Solaris on Sun Hardware Reference Manual Supplement

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054

Part No. 817-3060-10
July 2003, Revision A

Send comments about this document to: `docfeedback@sun.com`

Preface

The *Solaris on Sun Hardware Reference Manual Supplement* contains reference manual pages (man pages) for software provided to Sun hardware customers with the Solaris 8 product. These supplement the man pages provided in the general *Solaris 8 Reference Manual*. This edition has bee updated to include man pages found in the Solaris 8 HW 7/03 release.

Before you can access some of the information published in this book through the `man` command, you may need to install software from the Solaris Software Supplement CD for your Solaris release. In most cases, when you install a software product from the Solaris Software Supplement CD, a package containing man pages about the software will be automatically installed. For information about installing the man page software, refer to the *Solaris 8 Sun Hardware Platform Guide.*

How This Book Is Organized

This manual contains man pages in alphabetical order within each category:

- User Commands (1)
- System Administration Commands (1M)
- Smartcard commands (3smartcard)
- File Formats (4)
- Device and Network Interfaces (7)

The man pages apply to the following products:

- CD Read/Write drives: `cdrw`
- SunFDDI™ network adapter software: `nf`, `nf_fddidaemon`, `nf_install_agents`, `nf_macid`, `nf_smtmon`, `nf_snmd`, `nf_snmd_kill`, `nf_stat`, `nf_sync`, `pf`, `pf_fddidaemon`, `pf_install_agents`, `pf_macid`, `pf_smtmon`, `pf_snmd`, `pf_snmd_kill`, `pf_stat`, `smt`

- SunHSI/P™ (PCI bus) network adapter software: `hsip`, `hsip_init`, `hsip_loop`, `hsip_stat`
- SunHSI/S™ (Sbus) network adapter software: `hsi`, `hsi_init`, `hsi_loop`, `hsi_stat`, `hsi_trace`
- Gigabit Ethernet driver: `bge`
- Sun Remote System Control (RSC): `rscadm`
- Administration functions for Sun Fire™ V210 systems: `scadm`
- SunVTS™ diagnostic software: `sunvts`, vts_cmd, `vtsk`, `vtsprobe`, `vtstty`, `vtsui`
- Netra™ t server environmental monitoring software: `envmond`, `envmond.conf`
- Dynamic Reconfiguration for certain platforms: `cfgadm_sbd`

Accessing Sun Documentation Online

The `docs.sun.com`^{SM} web site enables you to access Sun technical documentation on the Web. You can browse the archive or search for a specific book title or subject at: `http://docs.sun.com`

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. You can email your comments to us at: `docfeedback@sun.com`

Please include the part number (817-3060-10) of your document in the subject line of your email.

NAME

cdrw – CD read and write

SYNOPSIS

cdrw -i [**-vSCO**] [**-d** *device*] [**-p** *speed*] [*image-file*]

cdrw -a [**-vSCO**] [**-d** *device*] [**-p** *speed*] [**-T** *audio-type*] *audio-file1* [*audio-file2...*]

cdrw -x [**-v**] [**-d** *device*] [**-T** *audio-type*] *track-number out-file*

cdrw -c [**-vSC**] [**-d** *device*] [**-p** *speed*] [**-m** *tmp-dir*] [**-s** *src-device*]

cdrw -b [**-v**] [**-d** *device*] all session fast

cdrw -L [**-v**] [**-d** *device*]

cdrw -M [**-v**] [**-d** *device*]

cdrw -l [**-v**]

cdrw -h

The **cdrw** command provides the ability to create data and audio CDs. It also provides the ability to extract audio tracks from an audio CD. **cdrw** also has the ability to create data DVDs. **cdrw** requires that the device be MMC-compliant in order to create a CD/DVD.

cdrw will search for a CD/DVD writer device connected to the system, unless the user specifies a device with the **-d** option. If it finds a single such writer device, it will use that as the default CD/DVD writer device for the command.

When more than one CD/DVD writer is connected to the system, use the **-d** option to indicate which device is desired. The device name can be specified in one of the following ways: /dev/rdsk/cNtNdNsN, cNtNdNsN, cNtNdN, or a symbolic name used by volume manager, such as **cdrom** or **cdrom1**. The **-l** option will provide a list of CD/DVD writers.

For instructions on adding a USB-mass-storage-class-compliant CD/DVD-RW to your system, see **scsa2usb**7D.

Creating Data CD

When creating data CDs, **cdrw** uses the Track-At-Once mode of writing. With the **-i** option, the user will specify a file that contains the data to write on CD media. In the absence of such a file, **cdrw** will read data from standard input.

In either case, the data will typically first have been prepared by using the **mkisofs**(1M)terefentry> command to convert the file and file information into the High Sierra format used on CDs. See the examples that include use of this command.

Creating Data DVDs

cdrw can create single-session data DVDs on DVD+RW/DVD-RW drives using images generated from **mkisofs**(1M)terefentry>. These disks can be mounted on Solaris as hsfs filesystems. When making data DVDs, **cdrw**uses Disk-At-Once mode of writing which will close the media when writing is completed and prevent any further sessions from being added. The **-d**option must be used when writing the image to the DVD media since DAO mode requires that the size of the image to be known in advance.

In either case, the data will typically first have been prepared by using the **mkisofsle>(1M)** command to convert the file and file information into the High Sierra format used on CD s. See the examples that include use of this command.

