the end of the open file `stream`. Note that `stream` must be opened for writing.

`hasmntopt()` Scans the `mnt_opts` field of the `mntent` structure `mnt` for a substring that matches `opt`. It returns the address of the substring if a match is found; 0 otherwise.

`endmntent()` Closes the file.

The following definitions are provided in `<mntent.h>`:

```
#define MNT_CHECKLIST    "/etc/fstab"
#define MNT_MNTTAB       "/etc/mnttab"
```

```

#define MNTMAXSTR      128      /* Max size string in mntent */

#define MNTTYPE_HFS     "hfs"    /* HFS file system */
#define MNTTYPE_CDFS    "hfs"    /* CD-ROM file system */
#define MNTTYPE_NFS     "nfs"    /* Network file system */
#define MNTTYPE_SWAP    "swap"   /* Swap device */
#define MNTTYPE_SWAPFS  "swapfs" /* File system swap */
#define MNTTYPE_IGNORE  "ignore" /* Ignore this entry */

#define MNTOPT_DEFAULTS "defaults" /* Use all default options */
#define MNTOPT_RO       "ro"       /* Read only */
#define MNTOPT_RW       "rw"       /* Read/write */
#define MNTOPT_SUID     "suid"     /* Set uid allowed */
#define MNTOPT_NOSUID   "nosuid"   /* No set uid allowed */
#define MNTOPT_QUOTA    "quota"    /* Enable disk quotas */
#define MNTOPT_NOQUOTA  "noquota"  /* Disable disk quotas */

```

The following definition is provided for device swap in <mntent.h>:

```

#define MNTOPT_END      "end"      /* swap after end of file system,
                                   Series 300/400/700 only */

```

The following definitions are provided for file system swap in <mntent.h>:

```

#define MNTOPT_MIN      "min"     /* minimum file system swap */
#define MNTOPT_LIM      "lim"     /* maximum file system swap */
#define MNTOPT_RES      "res"     /* reserve space for file system */
#define MNTOPT_PRI      "pri"     /* file system swap priority */

```

## NETWORKING FEATURES

### NFS

The following definitions are provided in <mntent.h>:

```

#define MNTOPT_BG       "bg"      /* Retry mount in background */
#define MNTOPT_FG       "fg"      /* Retry mount in foreground */
#define MNTOPT_RETRY    "retry"   /* Number of retries allowed */
#define MNTOPT_RSIZE    "rsize"   /* Read buffer size in bytes */
#define MNTOPT_WSIZE    "wsize"   /* Write buffer size in bytes */
#define MNTOPT_TIMEO    "timeo"   /* Timeout in 1/10 seconds */
#define MNTOPT_RETRANS  "retrans" /* Number of retransmissions */
#define MNTOPT_PORT     "port"    /* Server's IP NFS port */
#define MNTOPT_SOFT     "soft"    /* Soft mount */
#define MNTOPT_HARD     "hard"    /* Hard mount */
#define MNTOPT_INTR     "intr"    /* Interruptable hard mounts */
#define MNTOPT_NOINTR   "nointr"  /* Uninterruptable hard mounts */
#define MNTOPT_DEVS     "devs"    /* Device file access allowed */
#define MNTOPT_NODEVS   "nodevs"  /* No device file access allowed */

```

## RETURN VALUE

**setmntent()** Returns a null pointer on error.

**getmntent()** Returns a null pointer on error or EOF. Otherwise, **getmntent()** returns a pointer to a **mntent** structure. Some of the fields comprising a **mntent** structure are optional in **/etc/fstab** and **/etc/mnttab**. In the supplied structure, such missing character pointer fields are set to **NULL** and missing integer fields are set to **-1**.

**getmntent\_r()** Returns a **-1** on error or EOF, or if the supplied buffer is of insufficient length. If the operation is successful, **0** is returned.

**addmntent()** Returns **1** on error.

**endmntent()** Returns **1**.

## WARNING!

The **rel**  
**getmn**  
instead

## AUTHOR

addmn  
develop

## FILES

/etc/  
/etc/

## SEE ALSO

fstab(4)

**mntent(3X)**

ntent \*/

\*/

ons \*/

\*/

system,  
\*/

\*/

\*/  
ystem \*/  
\*/

ound \*/  
ound \*/  
owed \*/  
ytes \*/  
ytes\*/  
ids \*/  
ions \*/  
\*/

ounts \*/  
mounts\*/  
lowed \*/  
allowed \*/

:( ) returns a  
it structure are  
structure, such  
fields are set to

t length. If the

**getmntent(3X)**

#### WARNINGS

The return value for `getmntent()` points to static information that is overwritten in each call. Thus, `getmntent()` is unsafe for multi-thread applications. `getmntent_r()` is MT-Safe and should be used instead.

#### AUTHOR

`addmntent()`, `endmntent()`, `getmntent()`, `hasmntopt()`, and `setmntent()` were developed by The University of California, Berkeley, Sun Microsystems, Inc., and HP.

#### FILES

`/etc/fstab`  
`/etc/mnttab`

#### SEE ALSO

`fstab(4)`, `getfsent(3X)`, `mnttab(4)`.

**getmntent(3X)**

8



getpass(3C)

getprdfent(3)

getprdfent(3)

#### NAME

getprdfent, getprdfnam, setprdfent, endprdfent, putprdfnam - manipulate system default database entry for a trusted system

#### SYNOPSIS

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

struct pr_default *getprdfent(void);
struct pr_default *getprdfnam(const char *name);
void setprdfent(void);
void endprdfent(void);
int putprdfnam(const char *name, struct pr_default *pr);
```

#### DESCRIPTION

*getprdfent* and *getprdfnam* each returns a pointer to an object with the following structure containing the broken-out fields of a line in the system default database. Each line in the database contains a *pr\_default* structure, declared in the *<prot.h>* header file:

```
struct system_default_fields {
    time_t      fd_inactivity_timeout;
    char        fd_boot_authenticate;
};

struct system_default_flags {
    unsigned short
        fg_inactivity_timeout:1,
        fg_boot_authenticate:1,
};