Creating Audio CDs

For creating an audio CD, using the **-a** option, single or multiple audio files can be specified. All of the audio files should be in the supported audio formats. Currently approved formats are:

sun	Sun .au files with data in Red Book CDDA form
wav	RIFF (.wav) files with data in Red Book CDDA form
cda	.cda files having raw CD audio data (that is, 16 bit PCM stereo at 44.1 KHz sample rate in little-endian byteorder)
aur	.aur files having raw CD data in big-endian byteorder

If no audio format is specified, **cdrw** tries to understand the audio file format based on the file extension. The case of the characters in the extension is ignored. If a format is specified using the -**T** option, it will be assumed as the audio file type for all the files specified. Also, **-c** will close the session after writing the audio tracks. Therefore, the tracks to be written should be specified in a single command line.

Extracting Audio

cdrw can also be used for extracting audio data from an audio CD with the -**x** option. The CD should have tracks in Red Book CDDA form. By default, the output format is based on the file extension. A user can specify a **sun**, **wav**, **cda**, or **aur** output format using the -**T** option.

Copying CDs

cdrw can be used to copy single session data CD-ROMs and Red Book audio CDs. For copying a CD, **cdrw** looks for a specified source device. If no source device is specified when using the -**c** option, the current CD writing device is assumed to be the source. **cdrw** will extract the track or tracks into a temporary file and will look for a blank writable CD-R/RW media in the current CD writing device. If no such media is found, the user will be asked to insert a blank writable CD media in the current CD writing device. If enough space is not available in the default temporary directory, an alternative directory can be specified using the -**m** option.

Erasing CD-RW/DVD-RW/DVD+RW Media

Users have to erase the CD-RW media before it can be re-written. With the **-b** option, the following flavors of erasing are currently supported:

session	Erase the last session.
fast	Minimally erase the media.
all	Erase the entire media.

If the **session** erasing type is used, **cdrw** will erase the last session. If there is only one session recorded on the CD-RW (for example, a data/audio CD-RW created by this tool), then session erasing is useful as it will only erase the portion that is recorded, leaving behind a blank disk. This is faster than erasing the entire media. For DVD media, session erase will erase the whole media.

Fast erase will minimally erase the entire media by removing the PMA and TOC of the first session. It will not erase the user data and subsequent tracks on the media, but the media will be treated it was a blank disc.

If a complete erase is necessary, the media will have to be erased using the all option.

The **all** erasing type should be used if it is a multisession disk, or the last session is not closed, or disk status is unknown, and the user wishes to erase the disk. With this type of erase, **cdrw** will erase the entire disk.

Checking device-list or media-status

The user can get a list of CD/DVD writing devices currently present in the system with the **-l** option. Also, for a particular media, the user can get the blanking status and table of contents through the **-M** option. The **-M** option also prints information about the last session start address and the next writable address. This information, along with the **-O** option, can be used to create multisession CDs. Please refer to **mkisofs**(1M)terefentry> for more information.

The following options are supported:

-a Creates an audio disk. At least one *audio-file* name must be specified. A CD can not have more than 99 audio tracks, so no more than 99 audio files can be specified. Also, the maximum audio data that can be written to the media by default is 74 minutes, unless **-C** is specified.

-b Blanks a CD-RW media. The type of erasing must be specified by the **all**, **fast** or **session** argument.

-c Copies a CD. If no other argument is specified, the default CD writing device is assumed to be the source device as well. In this case, the copying operation will read the source media into a temporary directory and will prompt the user to place a blank media into the drive for copying to proceed.

-C Uses media stated capacity. Without this option, **cdrw** will use a default value for writable CD media, which is 74 minutes for an audio CD or 681984000 bytes for a data CD and 4.7 GB for DVD.

-d Specifies CD/DVD writing device.

-h Help. Prints usage message.

-i Specifies image file for creating data CD/DVDs. The file size should be less than what can be written on the media.

-l Lists all the CD/DVD writers found in the system.

-L Close the disk. If the media was left in an open state after the last write operation, it will be closed to prevent any further writing.

-m Uses an alternate temporary directory instead of system default temporary directory for storing track data while copying a CD/DVD. An alternate temporary directory might be required because the amount of data on a CD can be huge (as much as 800 Mbytes for an 80 minute audio CD and 4.7 GB for a DVD) and the system default temporary directory might not have that much space.

-M Reports media status. **cdrw** will report if the media is blank or not, its table of contents, the last session's start address, and the next writable address if the disk is open.

-O Keeps the disk open. **cdrw** will close the session, but it will keep the disk open so that another session can be added later on to create a multisession disk.

-p Sets the writing speed. For example, **-p 4** will set the speed to 4X. If this option is not specified, **cdrw** will use the default speed of the CD writer. If this option is specified, **cdrw** will try to set the drive write speed to this value, but there is no guarantee of the speed actually used by the drive.

-s Specifies source device for copying CD/DVD.

-S Simulation mode. In this mode, **cdrw** will do everything with the drive laser turned off, so nothing will be written to the media. This can be used to verify if the system can provide data at a rate good enough for CD writing.

-T Audio format to use extracting audio files or reading audio files for audio CD creation. The *audio-type* can be **sun**, **wav**, **cda**, or **aur**.

-v Verbose mode.

-x Extracts audio data from an audio track.