struct pr_default {
    char        dd_name[20];
    char        dg_name;
    struct pr_field prd;
    struct pr_flag prg;
    struct t_field tcd;
    struct t_flag tcg;
    struct dev_field devd;
    struct dev_flag devg;
    struct system_default_fields sfd;
    struct system_default_flags sflg;
};
```

Currently there is only one entry in the system default database, referenced by name **default**.

The System Default database contains default values for all parameters in the Protected Password, Terminal Control, and Device Assignment databases, as well as configurable system-wide parameters. The fields from the other databases are described in the corresponding manual entries. *fd\_inactivity\_timeout* is the number of seconds until a session is terminated on trusted systems.

*fd\_boot\_authenticate* is a boolean flag that indicates whether an authorized user must authenticate before the system begins operation.

*getprdfent* returns a pointer to the first *pr\_default* structure in the database when first called. Thereafter, it returns a pointer to the next *pr\_default* structure in the database, so that successive calls can be used to search the database (not currently supported).

*getprdfnam* searches from the beginning of the file until a default entry matching *name* is found, and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer. Currently, all programs access the default database by calling *getprdfnam* ("default").

g

the standard error  
returned to a null-  
pointer is returned.  
returning.

pected, the size of

A call to *setprdfent* has the effect of rewinding the default control file to allow repeated searches. *endprdfent* can be called to close the database when processing is complete.

*putprdfnam* puts a new or replaced default control entry *pr* with key *name* into the database. If the *prg.fd\_name* field is 0, the requested entry is deleted from the system default database. *putprdfnam* locks the database for all update operations, and performs an *endprdfent* after the update or failed attempt.

#### RETURN VALUE

*getprdfent* and *getprdfnam* return NULL pointers on EOF or error. *putprdfnam* returns 0 if it cannot add or update the entry.

#### WARNINGS

Do not delete the system default entry.

#### AUTHOR

SecureWare Inc.

#### FILES

/tcdb/files/auth/system/default System Defaults database

#### SEE ALSO

authcap(4), default(4), getprpwent(3), getprtcent(3), getdvagent(3).

#### NOTES

The value returned by *getprdfent* and *getprdfnam* refers to a structure that is overwritten by calls to these routines. To retrieve an entry, modify it, and replace it in the database, copy the entry using structure assignment and supply the modified buffer to *putprdfnam*.

Programs using these routines must be compiled with *-lsec*.

#### NAME

getprotoe  
getprotob

#### SYNOPSIS

#includ

struct

int ge

struct

int ge

struct

int ge

int se

int se

int en

int en

#### DESCRIPTIO

The get  
pointer to  
base, /et

The mem

p\_r

p\_a

p\_r

Functions:

get

set

end

get

get

If th

get

ypfil

#### Reentrant

getpro  
the addre  
paramete  
store dat  
file descri

protoent(3N)

function with parameter. If the database key. If finds the file. allocated data

protoent() name\_r() and

it is passed to should not be accessed.

l pointer (0) on

successful or, in returned other-

ea so it must be

oent(), and protoent\_r(), r(), and

getprpwent(3)

getprpwent(3)

NAME

getprpwent, getprpwuid, getprpwnam, setprpwent, endprpwent, putprpwnam - manipulate protected password database entry (for trusted systems only)

SYNOPSIS

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

struct pr_passwd *getprpwent(void);
struct pr_passwd *getprpwuid(int uid);
struct pr_passwd *getprpwnam(const char *name);
struct pr_passwd *getprpwaid(aid_t aid);
void setprpwent(void);
void endprpwent(void);
int putprpwnam(const char *name, struct pr_passwd *pr);
```

DESCRIPTION

getprpwent(), getprpwuid(), getprpwaid(), and getprpwnam() each returns a pointer to a pr\_passwd structure containing the broken-out fields of a line in the protected password database. Each line in the database contains a pr\_passwd structure, declared in the <prot.h> header file:

```
struct pr_field {
    /* Identity: */
    char fd_name[9]; /* uses 8 character maximum (and NULL) from utmp */
    ushort fd_uid; /* uid associated with name above */
    char fd_encrypt[xxx]; /* encrypted password */
    char fd_owner[9]; /* if a pseudo-user, the user accountable */
    char fd_boot_auth; /* boot authorization */
    mask_t fd_auditcntl; /* reserved */
    mask_t audit_reserve1; /* reserved */
    mask_t fd_auditdisp; /* reserved */
    mask_t audit_reserve2; /* reserved */
    aid_t fd_pw_audit; /* audit ID */
    int fd_pw_auditflg; /* audit flag */

    /* Password maintenance parameters: */
    time_t fd_min; /* minimum time between password changes */
    int fd_maxlen; /* maximum length of password */
    time_t fd_expire; /* expiration time duration in secs */
    time_t fd_lifetime; /* account death duration in seconds */
    time_t fd_schange; /* last successful change in secs past 1/1/70 */
    time_t fd_uchange; /* last unsuccessful change */
    time_t fd_acct_expire; /* absolute account lifetime in seconds */
    time_t fd_max_login; /* max time allowed between logins */
    time_t fd_pw_expire_warning; /* password expiration warning */
    ushort fd_pswduser; /* who can change this user's password */
    char fd_pick_pwd; /* can user pick his own passwords? */
    char fd_gen_pwd; /* can user get passwords generated for him? */
    char fd_restrict; /* should generated passwords be restricted? */
    char fd_nullpw; /* is user allowed to have a NULL password? */
    uid_t fd_pwchanger; /* who last changed user's password */
    long fd_pw_admin_num; /* password generation verifier */
    char fd_gen_chars; /* can have password of random ASCII? */
    char fd_gen_letters; /* can have password of random letters? */
    char fd_tod[AUTH_TOD_SIZE]; /* times when user may login */

    /* Login parameters: */
    time_t fd_slogin; /* last successful login */
}
```

```

time_t fd_ulogin; /* last unsuccessful login */
char fd_suctty[14]; /* tty of last successful login */
short fd_nlogins; /* consecutive unsuccessful logins */
char fd_unsuctty[14]; /* tty of last unsuccessful login */
short fd_max_tries; /* maximum unsuc login tries allowed */
char fd_lock; /* Unconditionally lock account? */
);

struct pr_flag {
    unsigned short
        /* Identity: */
        fg_name:1, /* Is fd_name set? */
        fg_uid:1, /* Is fd_uid set? */
        fg_encrypt:1, /* Is fd_encrypt set? */
        fg_owner:1, /* Is fd_owner set? */
        fg_boot_auth:1, /* Is fd_boot_auth set? */
        fg_pw_auid:1, /* Is fd_auid set? */
        fg_pw_auditflg:1, /* Is fd_auditdisp set? */

        /* Password maintenance parameters: */
        fg_min:1, /* Is fd_min set? */
        fg_maxlen:1, /* Is fd_maxlen set? */
        fg_expire:1, /* Is fd_expire set? */
        fg_lifetime:1, /* Is fd_lifetime set? */
        fg_schange:1, /* Is fd_schange set? */
        fg_uchange:1, /* Is fd_fchange set? */
        fg_acct_expire:1, /* Is fd_acct_expire set? */
        fg_max_llogin:1, /* Is fd_max_llogin set? */
        fg_pw_expire_warning:1, /* Is fd_pw_expire_warning set? */
        fg_pswduser:1, /* Is fd_pswduser set? */
        fg_pick_pwd:1, /* Is fd_pick_pwd set? */
        fg_gen_pwd:1, /* Is fd_gen_pwd set? */
        fg_restrict:1, /* Is fd_restrict set? */
        fg_nullpw:1, /* Is fd_nullpw set? */
        fg_pwchanger:1, /* Is fd_pwchanger set? */
        fg_pw_admin_num:1, /* Is fd_pw_admin_num set? */
        fg_gen_chars:1, /* Is fd_gen_chars set? */
        fg_gen_letters:1, /* Is fd_gen_letters set? */
        fg_tod:1, /* Is fd_tod set? */

        /* Login parameters: */
        fg_slogin:1, /* Is fd_slogin set? */
        fg_suctty:1, /* Is fd_suctty set? */
        fg_unsuctty:1, /* Is fd_unsuctty set? */
        fg_ulogin:1, /* Is fd_ulogin set? */
        fg_nlogins:1, /* Is fd_nlogins set? */
        fg_max_tries:1, /* Is fd_max_tries set? */
        fg_lock:1, /* Is fd_lock set? */
};

struct pr_passwd {
    struct pr_field ufld; /* user specific fields */
    struct pr_flag uflg; /* user specific flags */
    struct pr_field sfld; /* system wide fields */
    struct pr_flag sflg; /* system wide flags */
};

```

The protected password database stores user authentication profiles. The `pr_passwd` structure in the user-specific entry refers to parameters specific to a user. The `pr_passwd` structure in the system default database sets parameters that are used when there is no user-specific override.