Example 1: Creating a data CD or DVD

example% **cdrw -i /local/iso_image**

Example 2: Creating a CD/DVD from a directory

This example creates a CD/DVD from the directory tree **/home/foo**:

example% **mkisofs -r /home/foo** >**/image ; cdrw -i**
-p 1 /image

Example 3: Extracting an audio track number

This example extracts audio track number **1** to **/home/foo/song1.wav**:

example% **cdrw -x -T wav 1**
/home/foo/song1.wav

Example 4: Using wav files

This example creates an audio CD from **wav** files on disk:

example% **cdrw -a song1.wav song2.wav song3.wav**
song4.wav

Example 5: Erasing a CD-RW/DVD-RW/DVD+RW media

This example erases a rewritable media in a drive:

example% **cdrw -b all**

Example 6: Creating a data CD/DVD with multiple drives

This example creates a data CD/DVD on a system with multiple CD/DVD-R/RW drives:

example% **cdrw -d c1t6d0s2 -i**
/home/foo/iso-image

Example 7: Checking data delivery rate

This example checks if the system can provide data to a CD-RW or DVD drive at a rate sufficient for the write operation:

example% **cdrw -S -i**
/home/foo/iso-image

Example 8: Running at a higher priority

This example runs **cdrw** at a higher priority (for root user only):

example# **priocntl -e -p 60 cdrw -i**
/home/foo/iso-image

Example 9: Creating a multi-session disk

Create the first session image using **mkisofs**(1M)terefentry> and record it onto the disk without closing the disk:

example% **cdrw -O -i /home/foo/iso-image**

Additional sessions can be added to an open disk by creating an image with **mkisofs**(1M)terefentry> using the session start and next writable address reported by **cdrw**.

example% **cdrw -M**

```
Track No. |Type    |Start address
----------+--------+-------------
 1        |Data    | 0
Leadout   |Data    | 166564

Last session start address: 162140
```

Next writable address: 173464

example% **mkisofs -o /tmp/image2 -r -C 0,173464 -M** \
   /dev/rdsk/c0t2d0s2 /home/foo

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcdrw

audioconvert>(1), **mkisofs**(1M)terefentry>, **priocntl**(1), **attributes**(5), **rbac**(5), **scsa2usb**7D, **sd**(7D)

The CD/DVD writing process requires data to be supplied at a constant rate to the drive. It is advised to keep I/O activity to a minimum and shut down the related applications while writing CDs.

When making copies or extracting audio tracks, it is better to use an MMC compliant source CD-ROM drive. The CD writing device can be used for this purpose.

Before writing a CD, ensure that the media is blank by using the **-M** option and use the **-S** simulation mode to test the system to make sure it can provide data at the required rate. In case the system is not able to provide data at the required rate, try simulation with a slower write speed set through the **-p** option. Users can also try to run **cdrw** at a higher priority using the **priocntl**1 command.

The **-p** option is provided for users who are aware of the CD-R/RW drive and its capabilities to operate at different write speeds. Some commercially available drives handle the drive speed setting command differently, so use this option judiciously.

Most commercially available drives allow writing beyond 74 minutes as long as the media has the capacity (such as 80-minute media). However, such capability of writing beyond 74 minutes might not be supported by the drive in use. If the drive being used supports such capability, then use the **-C** option to indicate that the tool should rely on the capacity indicated by the media.

The **cdrw** command uses **rbac**(5) to control user access to the devices. By default, **cdrw** is accessible to all users but can be restricted to individual users. Please refer to "Administering CD-R/CD-RW devices" in the for more information.

NAME

rmformat – removable rewritable media format utility

SYNOPSIS

rmformat [-**DeHpUv**] [-**b** *label*] [-**c** *blockno*] [-**F**quick | long | force] [-**R** enable | disable] [-**s** *filename*] [-**w** enable | disable] [-**W** enable | disable] [*devname*]

rmformat **-V** read | write *devname*

The **rmformat** utility is used to format, label, partition, and perform other miscellaneous functions on removable, rewritable media that include floppy drives, IOMEGA Zip/Jaz products, and the **PCMCIA** memory and ata cards. In addition, the **rmformat** utility should also be used with all **USB** mass storage devices, including **USB** hard drives. This utility can also be used for the verification and surface analysis and for repair of the bad sectors found during verification if the drive or the driver supports bad block management.

rmformat provides functionality to read/write protect the media with or without a password. The password protection enabling or disabling is possible only with selective rewritable media such as the IOMEGA Zip/Jaz products.

After formatting, **rmformat** writes the label, which covers the full capacity of the media as one slice on floppy and **PCMCIA** memory cards to maintain compatibility with the behavior of **fdformat**. On Zip/Jaz devices, the driver exports one slice covering the full capacity of the disk as default. **rmformat** does not write the label on Zip/Jaz media, unless explicitly requested. The partition information can be changed with the help of other options provided by **rmformat**.

The following options are supported:

-b *label*
Labels the media with a SUNOS label. A SUNOS volume label name is restricted to 8 characters. For writing a **DOS** Volume label, the user should use **mkfs_pcfs**(1M).

-**c** *blockno*
Corrects and repairs the given block. This correct and repair option may not be applicable to all devices supported by **rmformat**, as some devices may have a drive with bad block management capability and others may have this option implemented in the driver. If the drive or driver supports bad block management, a best effort is made to rectify the bad block. If the bad block still cannot be rectified, a message is displayed to indicate the failure to repair. The block number can be provided in decimal, octal, or hexadecimal format.

The normal floppy and **PCMCIA** memory and ata cards do not support bad block management.

-D Formats a 720KB (3.5 inch) double density diskette. This is the default for double density type drives. This option is needed if the drive is a high or extended-density type.

-e Ejects the media upon completion. This feature may not be available if the drive does not support motorized eject.

-F quick | long | force

Formats the media.

The **quick** option starts a format without certification or format with limited certification of certain tracks on the media.

The **long** option starts a complete format. For some devices this might include the certification of the whole media by the drive itself.

The **force** option to format is provided to start a long format without user confirmation before the format is started. For drives which have a password protection mechanism, it clears the password while formatting. This feature is useful when a password is no longer available. On those media which do not have such password protection, **force** starts a long format.

In legacy media such as floppy drives, all options start a long format depending on the mode (Extended Density mode, High Density mode, or Double Density mode) with which the floppy drive operates by default. On **PCMCIA** memory cards, all options start a long format.

-H Formats a 1.44 MB (3.5 inch) high density diskette. This is the default for high density type drives. It is needed if the drive is the Extended Density type.

-p Prints the protection status of the media. This option prints information whether the media is write, read, or password protected.

-R enable | disable

Enables read / write protection with a password or disables the password read / write protection. This always works in interactive mode, as the password is requested from the user in an interactive manner to maintain security.

A password length of 32 bytes (maximum) is allowed for the IOMEGA products that support this feature. This option is applicable only for IOMEGA products. IOMEGA products do not allow read / write protection without a password. On the devices which do not have such software read / write protect facility, warnings indicating the non-availability of this feature are provided.

-s *filename*

Enables the user to lay out the partition information in the SUNOS label.

The user should provide a file as input with information about each slice in a format providing byte offset, size required, tags, and flags, as follows:

slices: *n* = *offset*, *size* [, *flags*, *tags*]

where *n* is the slice number, *offset* is the byte offset at which the slice *n* starts, and *size* is the required size for slice *n*. Both *offset* and *size* must be a multiple of 512 bytes.

These numbers can be represented as decimal, hexadecimal, or octal numbers. No floating point numbers are accepted. Details about maximum number of slices can be obtained from the *System Administration Guide: Basic Administration*.

To specify the *size* or *offset* in kilobytes, megabytes, or gigabytes, add **KB**, **MB**, **GB**, respectively. A number without a suffix is assumed to be a byte offset. The flags are represented as follows:

wm = read-write, mountable
wu = read-write, unmountable
ru = read-only, unmountable

The tags are represented as follows: **unassigned**, **boot**, **root**, **swap**, **usr**, **backup**, **stand**, **var**, **home**, **alternates**.

The tags and flags can be omitted from the four tuple when finer control on those values is not required. It is required to omit both or include both. If the tags and flags are omitted from the four tuple for a particular slice, a default value for each is assumed. The default value for flags is **wm** and for tags is **unassigned**.

Either full tag names can be provided or an abbreviation for the tags can be used. The abbreviations can be the first two or more letters from the standard tag names. **rmformat** is case insensitive in handling the defined tags & flags.

Slice specifications are separated by :

For example:

```
slices: 0 = 0, 30MB, "wm", "home" :
        1 = 30MB, 51MB :
        2 = 0, 100MB, "wm", "backup" :
        6 = 81MB, 19MB
```

rmformat does the necessary checking to detect any overlapping partitions or illegal requests to addresses beyond the capacity of the media under consideration. There can be only one slice information entry for each slice *n*. If multiple slice information entries for the same slice *n* are provided, an appropriate error message is displayed. The slice **2** is the backup slice covering the whole disk capacity. The pound sign character, #, can be used to describe a line of comments in the input file. If the line starts with #, then **rmformat** ignores all the characters following # until the end of the line.

Partitioning some of the media with very small capacity is permitted, but be cautious in using this option on such devices.

-U Performs **umount** on any file systems and then formats. See **mount**(1M). This option unmounts all the mounted slices and issues a long format on the device requested.

-V read | write
Verifies each block of media after format. The write verification is a destructive mechanism. The user is queried for confirmation before the verification is started. The output of this option is a list of block numbers, which are identified as bad.

The read verification only verifies the blocks and report the blocks which are prone to errors.

The list of block numbers displayed can be used with the **-c** option for repairing.

-w enable | disable
Enables or disables the write protection on media. On devices that do not have a software write protect facility, a message indicating non-availability of this feature is displayed.

-W enable | disable
Enables or disables write protection with password. This option always works in interactive mode, as a password is requested from the user to maintain security.

A maximum password length of 32 bytes is allowed for IOMEGA products that support this feature. On devices that do not have the write protection with password, the software displays appropriate messages indicating the non-availability of such features.

The following operand is supported:

devname
devname can be provided as absolute device pathname or relative pathname for the device from the current working directory or the nickname as exported by the System Volume manager. See **vold**(1M).

For floppy devices, to access the first drive use **/dev/rdiskette0** (for systems without volume management) or **floppy0** (for systems with volume management). Specify **/dev/rdiskette1** (for systems without volume management) or **floppy1** (for systems with volume management) to use the second drive.

For systems without volume management running, the user can also provide the absolute device pathname as **/dev/rdsk/c***?*t*?*d*?*s*?* or the appropriate relative device pathname from the current working directory.

Example 1: Formatting a diskette

```
example$ rmformat -F quick /dev/rdiskette
Formatting will erase all the data on disk.
Do you want to continue? (y/n)y
```

Example 2: Formatting a Zip drive

```
example$ rmformat -F quick /vol/dev/aliases/zip0
Formatting will erase all the data on disk.
Do you want to continue? (y/n)y
```

Example 3: Formatting a diskette for a UFS file system

The following example formats a diskette and creates a UFS file system:

```
example$ rmformat -F quick /vol/dev/aliases/floppy0
Formatting will erase all the data on disk.
Do you want to continue? (y/n)y
example$ su
# /usr/sbin/newfs /vol/dev/aliases/floppy0
newfs: construct a new file system /dev/rdiskette: (y/n)? y
/dev/rdiskette: 2880 sectors in 80 cylinders of 2 tracks, 18 sectors
         1.4MB in 5 cyl groups (16 c/g, 0.28MB/g, 128 i/g)
 super-block backups (for fsck -F ufs -o b=#) at:
 32, 640, 1184, 1792, 2336,
#
```

Example 4: Formatting removable media for a PCFS file system

The following example shows how to create an alternate **fdisk** partition:

```
example$ rmformat -F quick /dev/rdsk/c0t4d0s2:c
Formatting will erase all the data on disk.
Do you want to continue? (y/n)y
example$ su
# fdisk /dev/rdsk/c0t4d0s2:c
# mkfs -F pcfs /dev/rdsk/c0t4d0s2:c
Construct a new FAT file system on /dev/rdsk/c0t4d0s2:c: (y/n)? y
#
```