The user-  
entry for  
encrypted  
multiple

`fd_owner`  
default fi  
authentic

`fd_min` is  
maximum  
password  
account is

`fd_schang`

The `fd_a`  
absolute  
different  
with each

`fd_max` b  
becomes  
warns th  
change p

The next  
password  
use of th  
`fd_gen_le`  
ters and  
user to j  
`fd_gen_p`

`fd_pwcha`  
the accou  
been chos  
dictionary

The `fd_to`  
ing which

The next  
`fd_slogin`  
and `fd_u`  
last login

`fd_nlogin`  
after a s  
sidered l

`fd_lock` is  
(locked) i  
1. if the p  
2. if the n  
3. if the a  
4. if the a  
5. if the t

When ge  
database;  
sive calls  
/etc/pa

`getprpw`  
found an  
like `getp`

The user-specific entry is keyed on the *fd\_name* field, which is a cross reference to the */etc/passwd* entry for the user. The *fd\_uid* field must match the UID in that file as well. The *fd\_encrypt* field is the encrypted password. The password is encrypted in eight character segments, so the size of this field is a multiple of the number of characters in an encrypted segment (*AUTH\_CIPHERTEXT\_SIZE* macro).

*fd\_owner* is the user name accountable for the account. The *fd\_boot\_auth* field is used when the system default file specifies boot authorization is required. *init(1M)* prompts for a user name and password. If the authentication succeeds, a value in this field allows the user to continue the system boot process.

*fd\_min* is the time, in seconds, that must elapse before the user can change passwords. *fd\_maxlen* is the maximum password length (in characters) for the user. *fd\_expire* is the time, in seconds, until the user's password expires. *fd\_lifetime* is the number of seconds that must elapse before the password dies. The account is considered locked if the password is dead.

*fd\_schange* and *fd\_uchange* record the last successful and unsuccessful password change times.

The *fd\_acct\_expire* field specifies the absolute period of time in seconds that the account can be used. An absolute expiration date may be specified, which is then converted into seconds stored in this field. This is different from *fd\_expire* in that *fd\_acct\_expire* specifies an absolute expiration date, while *fd\_expire* is reset with each password change.

*fd\_max\_login* specifies the maximum time in seconds allowed since the last login before the account becomes locked. *fd\_pw\_expire\_warning* is the time in seconds before the end of *fd\_expire* that the system warns the user the password is about to expire. *fd\_pswduser* stores the user ID of the user allowed to change passwords for the account. Typically, this is the account owner.

The next flag fields control password generation. *fd\_pick\_pwd*, if set, allows the user to pick his or her own password. *fd\_nullpw*, if set, allows the account to be used without a password. *fd\_gen\_pwd* enables the use of the random pronounceable password generator for passwords for this account. *fd\_gen\_chars* and *fd\_gen\_letters* allow the password generator to generate passwords composed of random printable characters and random letters, neither of which is easy to remember. The password change software allows the user to pick from whichever options are available for his or her account. One of these three fields (*fd\_gen\_pwd*, *fd\_gen\_chars*, or *fd\_gen\_letters*) must be set.

*fd\_pwchanger* is the user ID of the user who last changed the password on the user's account, if it was not the account owner. *fd\_restrict*, if set, causes triviality checks to be made after the account password has been chosen to avoid palindromes, user name and machine name permutations, and words appearing in the dictionary.

The *fd\_tod* specifier is a string, formatted like the UUCP *Systems* file, which specifies time intervals during which the user can log in.

The next fields are used to protect against login spoofing, listing the time and location of last login. *fd\_slogin* and *fd\_ologin* are time stamps of the last successful and unsuccessful login attempts. *fd\_suctty* and *fd\_unsuctty* are the terminal device or (if supported) host names of the terminal or host from which the last login attempt occurred.

*fd\_nlogins* is the number of unsuccessful login attempts since the last successful login. It is reset to zero after a successful login. *fd\_max\_tries* is the number of unsuccessful attempts until the account is considered locked.

*fd\_lock* indicates whether the administrative lock on the account is set. The account is considered disabled (locked) if one or more of these activities has occurred:

1. if the password is dead,
2. if the maximum number of unsuccessful attempts has been exceeded,
3. if the administrative lock is set,
4. if the account expiration is reached, or
5. if the time since last login is exceeded.

When *getprpwent()* is first called, it returns a pointer to the first user *pr\_passwd* structure in the database; thereafter, it returns a pointer to the next *pr\_passwd* structure in the database so that successive calls can be used to search the database. Note that entries without a corresponding entry in */etc/passwd* are skipped. The entries are scanned in the order they appear in */etc/passwd*.

*getprpwuid()* searches from the beginning of the database until a numerical user ID matching *uid* is found and returns a pointer to the particular structure in which it was found. *getprpwuid()* functions like *getprpwent()* only it uses the audit ID instead of the uid.

structure in the  
in the system

**getprpwnam()** searches from the beginning of the database until a login name matching *name* is found, and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to **setprpwent()** has the effect of rewinding the protected password database to allow repeated searches. **endprpwent()** can be called to close the protected password database when processing is complete.

**putprpwnam()** puts a new or replaced protected password entry *pr* with key *name* into the database. If the *uflag\_name* field is 0, the requested entry is deleted from the protected password database. **putprpwnam()** locks the database for all update operations, and performs a **endprpwent()** after the update or failed attempt.

#### Notes

The value returned by **getprpwent()** and **getprpwnam()** refers to a structure that is overwritten by calls to these routines. To retrieve an entry, modify it, and replace it in the database, copy the entry using structure assignment and supply the modified buffer to **putprpwnam()**.

On systems supporting network connections, the *fd\_succty* and *fd\_unsuccty* fields can be the ASCII representation of the network address of the host from which the last successful or unsuccessful remote login to the account occurred. Use **getdvagnam(3)** to investigate the type of device to determine whether a host or a terminal was used for the last successful or unsuccessful login.

Programs using these routines must be compiled with **-lsec**.

**getprpwent()** assumes one name per UID and one UID per name. The sequential scan loops between the first two instances of a multiple UID.

**getprpwent()** uses **getpwent(3)** routines to sequentially scan databases. User program references to password entries obtained using **getpwent(3)** routines will not be valid after using any routines described here (ie., the \*prp\* routines).

#### RETURN VALUE

**getprpwent()**, **getprpwuid()**, and **getprpwnam()** return NULL pointers on EOF or error. **putprpwnam()** returns 0 if it cannot add or update the entry.

#### AUTHOR

SecureWare Inc.

#### FILES

/etc/passwd	System Password file
/tcb/files/auth/**	Protected Password database
/tcb/files/auth/system/default	System Defaults database

#### SEE ALSO

**authcap(4)**, **getpwent(3)**, **getprdfent(3)**, **prpwd(4)**.

#### NAME

**getprtcnt**  
for a trust

#### SYNOPSIS

```
#include
#include
#include
```

```
struct
```

```
struct
```

```
void se
```

```
void er
```

```
int put
```

#### DESCRIPTION

**getprtcnt**  
broken-ou  
*pr\_term* st

```
struct
```

```
ch
```

```
us
```

```
tin
```

```
us
```

```
tin
```

```
us
```

```
us
```

```
tin
```

```
ch
```

```
us
```

```
};
```

```
struct
```

```
un
```

```
f
```

```
f
```

```
f
```

```
f
```

```
f
```

```
f
```

```
f
```

```
f
```

```
f
```

```
};
```

```
struct
```

```
str
```

```
str
```

```
str
```

```
str
```

```
};
```

The system  
login (*fd\_*  
*fd\_nlogin*  
field is a l  
also be ap  
that the  
seconds fr

pwent(3)

me is found,  
r an error is

low repeated  
ssing is com-

database. If  
rd database.  
t() after the

erwritten by  
e entry using

e the ASCII  
ssful remote  
e whether a

ops between

ferences to  
es described

F or error.

getprtcnt(3)

getprtcnt(3)

NAME

getprtcnt, getprtcnam, setprtcnt, endprtcnt, putprtcnam - manipulate terminal control database entry for a trusted system

SYNOPSIS

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

struct pr_term *getprtcnt(void);
struct pr_term *getprtcnam(const char *name);
void setprtcnt(void);
void endprtcnt(void);
int putprtcnam(const char *name, struct pr_term *pr);
```

DESCRIPTION

getprtcnt and getprtcnam each returns a pointer to an object with the following structure containing the broken-out fields of an entry in the terminal control database. Each entry in the database contains a pr\_term structure, declared in the <prot.h> header file:

```
struct t_field {
    char fd_devname[14]; /* Terminal (or host) name */
    ushort fd_uid; /* uid of last successful login */
    time_t fd_slogin; /* time stamp of successful login */
    ushort fd_uuid; /* uid of last unsuccessful login */
    time_t fd_ulin; /* time stamp of unsuccessful login */
    ushort fd_nlogins; /* consecutive failed attempts */
    ushort fd_max_tries; /* maximum unsuc login tries allowed */
    time_t fd_logdelay; /* delay between login tries */
    char fd_lock; /* terminal locked? */
    ushort fd_login_timeout; /* login timeout in seconds */
};

struct t_flag {
    unsigned short
        fg_devname:1, /* Is fd_devname set? */
        fg_uid:1, /* Is fd_uid set? */
        fg_slogin:1, /* Is fd_stime set? */
        fg_uuid:1, /* Is fd_uuid set? */
        fg_ulin:1, /* Is fd_ftime set? */
        fg_nlogins:1, /* Is fd_nlogins set? */
        fg_max_tries:1, /* Is fd_max_tries set? */
        fg_logdelay:1, /* Is fd_logdelay set? */
        fg_lock:1, /* Is fd_lock set? */
        fg_login_timeout:1; /* is fd_login_timeout valid? */
};

struct pr_term {
    struct t_field ufid;
    struct t_flag uflag;
    struct t_field sfid;
    struct t_flag sflag;
};
```

The system stores the user ID and time of the last successful login (*fd\_uid* and *fd\_slogin*) and unsuccessful login (*fd\_uuid* and *fd\_ulin*) in the appropriate Terminal Control database entry. The system increments *fd\_nlogins* with each unsuccessful login, and resets the field to 0 on a successful login. The *fd\_max\_tries* field is a limit on the number of unsuccessful logins until the account is locked. An administrative lock can also be applied, indicated by a non-zero *fd\_lock* field. *fd\_logdelay* stores the amount of time (in seconds) that the system waits between unsuccessful login attempts, and *fd\_login\_timeout* stores the number of seconds from the beginning of an authentication attempt until the login attempt is terminated.

Note that *ufl*d and *ufl*g refer to user specific entries, and *sfl*d and *sfl*g refer to the system default values (see *authcap*(4)).

The value returned by *getprtcnt* or *getprtcnam* refers to a structure that is overwritten by calls to these routines. To retrieve an entry, modify it, and replace it in the database, copy the entry using structure assignment and supply the modified buffer to *putprtcnam*.

*getprtcnt* returns a pointer to the first terminal *pr\_term* structure in the database when first called. Thereafter, it returns a pointer to the next *pr\_term* structure in the database, so successive calls can be used to search the database. *getprtcnam* searches from the beginning of the database until a terminal name matching *name* is found, and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a **NULL** pointer.

A call to *setprtcnt* has the effect of rewinding the Terminal Control database to allow repeated searches. *endprtcnt* can be called to close the Terminal Control database when processing is complete.

*putprtcnam* puts a new or replaced terminal control entry *pr* with key *name* into the database. If the *fg\_devname* field is 0, the requested entry is deleted from the Terminal Control database. *putprtcnam* locks the database for all update operations, and performs an *endprtcnt* after the update or failed attempt.

#### RETURN VALUE

*getprtcnt* and *getprtcnam* return **NULL** pointers on EOF or error. *putprtcnam* returns 0 if it cannot add or update the entry.

#### AUTHOR

SecureWare Inc.

#### FILES

/tcb/files/ttys	Terminal Control database
/tcb/files/auth/system/default	System Defaults database

#### SEE ALSO

*getprdfent*(3), *authcap*(4), *ttys*(4).

#### NOTES

The *fd\_devname* field, on systems supporting connections, may refer to the ASCII representation of a host name. This can be determined by using *getdvagname*(3) to interrogate the Device Assignment database as to the type of the device, passing in the *fd\_devname* field of the Terminal Control structure as an argument. This allows lockout by machine, instead of the device (typically pseudo tty) on which the session originated.

Programs using these routines must be compiled with **-lsec**.

The *sfl*d and *sfl*g structures are filled from corresponding fields in the system default database. Thus, a program can easily extract the user-specific or system-wide parameters for each database field (see *getprpwent* and *getdvagent*).

#### NAME

*getpw()* -

#### SYNOPSIS

#include

int get

#### DESCRIPTION

*getpw()*

file in whi

*uid* cannot

This routi

for routi

#### NETWORKING

NFS

This routi

Informati

#### RETURN VALUES

*getpw()*

#### WARNINGS

The above

programs

#### AUTHOR

*getpw()*

#### FILES

/etc/passw

#### SEE ALSO

*getpwent*(

#### STANDARDS

*getpw()*



## getrpccent(3C)

## getrpccent(3C)

### NAME

getrpccent(), getrpcbyname(), getrpcbynumber() - get rpc entry

### SYNOPSIS

```
#include <netdb.h>

struct rpccent *getrpccent();
struct rpccent *getrpcbyname(char *name);
struct rpccent *getrpcbynumber(int number);
int setrpccent(int stayopen);
int endrpccent();
```

### DESCRIPTION

getrpccent(), getrpcbyname(), and getrpcbynumber() each return a pointer to an object with the following structure containing the broken-out fields of a line in the rpc program number data base, /etc/rpc.