The following example describes how to create a **PCFS** file system *without* an **fdisk** partition:

```
example$ rmformat -F quick /dev/rdiskette
Formatting will erase all the data on disk.
Do you want to continue? (y/n)y
example$ su
# mkfs -F pcfs -o nofdisk,size=2 /dev/rdiskette
Construct a new FAT file system on /dev/rdiskette: (y/n)? y
#
```

Example 5: Enabling or disabling read or write protection

The following example shows how to enable write protection and set a password on a Zip drive:

```
example$ rmformat -W enable /vol/dev/aliases/zip0
Please enter password (32 chars maximum): xxx
Please reenter password: xxx
```

The following example shows how to disable write protection and remove the password on a Zip drive:

```
example$ rmformat -W disable /vol/dev/aliases/zip0
Please enter password (32 chars maximum): xxx
```

The following example shows how to enable read protection and set a password on a Zip drive:

```
example$ rmformat -R enable /vol/dev/aliases/zip0
Please enter password (32 chars maximum): xxx
Please reenter password: xxx
```

The following example shows how to disable read protection and remove the password on a Zip drive:

```
example$ rmformat -R disable /vol/dev/aliases/zip0
Please enter password (32 chars maximum): xxx
```

/vol/dev/diskette0
Directory providing block device access for the media in floppy drive 0.

/vol/dev/rdiskette0
Directory providing character device access for the media in floppy drive 0.

/vol/dev/aliases
Directory providing symbolic links to the character devices for the different media under the control of volume management using appropriate alias.

/vol/dev/aliases/floppy0
Symbolic link to the character device for the media in floppy drive 0.

/vol/dev/aliases/zip0
Symbolic link to the character device for the media in Zip drive 0.

/vol/dev/aliases/jaz0
Symbolic link to the character device for the media in Jaz drive 0.

/dev/rdiskette
Symbolic link providing character device access for the media in the primary floppy drive, usually drive 0.

/vol/dev/dsk
Directory providing block device access for the **PCMCIA** memory and ata cards and removable media devices.

/vol/dev/rdsk
Directory providing character device access for the **PCMCIA** memory and ata cards and removable media devices.

/vol/dev/aliases/pcmemS
Symbolic link to the character device for the **PCMCIA** memory card in socket S, where S represents a **PCMCIA** socket number.

/vol/dev/aliases/rmdisk0
Symbolic link to the generic removable media device that is not a Zip, Jaz, **CD-**

ROM, floppy, **DVD-ROM**, **PCMCIA** memory card, and so forth.

/dev/rdsk
Directory providing character device access for the **PCMCIA** memory and ata cards and other removable devices.

/dev/dsk
Directory providing block device access for the **PCMCIA** memory and ata cards and other removable media devices.

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu

cpio(1), **eject**(1), **fdformat**(1), **tar**(1), **volcancel**(1), **volcheck**(1), **volmissing**(1), **volrmmount**(1), **format**(1M), **mkfs_pcfs**(1M), **mount**(1M), **newfs**(1M), **prtvtoc**(1M), **rmmount**(1M), **rpc.smserved**(1M), **vold**(1M), **rmmount.conf**(4), **vold.conf**(4), **attributes**(5), **scsa2usb**(7D), **sd**(7D), **pcfs**(7FS), **udfs**(7FS)

System Administration Guide: Basic Administration

A rewritable media or **PCMCIA** memory card or **PCMCIA** ata card containing a **ufs** file system created on a SPARC-based system (using **newfs**(1M)) is not identical to a rewritable media or **PCMCIA** memory card containing a **ufs** file system created on an based system. Do not interchange any removable media containing **ufs** between these platforms; use **cpio**(1) or **tar**(1) to transfer files on diskettes or memory cards between them. For interchangeable filesystems refer to **pcfs**(7FS) and **udfs**(7FS).

Currently, bad sector mapping is not supported on floppy diskettes or **PCMCIA** memory cards. Therefore, a diskette or memory card is unusable if **rmformat** finds an error (**bad sector**).

NAME

cfgadm_sbd – **cfgadm** commands for system board administration

SYNOPSIS

cfgadm **-l** **-a** **-o** parsable *ap_id*

cfgadm **-c** *function* **-f** **-y** | **-n** **-o** unassign | nopoweroff *ap_id*

cfgadm **-t** **-f** *ap_id*

cfgadm **-x** *function* *ap_id*

DESCRIPTION

The **sbd.so.1** plugin provides dynamic reconfiguration functionality for connecting, configuring, unconfiguring, and disconnecting class **sbd** system boards. It also enables you to connect or disconnect a system board from a running system without having to reboot the system.

The **cfgadm** command resides in **/usr/sbin**. See **cfgadm**(1M).

Each board slot appears as a single attachment point in the device tree. Each component appears as a dynamic attachment point. You can view the type, state, and condition of each component, and the states and condition of each board slot by using the **-a** option.

Component Types

The following are the names and descriptions of the component types:

cpu	**CPU**
pci	**I/O** device
memory	Memory

Note: An operation on a memory component affects all of the memory components on the board.

Component Conditions

The following are the names and descriptions of the component conditions:

failed	The component failed testing.
ok	The component is operational.
unknown	The component has not been tested.

Component States

The following is the name and description of the receptacle state for components:

connected	The component is connected to the board slot.

The following are the names and descriptions of the occupant states for components:

configured	The component is available for use by the Solaris operating environment.
unconfigured	The component is not available for use by the Solaris operating environment.

Board Conditions

The following are the names and descriptions of the board conditions.

failed	The board failed testing.
ok	The board is operational.

unknown	The board has not been tested.
unusable	The board slot is unusable.