```
struct rpccent {
    char *r_name;           /* name of server for this rpc program */
    char **r_aliases;       /* NULL terminated list of aliases */
    int r_number;           /* rpc program number for this service */
};
```

### Functions

getrpccent()	Read the next line of the file, opening the file if necessary.
setrpccent()	Open and rewind the file. If the <i>stayopen</i> flag is non-zero, the rpc database is not closed after each call to getrpccent() (either directly or indirectly through one of the other getrpc*() calls).
endrpccent()	Close the file.
getrpcbyname()	Sequentially search from the beginning of the file until a matching rpc program name is found, or until EOF is encountered.
getrpcbynumber()	Sequentially search from the beginning of the file until a matching rpc program number is found, or until EOF is encountered.

### RETURN VALUE

getrpccent(), getrpcbyname(), and getrpcbynumber() return a null pointer (0) on EOF or when unable to access the information in /etc/rpc either directly or through a Network Information Service database.

### WARNINGS

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

### AUTHOR

getrpccent() was developed by Sun Microsystems, Inc.

### FILES

/etc/rpc

### SEE ALSO

rpcinfo(1M), rpc(4).

ent(3C)

ed systems

ntry. Each

sequent calls  
search all  
entries from

xpire, or  
unentered in  
erno is set

spent() is  
mechanism is  
sed to indi-

tspent(),  
rd database  
any way by

onverted to  
tprpwent(3)

## getspwent(3X)

## getspwent(3X)

### NAME

getspwent(), getspwent\_r(), getspwuid(), getspwuid\_r(), getspwaid(), getspwaid\_r(), getspwnam(), getspwnam\_r(), setspwent(), setspwent\_r(), endspwent(), endspwent\_r(), fgetspwent(), fgetspwent\_r() - get secure password file entry, on trusted systems

### SYNOPSIS

```
#include <pwd.h>

struct s_passwd *getspwent(void);

int getspwent_r(struct s_passwd *result, char *buffer, int buflen,
FILE **pwfp);

struct s_passwd *getspwuid(uid_t uid);

int getspwuid_r(uid_t uid, struct s_passwd *result,
char *buffer, int buflen);

struct s_passwd *getspwaid(aid_t aid);

int getspwaid_r(aid_t aid, struct s_passwd *result,
char *buffer, int buflen);

struct s_passwd *getspwnam(const char *name);

int getspwnam_r(char *name, struct s_passwd *result,
char *buffer, int buflen);

void setspwent(void);

void setspwent_r(FILE **pwfp);

void endspwent(void);

void endspwent_r(FILE **pwfp);

struct s_passwd *fgetspwent(FILE *stream);

int fgetspwent_r(FILE *f, struct s_passwd *result,
char *buffer, int buflen);
```

### DESCRIPTION

These privileged routines provide access to the protected password database in a manner similar to the way *getpwent(3C)* routines handle the regular password file, (*/etc/passwd*).

These routines are particularly useful in situations where it is not necessary to get information from the regular password file. *getspwent(3X)* can be used on a trusted system to return the password, audit ID, and audit flag information. Programs using these routines must be linked with the security library, *libsec*.

Note that *getspwent()* routines are provided for backward compatibility. New applications accessing the protected password database on trusted systems should use the *getprpwent()* routines. See *getprpwent(3)*.

*getspwent()*, *getspwuid()*, *getspwaid()*, and *getspwnam()* each returns a pointer to an object of *s\_passwd* structure. The *s\_passwd* structure is maintained for compatibility with existing software and consists of five fields as follows:

```
struct s_passwd {
    char *pw_name;      /* login name */
    char *pw_passwd;    /* encrypted password */
    char *pw_age;       /* password age */
    int pw_auid;        /* audit ID */
    int pw_audflg;      /* audit flag 1=on, 0=off */
};
```

Since the *s\_passwd* structure is declared in the *<pwd.h>* header file, it is unnecessary to redeclare it.

To access other fields in the protected password database that are not included in the *s\_passwd* structure, use *getprpwent()*. See *getprpwent(3)* for more information.

*getspwent()* When first called, *getspwent()* returns a pointer to each *s\_passwd* structure obtained from the protected password database for each user in sequence.

Subsequent calls can be used to search the entire database.

**getspwuid()** Searches for an entry that matches the specified *uid*. It then returns a pointer to the particular structure in which *uid* is found.

**getspwaid()** Similarly searches for a numerical audit ID matching *aid* and returns a pointer to the particular structure in which *aid* is found (see *spasswd*(4) for details on this field).

**getspwnam()** Searches for an entry that matches the specified *name*. Returns a pointer to the particular structure in which *name* is found.

**setspwent()** Resets the protected password database pointer to the beginning of the file to allow repeated searches.

**endspwent()** Should be called to close the protected password database file when processing is complete.

**fgetspwent()** Is no longer supported. It is provided for those applications that did not use */.secure/etc/passwd*.

**Reentrant Interfaces**

**getspwuid\_r()**, **getspwaid\_r()**, **getspwnam\_r()**, and **fgetspwent\_r()** expect to be passed three extra parameters:

1. The address of a *s\_passwd* structure where the result will be stored;
2. A buffer to store character strings (such as the password) to which fields in the *s\_passwd* structure will point;
3. The length of the user-supplied buffer.

In addition to the above three parameters, **getspwent\_r()** requires a pointer to a (*FILE \**) variable. **setspwent\_r()** and **endspwent\_r()** are to be used only in conjunction with **getspwent\_r()** and take the same pointer to a (*FILE \**) variable as a parameter. **setspwent\_r()** can be used to rewind or open the protected password database. **endspwent\_r()** should be called when done to close the file.

The */.secure/etc/passwd* file is no longer supported and these routines provide an interface to the protected password database.

**fgetspwent\_r()** is no longer supported, but is included for those users that did not use the */.secure/etc/passwd* file.

Note that the (*FILE \**) variable must be initialized to NULL before it is passed to **getspwent\_r()** or **setspwent\_r()** for the first time. Thereafter it should not be modified in any way.

A buffer length of 1024 is recommended.

**RETURN VALUE**

**getspwent()** returns a NULL pointer if any of its routines encounters an end-of-file or error while searching, or if the effective user ID of the calling process is not zero.

**getspwent\_r()** returns a -1 if any of its routines encounters an end-of-file or error, or if the supplied buffer has insufficient length. If the operation is successful, 0 is returned.

**WARNINGS**

The above routines use *<stdio.h>*, which causes them to increase the size of programs by more than might otherwise be expected.

Since all information for **getspwent()**, **getspwuid()**, **getspwaid()**, **getspwnam()**, **setspwent()**, **endspwent()**, and **fgetspwent()** is contained in a static area, it must be copied to be saved.

**getspwent()**, **getspwuid()**, **getspwaid()**, **getspwnam()**, **setspwent()**, **endspwent()**, and **fgetspwent()** are unsafe in multi-thread applications. **getspwent\_r()**, **getspwuid\_r()**, **getspwaid\_r()**, **getspwnam\_r()**, **setspwent\_r()**, **endspwent\_r()**, and **fgetspwent\_r()** are MT-Safe and should be used instead.

Network Information Service is not supported on trusted systems.

**EXAMPLE**

The fol

i:  
s:  
c:  
F

s:  
w

e:

**AUTHOR**

getsp

**FILES**

/tcb/

**SEE ALSO**

ypcat(1

spwent(3X)

getspwent(3X)

getspwent(3X)

#### EXAMPLE

The following code excerpt counts the number of entries in the protected password database:

```
int count = 0;
struct s_passwd pwbuf;
char buffer[1024];
FILE *pwf = NULL;

setspwent_r(&pwf);
while (getspwent_r(&pwbuf, buffer, 1024, &pwf) != -1)
    count++;
endspwent_r(&pwf);
```

#### AUTHOR

getspwent() was developed by HP.

#### FILES

/tcb/files/auth/\*/\* Protected Password database

#### SEE ALSO

ypcat(1), getgrent(3C), getlogin(3C), getpwent(3C), getprpwent(3), putspwent(3X), passwd(4).

pointer to the

pointer to the  
this field).

ter to the par-

ie file to allow

essing is com-

did not use

e passed three

swd structure

E \*) variable.  
\_r() and take  
wind or open  
le.

terface to the

not use the

went\_r() or

while search-

the supplied

y more than

spwnam(),  
be copied to

spwent(),  
wuid\_r(),  
pwent\_r()

\_lock(3I)

io\_on\_interrupt(3I)

Series 800 Only

io\_on\_interrupt(3I)

file. Ensur-

## NAME

io\_on\_interrupt() - device interrupt (fault) control

## SYNOPSIS

```
#include <dvio.h>

int (*io_on_interrupt(
    int eid,
    struct interrupt_struct *causevec,
    int (*handler)(int, struct interrupt_struct *)
))(int, struct interrupt_struct *);
```

## DESCRIPTION

*eid* is an entity identifier of an open HP-IB raw bus, Centronics-compatible parallel interface, or GPIO device file, obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call.

*causevec* is a pointer to a structure of the form:

```
struct interrupt_struct {
    integer    cause;
    integer    mask;
};
```

The `interrupt_struct` structure is defined in the file `dvio.h`.

*cause* is a bit vector specifying which of the interrupt or fault events can cause the handler routine to be invoked. The interrupt causes are often specific to the type of interface being considered. Also, certain exception (error) conditions can be handled using the `io_on_interrupt()` capability. Specifying a zero valued *cause* vector effectively turns off the interrupt for that *eid*.

The *mask* parameter is used when an HP-IB parallel poll interrupt is being defined. *mask* is an integer that specifies which parallel poll response lines are of interest. The value of *mask* is viewed as an 8-bit binary number where the least significant bit corresponds to line DIO1; the most significant bit to line DIO8. For example, to activate an interrupt handler when a response occurs on lines 2 or 6, the correct binary number is 00100010. Thus a hexadecimal value of 22 is the correct argument value for *mask*.

When an enabled interrupt condition on the specified *eid* occurs, the receiving process executes the interrupt-handler function pointed to by *handler*. The entity identifier *eid* and the interrupt condition *cause* are returned as the first and second parameters, respectively.

When an interrupt that is to be caught occurs during a `read()`, `write()`, `open()`, or `ioctl()` system call on a slow device such as a terminal (but not a file), during a `pause()` system call, a `sig-pause()` system call, or a `wait()` system call that does not return immediately due to the existence of a previously stopped or zombie process, the interrupt handling function is executed and the interrupted system call returns -1 to the calling process with `errno` set to `EINTR`.

Interrupt *handlers* are not inherited across a `fork()`. *eids* for the same device file produced by `dup()` share the same *handler*.

An interrupt for a given *eid* is implicitly disabled after the occurrence of the event. The interrupt condition can be re-enabled by using `io_interrupt_ctl()` (see `io_interrupt_ctl(3I)`).

When an event specified by *cause* occurs, the receiving process executes the interrupt *handler* function pointed to by *handler*. When the *handler* returns, the user process resumes at the execution point where the event occurred.

Two parameters are passed to *handler*: the *eid* associated with the event, and a pointer to a *causevec* structure. The cause of the interrupt can be determined by the value returned in the *cause* field of the *causevec* structure (more than 1 bit can be set, indicating that more than 1 interrupting condition has occurred). If the interrupt *handler* was invoked due to a parallel poll interrupt, the *mask* field of the *causevec* structure contains the parallel poll response byte.

## HP-IB Interrupts

This section describes interrupt causes specific to an HP-IB device. For an HP-IB device, the cause is a bit vector which is used as follows. To enable a given event, the appropriate bit (in *cause*), shown below, must be set to 1:

SRQ	SRQ and active controller
TLK	Talker addressed
LTN	Listener addressed
TCT	Controller in charge
IFC	IFC has been asserted
REN	Remote enable
DCL	Device clear
GET	Group execution trigger
PPOLL	Parallel poll

**GPIO Interrupts**

This section describes interrupt causes specific to a GPIO device. For a GPIO device, *cause* is a bit vector which is used as follows. To enable a given event, the appropriate bit (in *cause*), shown below, must be set to 1:

EIR	External interrupt
SIE0	Status line 0
SIE1	Status line 1

**Parallel Interrupts**

This section describes interrupt causes specific to a Centronics-compatible parallel device. For a Centronics-compatible parallel device, *cause* is a bit vector which is used as follows. To enable a given event, the appropriate bit (in *cause*), shown below, must be set to 1:

NERROR	Nerror interrupt
SELECT	Select interrupt
PE	Paper error interrupt

**RETURN VALUE**

`io_on_interrupt()` returns a pointer to the previous *handler* if the new *handler* is successfully installed; otherwise it returns a -1 and sets `errno` to indicate the error.

**ERRORS**

`io_on_interrupt()` fails for any of the following reasons and sets `errno` to the value indicated:

[EACCES]	The interface associated with this <i>eid</i> is locked by another process and <code>O_NDELAY</code> is set for this <i>eid</i> (see <code>io_lock(3I)</code> ).
[EBADF]	<i>eid</i> does not refer to an open file.
[ENOTTY]	<i>eid</i> does not refer to a GPIO, Centronics-compatible parallel, or a raw HP-IB device file.
[EFAULT]	<i>handler</i> points to an illegal address. The reliable detection of this error is implementation dependent.
[EFAULT]	<i>causevec</i> points to an illegal address. The reliable detection of this error is implementation dependent.

**DEPENDENCIES**

For the HP 27114 API interface, only the EIR interrupt is available.

**AUTHOR**

`io_on_interrupt()` was developed by HP.

**SEE ALSO**

`dup(2)`, `creat(2)`, `fcntl(2)`, `open(2)`, `pause(2)`, `sigpause(2)`, `io_interrupt_ctl(3I)`.

**NAME**

`io_reset`

**SYNOPSIS**

`#include`

`int i`

**DESCRIPT**

`io_re`

pheral

HP-IB,

`fcntl`

`io_re`

interfac

**RETURN V**

`io_re`

**ERRORS**

`io_re`

cated:

[E]

[E]

[E]

[E/

**AUTHOR**

`io_re`

cale(3C)

ng usage is

next call to  
subsequent

is not port-

to getlo-  
call.

e\_r() and

ster set to  
the addition  
e by LANG.  
MESSAGES.  
variables are

atalogs for

ale() has

source>

f the compile  
red libraries

ped by OSF

langinfo(3C),  
), strtod(3C),  
X), wstol(3X),

0: June 1995

shl\_load(3X)

shl\_load(3X)

NAME

shl\_load(), shl\_definesym(), shl\_findsym(), shl\_gethandle(), shl\_getsymbols(), shl\_unload(), shl\_get(),  
shl\_gethandle\_r(), shl\_get\_r() - explicit load of shared libraries

SYNOPSIS