Board States

Inserting a board changes the receptacle state from empty to disconnected. Removing a board changes the receptacle state from disconnected to empty.

Caution: Removing a board that is in the connected state or that is powered on and in the disconnected state crashes the operating system and can result in permanent damage to the system.

The following are the names and descriptions of the receptacle states for boards:

connected	The board is powered on and connected to the system bus. You can view the components on a board only after it is in the connected state.
disconnected	The board is disconnected from the system bus. A board can be in the disconnected state without being powered off. However, a board must be powered off and in the disconnected state before you remove it from the slot.
empty	A board is not present.

The occupant state of a disconnected board is always unconfigured. The following table contains the names and descriptions of the occupant states for boards:

configured	At least one component on the board is configured.
unconfigured	All of the components on the board are unconfigured.

Dynamic System Domains

Platforms based on dynamic system domains (DSDs, referred to as domains in this document) divide the slots in the chassis into electrically isolated hardware partitions (that is, DSDs). Platforms that are not based on DSDs assign all slots to the system permanently.

A slot can be empty or populated, and it can be assigned or available to any number of domains. The number of slots available to a given domain is controlled by an available component list (**ACL**) that is maintained on the system controller. The **ACL** is not the access control list provided by the Solaris operating environment.

A slot is visible to a domain only if the slot is in the domain’s **ACL** and if it is not assigned to another domain. An unassigned slot is visible to all domains that have the slot in their **ACL**. After a slot has been assigned to a domain, the slot is no longer visible to any other domain.

A slot that is visible to a domain, but not assigned, must first be assigned to the domain before any other state changing commands are applied. The assign can be done explicitly using **-x** assign or implicitly as part of a connect. A slot must be unassigned from a domain before it can be used by another domain. The unassign is always explicit, either directly using **-x** unassign or as an option to disconnect using **-o** unassign.

State Change Functions

Functions that change the state of a board slot or a component on the board can be issued concurrently against any attachment point. Only one state changing operation is permitted at a given time. A **Y** in the Busy field in the state changing information indicates an operation is in progress.

The following list contains the functions that change the state:

- **configure**
- **unconfigure**
- **connect**
- **disconnect**

Availability Change Functions

Commands that change the availability of a board can be issued concurrently against any attachment point. Only one availability change operation is permitted at a given time. These functions also change the information string in the **cfgadm -l** output. A **Y** in the Busy field indicates that an operation is in progress.

The following list contains the functions that change the availability:

- **assign**
- **unassign**

Condition Change Functions

Functions that change the condition of a board slot or a component on the board can be issued concurrently against any attachment point. Only one condition change operation is permitted at a given time. These functions also change the information string in the **cfgadm -l** output. A **Y** in the Busy field indicates an operation is in progress.

The following list contains the functions that change the condition:

- **poweron**
- **poweroff**
- **test**

Unconfigure Process

This section contains a description of the unconfigure process, specifically illustrating the copy-rename source and target board states at different stages of the process.

In the following code examples, the permanant memory on board 0 must be moved to another board in the domain. Thus, board 0 is the source, and board 1 is the target.

A status change operation cannot be initiated on a board while it is marked as busy. For brevity, the **CPU** information has been removed from the code examples.

The process is started with the following command:

```
# cfgadm -c unconfigure -y sbd/slot0::memory
```

First, the memory on board 1 in the same address range as the permanant memory on board 0 must be deleted. During this phase, the source board, the target board, and the memory attachment points are marked as busy. You can display the status with the following command:

```
# cfgadm -a -s cols=ap_id:type:r_state:o_state:busy  sbd/slot0 sbd/slot1
Ap_Id              Type      Receptacle    Occupant      Busy
sbd/slot0          cpu/mem   connected     configured    y
sbd/slot0::memory  memory    connected     configured    y
sbd/slot1          cpu/mem   connected     configured    y
sbd/slot1::memory  memory    connected     configured    y
```

After the memory has been deleted on board 1, it is marked as unconfigured. The memory on board 0 remains configured, but it is still marked as busy, as in the following example.

```
Ap_Id              Type      Receptacle    Occupant      Busy
sbd/slot0          cpu/mem   connected     configured    y
sbd/slot0::memory  memory    connected     configured    y
sbd/slot1          cpu/mem   connected     configured    y
sbd/slot1::memory  memory    connected     unconfigured  n
```

The memory from board 0 is then copied to board 1. After it has been copied, the occupant state for the memory is switched. The memory on board 0 becomes unconfigured, and the memory on board 1 becomes configured. At this point in the process, only board 0 remains busy, as in the following example.

```
Ap_Id              Type      Receptacle    Occupant      Busy
sbd/slot0          cpu/mem   connected     configured    y
sbd/slot0::memory  memory    connected     unconfigured  n
sbd/slot1          cpu/mem   connected     configured    n
sbd/slot1::memory  memory    connected     configured    n
```

After the entire process has been completed, the memory on board 0 remains unconfigured, and the attachment points are not busy, as in the following example.

```
Ap_Id              Type      Receptacle    Occupant      Busy
sbd/slot0          cpu/mem   connected     configured    n
sbd/slot0::memory  memory    connected     unconfigured  n
sbd/slot1          cpu/mem   connected     configured    n
sbd/slot1::memory  memory    connected     configured    n
```

The nonpageable memory has been moved, and the memory on board 0 has been unconfigured. At this point, you can initiate a new state changing operation on either board.

Platform-Specific Options

You can specify platform-specific options that follow the options interpreted by the system board plugin. All platform-specific options must be preceded by the **platform** keyword. The following example contains the general format of a command with platform-specific options:

command -o *sbd_options*,platform=*platform_options*

OPTIONS

This man page does not include the **-v**, **-s**, or **-h** options for the **cfgadm** command. See **cfgadm**(1M) for descriptions of those options. The following options are supported by the **cfgadm_sbd** plugin:

-**c** *function* Performs a state change function. You can use the following functions:

unconfigure

Changes the occupant state to unconfigured. This function applies to system board slots and to all of the components on the system board.