```
#include <dl.h>

shl_t shl_load(const char *path, int flags, long address);

int shl_findsym(
    shl_t *handle,
    const char *sym,
    short type,
    void *value
);

int shl_definesym(
    const char *sym,
    short type,
    long value,
    int flags
);

int shl_getsymbols(
    shl_t handle,
    short type,
    int flags,
    void *(*memory) (),
    struct shl_symbol **symbols,
);

int shl_unload(shl_t handle);

int shl_get(int index, struct shl_descriptor **desc);

int shl_gethandle(shl_t handle, struct shl_descriptor **desc);

int shl_get_r(int index, struct shl_descriptor *desc);

int shl_gethandle_r(shl_t handle, struct shl_descriptor *desc);
```

DESCRIPTION

These routines can be used to programmatically load and unload shared libraries, and to obtain information about the libraries (such as the addresses of symbols defined within them). The routines themselves are accessed by specifying the -ldld option on the command line with the cc or ld command (see cc(1) and ld(1)). In addition, the -E option to the ld command can be used to ensure that all symbols defined in the program are available to the loaded libraries.

Shared libraries are created by compiling source files with the +z or +Z (position-independent code) options, and linking the resultant object files with the -b (create shared library) option.

**shl\_load()** Attaches the shared library named by *path* or the shared library name that is constructed by using the path part of *path* plus the shared library basename followed by the suffix .0 (e.g. /usr/lib/libname.0) to the process, along with all its dependent libraries. A .0 version is looked for first for those shared libraries that do not have internal names. See ld(1). The library is mapped at the specified *address*. If *address* is 0L, the system chooses an appropriate address for the library. This is the recommended practice because the system has the most complete knowledge of the address space; currently, the *address* field is ignored, and assumed to be 0L. The flags argument is made up of several fields. One of the following must be specified:

- BIND\_IMMEDIATE** Resolve symbol references when the library is loaded.
- BIND\_DEFERRED** Delay code symbol resolution until actual reference.

Zero or more of the following can be specified by doing a bitwise OR operation:

- BIND\_FIRST** Place the library at the head of the symbol search order. In default mode, the library and its dependent libraries

S

	are bound independently of each other (see <code>BIND_TOGETHER</code> ).
<code>BIND_NONFATAL</code>	Default <code>BIND_IMMEDIATE</code> behavior is to treat all unsatisfied symbols as fatal. This flag allows binding of unsatisfied code symbols to be deferred until use.
<code>BIND_NOSTART</code>	Do not call the initializers for the shared library when the library is loaded, nor on a future call to <code>shl_unload()</code> ; by default, all the initializers registered with the specified library are invoked upon loading.
<code>BIND_VERBOSE</code>	Print verbose messages concerning possible unsatisfied symbols.
<code>BIND_RESTRICTED</code>	Restrict symbols visible to the library to those present at the time the library is loaded.
<code>DYNAMIC_PATH</code>	Allow the loader to dynamically search for the library specified by the <i>path</i> argument. The directories to be searched are determined by the <code>+s</code> and <code>+b</code> options of the <code>ld</code> command used when the program was linked.
<code>BIND_TOGETHER</code>	When used with <code>BIND_FIRST</code> , the library being mapped and its dependent libraries will be bound together. This is the default behavior for all <code>shl_load()</code> requests not using <code>BIND_FIRST</code> .

If successful, `shl_load()` returns a handle which can be used in subsequent calls to `shl_findsym()`, `shl_unload()`, `shl_gethandle()`, or `shl_gethandle_r()`; otherwise NULL is returned.

`shl_findsym()` Obtains the address of an exported symbol *sym* from a shared library. The *handle* argument should be a pointer to the handle of a loaded shared library that was returned from a previous call to `shl_load()` or `shl_get()`. If a pointer to NULL is passed for this argument, `shl_findsym()` searches all currently loaded shared libraries and the program to find the symbol; otherwise `shl_findsym()` searches only the specified shared library. The return value of *handle* will be NULL if the symbol found was generated via `shl_definesym()`. Otherwise the handle of the library where the symbol was found is returned. The special handle `PROG_HANDLE` can be used to refer to the program itself, so that symbols exported from the program can also be accessed dynamically. The *type* argument specifies the expected type for the symbol, and should be one of the defined constants `TYPE_PROCEDURE`, `TYPE_DATA`, `TYPE_STORAGE`, or `TYPE_UNDEFINED`. The latter value suppresses type checking. The address of the symbol is returned in the variable pointed to by *value*. If a shared library contains multiple versions of the requested symbol, the latest version is returned. This routine returns 0 if successful; otherwise -1 is returned.

`shl_definesym()`

Adds a symbol to the shared library symbol table for the current process making it the most visible definition. If the *value* falls in the range of a currently loaded library, an association will be made and the symbol is undefined once the associated library is unloaded. The defined symbol can be overridden by a subsequent call to this routine or by loading a more visible library that provides a definition. Symbols overridden in this manner may become visible again if the overriding definition is removed.

Possible symbol types include:

<code>TYPE_PROCEDURE</code>	Symbol is a function.
<code>TYPE_DATA</code>	Symbol is data.

Possible flag values include: None defined at the present time. Zero should be passed in to prevent conflicts with future uses of this flag.



## shl\_load(3X)

## shl\_load(3X)

## shl\_load(3X)

each other (see

or is to treat all  
; allows binding of  
until use.

ared library when  
future call to  
the initializers  
are invoked upon

ossible unsatisfied

to those present at

ch for the library  
e directories to be  
and +b options of  
am was linked.

the library being  
es will be bound  
behavior for all  
D\_FIRST.

in subsequent calls  
andle(), or

brary. The *handle*  
d library that was  
) . If a pointer to  
all currently loaded  
*shl\_findsym()*  
dle will be NULL if  
rwise the *handle* of  
he special *handle*  
t symbols exported  
ument specifies the  
defined constants  
UNDEFINED. The  
l is returned in the  
ple versions of the  
arns 0 if successful;

rocess making it the  
y loaded library, an  
associated library is  
t call to this routine  
mbols overridden in  
s removed.

ro should be passed

### shl\_getsymbols()

Provides an array of symbol records, allocated using the supplied memory allocator, that are associated with the library specified by *handle*. If the *handle* argument is a pointer to NULL, symbols defined using *shl\_definesym()* are returned. If multiple versions of the same symbol have been defined within a library or with *shl\_definesym()*, only the version from the specified symbol information source that would be considered for symbol binding is returned. The *type* argument is used to restrict the return information to a specific type. Values of *TYPE\_PROCEDURE*, *TYPE\_DATA*, and *TYPE\_STORAGE* can be used to limit the returned symbols to be either code, data, or storage respectively; the *TYPE\_DATA* value causes both data and storage symbols to be returned. The constant *TYPE\_UNDEFINED* can be used to return all symbols, regardless of type. The *flags* argument must have one of the following values:

#### IMPORT\_SYMBOLS

Return symbols found on the import list.

#### EXPORT\_SYMBOLS

Return symbols found on the export list. All symbols defined by *shl\_definesym()* are export symbols.

#### INITIALIZERS

Return symbols that are specified as the initializers of the library.

Zero or more of the following can be specified by doing a bitwise OR operation:

**NO\_VALUES** Only makes sense when combined with *EXPORT\_SYMBOLS* or *INITIALIZERS*. Do not calculate the value field in the return structure to avoid symbol binding by the loader to resolve symbol dependencies. If only a few symbol values are needed, *shl\_findsym()* can be used to find the values of interesting symbols. Not to be used with *GLOBAL\_VALUES*.

#### GLOBAL\_VALUES

Only makes sense when combined with *EXPORT\_SYMBOLS* or *INITIALIZERS*. Use the name and type information of each return symbol and find the most visible occurrence using all symbol information sources. The value and handle fields in the symbol return structure reflect where the most visible occurrence was found. Not to be used with *NO\_VALUES*.

The *memory* argument should point to a function with the same interface as *malloc()* (see *malloc(3C)*).

The return information consists of an array of the following records (defined in *<dl.h>*):

```
struct shl_symbol {
    char *name,
    short type,
    void *value,
    shl_t handle,
};
```

The *type* field in the return structure can have the values *TYPE\_PROCEDURE*, *TYPE\_DATA*, or *TYPE\_STORAGE*, where *TYPE\_STORAGE* is a subset of *TYPE\_DATA*. The *value* and *handle* fields are only valid if export symbols are requested and the *NO\_VALUES* flag is not specified. The *value* field contains the address of the symbol, while the *handle* field is the handle of the library that defined the symbol, or NULL for symbols defined via the *shl\_definesym()* routine and is useful in conjunction with the *GLOBAL\_VALUES* flag.

If successful, *shl\_getsymbols()* returns the number of symbols found; otherwise it returns -1.

**shl\_unload()** Can be used to detach a shared library from the process. The *handle* argument should be the handle returned from a previous call to **shl\_load()**. **shl\_unload()** returns 0 if successful; otherwise -1 is returned. Any initializers registered with the library are called before detachment. All explicitly loaded libraries are detached automatically on process termination.

**shl\_get()** Returns information about currently loaded libraries, including those loaded implicitly at startup time. The *index* argument is the ordinal position of the shared library in the shared library search list for the process. A subsequent call to **shl\_unload()** decrements the index values of all libraries having an index greater than the unloaded library. The index value -1 refers to the dynamic loader. The *desc* argument is used to return a pointer to a statically allocated buffer containing a descriptor for the shared library. The format of the descriptor is implementation dependent; to examine its format, look at the contents of file `/usr/include/dl.h`. Information common to all implementations includes the library handle, pathname, and the range of addresses the library occupies. The buffer for the descriptor used by **shl\_get()** is static; the contents should be copied elsewhere before a subsequent call to the routine. The routine returns 0 normally, or -1 if an invalid *index* is given.

**shl\_gethandle()** Returns information about the library specified by the *handle* argument. The special handle `PROG_HANDLE` can be used to refer to the program itself. The descriptor returned is the same as the one returned by the **shl\_get()** routine. The buffer for the descriptor used by **shl\_gethandle()** is static; the contents should be copied elsewhere before a subsequent call to the routine. The routine returns 0 normally, or -1 on error.

**shl\_get\_r()** This is a reentrant version of **shl\_get()**. The *desc* argument must point to a buffer of enough user-defined storage to be filled with the library descriptor described in `/usr/include/dl.h`. Its semantics are otherwise identical to **shl\_get()**.

**shl\_gethandle\_r()** This is a reentrant version of **shl\_gethandle()**. The *desc* argument must point to a buffer of enough user-defined storage to be filled with the library descriptor described in `/usr/include/dl.h`. Its semantics are otherwise identical to **shl\_gethandle()**.

## DIAGNOSTICS

If a library cannot be loaded, **shl\_load()** returns NULL and sets `errno` to indicate the error. All other functions return -1 on error and set `errno`.

If **shl\_findsym()** cannot find the indicated symbol, `errno` is set to zero. If **shl\_findsym()** finds the indicated symbol but cannot resolve all the symbols it depends on, `errno` is set to `ENOSYM`.

## ERRORS

Possible values for `errno` include:

[ENOEXEC]	The specified file is not a shared library, or a format error was detected.
[ENOSYM]	Some symbol required by the shared library could not be found.
[EINVAL]	The specified handle or index is not valid or an attempt was made to load a library at an invalid address.
[ENOMEM]	There is insufficient room in the address space to load the library.
[ENOENT]	The specified library does not exist.
[EACCES]	Read or execute permission is denied for the specified library.

## WARNINGS

**shl\_unload()** detaches the library from the process and frees the memory allocated for it, but does not break existing symbolic linkages into the library. In this respect, an unloaded shared library is much like a block of memory deallocated via **free()** (see *free(3C)*).

Some implementations may not, by default, export all symbols defined by a program (instead exporting only those symbols that are imported by a shared library seen at link time). Therefore the `-E` option to **ld(1)** should be used when using these routines if the loaded libraries are to refer to program symbols.

All sym  
the ass  
**AUTHOR**  
*shl\_load*  
**SEE ALSO**  
System 1  
*ld(1)*  
**Miscella**  
*dld.sl(5)*  
**Texts an**  
*Program*

## shl\_load(3X)

## shl\_load(3X)

All symbol information returned by `shl_getsymbols()`, including the name field, become invalid once the associated library is unloaded by `shl_unload()`.

### AUTHOR

*shl\_load(3X)* and related functions were developed by HP.

### SEE ALSO

#### System Tools:

*ld(1)* invoke the link editor

#### Miscellaneous:

*dld.sl(5)* dynamic loader

#### Texts and Tutorials

*Programming on HP-UX*

S