The **unconfigure** function removes the **CPU**s from the **CPU** list and deletes the physical memory from the system memory pool. If any device is still in use, the **cfgadm** command fails and reports the failure to the user. You can retry the command as soon as the device is no longer busy. If a **CPU** is in use, you must ensure that it is off line before you proceed. See **pbind**(1M), **psradm**(1M) and **psrinfo**(1M).

The **unconfigure** function moves the physical memory to another system board before it deletes the memory from the board you want to unconfigure. Depending of the type of memory being moved, the command fails if it cannot find enough memory on another board or if it cannot find an appropriate physical memory range.

For permanant memory, the operating system must be suspended (that is, quiesced) while the memory is moved and the memory controllers are reprogrammed. If the operating system must be suspended, you will be prompted to proceed with the operation. You can use the **-y** or -**n** options to to always answer yes or no respectively.

Moving memory can take several minutes to complete, depending on the amount of memory and the system load. You can monitor the progress of the operation by issuing a status command against the memory attachment point. You can also interrupt the memory operation by stopping the **cfgadm** command. The deleted memory is returned to the system memory pool.

disconnect Changes the receptacle state to disconnected. This function applies only to system board slots.

If the occupant state is configured, the **disconnect** function attempts to unconfigure the occupant. It then powers off the system board. At this point, the board can be removed from the slot.

This function leaves the board in the assigned state on platforms that support dynamic system domains.

If you specify **-o nopoweroff**, the **disconnect** function leaves the board powered on. If you specify -**o unassign**, the **disconnect** function unassigns the board from the domain.

If a board is unassigned from a domain, it is available to be assigned to another domain. If it is, it will not be available to the domain from which is was unassigned.

configure Changes the occupant state to configured. This function applies to system board slots and to any components on the system board.

If the receptacle state is disconnected, the **configure** function attempts to connect the receptacle. It then walks the tree of devices that is created by the **connect** function, and attaches the devices if necessary. Running this function configures all of the components on the board, except those that have already been configured.

For **CPU**s, the **configure** function adds the **CPU**s to the **CPU** list. For memory, the **configure** function ensures that the memory is initialized then adds the memory to the system memory pool. The **CPU**s and the memory are ready for use after the **configure** function has been completed successfully.

For I/O devices, you must use the **mount** and the **ifconfig** commands before the devices can be used. See **ifconfig**(1M) and **mount**(1M).

connect Changes the receptacle state to connected. This function applies only to system board slots.

If the board slot is not assigned to the domain, the **connect** function attempts to assign the slot to the domain. Next, it powers on and tests the board, then it connects the board electronically to the system bus and probes the components.

After the **connect** function is completed successfully, you can use the **-a** option to view the status of the components on the board. The **connect** function leaves all of the components in the unconfigured state.

The assignment step applies only to platforms that support dynamic system domains.

-f Overrides software state changing constraints. With the **-t** option, the **-f** option forces the board to be tested, even if the system board has already been tested.

The **-f** option never overrides fundamental safety and availability constraints of the hardware and operating system.

-l Lists the general and platform-specific information for each attachment point type. Platform-specific information is appended to the **info** field.

The parsable **info** field is composed of the following:

cpu The **cpu** type displays the following information:

cpuid=# Where # is a number, representing the **ID** of the **CPU**.

speed=# Where # is a number, representing the speed of the **CPU** in **MHz**.

ecache=# Where # is a number, representing the size of the ecache in **MBytes**.

memory The **memory** type displays the following information, as appropriate:

address=# Where # is a number, representing the base physical address.

size=# Where # is a number, representing the size of the memory in **KBytes**.

permanent=#
Where # is a number, representing the size of nonpageable memory in **KBytes**.

unconfigurable
An operating system setting that prevents the memory from being unconfigured.

inter-board-interleave
The board is participating in interleaving with other boards.

source=*ap_id*
Represents the source attachment point.

target=*ap_id*
Represents the target attachment point.

deleted=# Where # is a number, representing the amount of memory that has already been deleted in **KBytes**.

remaining=#
Where # is a number, representing the amount of memory to be deleted in **KBytes**.

io The **io** type displays the following information:

device=*path*
Represents the physical path to the I/O component.

referenced The I/O component is referenced.

board The **board** type displays the following information:

assigned The board is assigned to the domain.

powered-on
The board is powered on.

The same items appear in the **info** field in a more readable format if the **-o parsable** option is not specified.

-o parsable Returns the information in the **info** field as a boolean *name* or a set of **name=value** pairs, separated by a space character and enclosed in double quote **marks. Escapes double quotes and backslash (\fR) characters with a backslash**

(\fR). The absence of a boolean indicates that the opposite applies.

The **-**o parsable option can be used in conjunction with the **-s** option. See the **cfgadm**(1M) man page for more information about the **-s** option.

-t Tests the board.

Before a board can be connected, it must pass the appropriate level of testing. By default, if the board has already passed the appropriate level of testing, it is not tested again; however, you can use the **-f** option to force another test.

-x *function* Performs an sbd-class function. You can use the following functions:

assign Assigns a board to a domain.

The receptacle state must be disconnected or empty. The board must also be listed in the domain available component list. See Dynamic System Domains.

unassign Unassigns a board from a domain.

The receptacle state must be disconnected or empty. The board must also be listed in the domain available component list. See Dynamic System Domains.

poweron Powers the system board on.

The receptacle state must be disconnected.

poweroff Powers the system board off.

The receptacle state must be disconnected.

OPERANDS

The following operands are supported:

Receptacle *ap_id*
The receptacle attachment point **ID** takes the form **sbd/***slot_nameX*, where *X* equals the slot number.

Component *ap_id*
The component attachment point **ID** takes the form *component_typeX*, where *component_type* equals one of the component types described in Component Types and *X* equals the component number.

The component number is a board-relative unit number.

EXAMPLES

Example 1: Listing All of the System Board Attachments Points

```
# cfgadm -a -s select=class(sbd)
Ap_Id              Type       Receptacle    Occupant     Condition
sbd/slot0          cpu/mem    connected     configured   ok
sbd/slot0::cpu0    cpu        connected     configured   ok
sbd/slot0::memory  memory     connected     configured   ok
sbd/slot1          pci        connected     configured   ok
sbd/slot1::pci0    pci        connected     configured   ok
sbd/slot1::pci1    pci        connected     configured   failed
```

```
sbd/slot2           cpu/mem   disconnected   unconfigured   failed
sbd/slot3           cpu/mem   disconnected   unconfigured   unknown
sbd/slot4           unknown   empty          unconfigured   unusable
```

This example demonstrates the mapping of the following conditions:

- The second PCI node in Slot 1 failed testing.
- The board in Slot 2 failed testing.
- Slot 4 is unusable; thus, you cannot hot plug a board into that slot.

Example 2: Listing All of the CPUs on the System Board Attachments Points

```
 # cfgadm -a -s select=class(sbd):type(cpu)
Ap_Id              Type     Receptacle    Occupant     Condition
sbd/slot0::cpu0     cpu     connected     configured    ok
sbd/slot0::cpu1     cpu     connected     configured    ok
sbd/slot0::cpu2     cpu     connected     configured    ok
sbd/slot0::cpu3     cpu     connected     configured    ok
```

Example 3: Displaying the CPU Information Field

```
 # cfgadm -l -s noheadings,cols=info sbd/slot0::cpu0
cpuid 16 speed 400 Mhz ecache 8 Mbytes
```

Example 4: Displaying the CPU Information Field in parsable Format

```
 # cfgadm -l -s noheadings,cols=info -o parsable sbd/slot0::cpu0
"cpuid=16", "speed=400", "ecache=8"
```

Example 5: Displaying the Devices on an I/O Board

```
 # cfgadm -a -s noheadings,cols=ap_id:info -o parsable sbd/slot1
sbd/slot1::pci0 "device=/devices/saf@0/pci@0,2000" referenced
sbd/slot1::pci1 "device=/devices/saf@0/pci@1,2000" referenced
```

Example 6: Monitoring an Unconfigure Operation

In the following example, the memory sizes are displayed in Mbytes.

```
 # cfgadm -c unconfigure -y sbd/slot0::memory  # cfgadm -l -s noheadings,cols=info -o parsable
sbd/slot0::memory sbd/slot1::memory
"address=0x0", "size=16384", "target=sbd/slot1::memory", "deleted=1240", "remaining=6144",
"address=0x1000000", "size=16384", "source=sbd/slot0::memory"
```

Example 7: Assigning a Slot to a Domain

```
# cfgadm -x assign sbd/slot2
```

Example 8: Unassigning a Slot from a Domain

```
# cfgadm -x unassign sbd/slot3
```

FILES

The following files are supported:

/usr/platform/sun4u/cfgadm/sbd.so.1
plugin library module

/dev/cfg/sbd/slot*

symbolic names

/usr/sbin/cfgadm
: **cfgadm** command

AVAILABILITY

SUNWkvm.u

SEE ALSO

cfgadm(1M), **devfsadm**(1M), **ifconfig**(1M), **mount**(1M), **pbind**(1M), **psradm**(1M), **psrinfo**(1M), **config_admin**(3CFGADM), **attributes**(5)

NOTES

This section contains information about how to monitor the progress of a memory delete operation and on platform-specific behaviors of the **cfgadm** command.

Memory Delete Monitoring

The following shell script can be used to monitor the progress of a memory delete operation.

```
 # cfgadm -c unconfigure -y sbd/slot0::memory
#!/bin/sh

while true
do
        eval ‘cfgadm -l -s noheadings,cols=info -o parsable sbd/slot15.0::memory‘
        if [ -n "$remaining" ]
        then
                echo $remaining mbytes
        else
                echo memory delete is done
                exit 0
        fi
        sleep 1
done
```

Sun Fire 15000 Platform Notes

The **-t** and **-x** options behave differently on the Sun Fire 15000 platform. The following list describes their behavior:

-t
: The system controller uses a CPU to test system boards by running **LPOST**, sequenced by the **hpost** command. To test I/O boards, the driver starts the testing in response to the **-t** option, and the test runs automatically without user intervention. The driver unconfigures a CPU and a stretch of contiguous physical memory. Then, it sends a command to the system controller to test the board. The system controller uses the CPU and memory to test the I/O board from inside of a transaction/error cage.

-x assign | unassign
: In the Sun Fire 15000 system administration model, the platform administrator controls the platform hardware from the system controller. Only the platform administrator can assign or unassign free boards to or from a domain by adding the board to the available component list for that domain. The domain administrator is not allowed to assign or unassign

boards to or from a domain, unless the board is already in the available component list for that domain.

For the Sun Fire 15000 platform, a logical system slot is represented as *slot_nameX.Y*. Where *X* represents the expander position (0 to 17) and *Y* represents the slot number (0 or 1).

In the following example, the domain contains three CPU/memory boards and two I/O boards.

```
 # cfgadm -l -s "select=class(sbd)"
Ap_Id          Type      Receptacle   Occupant     Condition
sbd/slot3.0    CPU       connected    configured   ok
sbd/slot11.0   CPU       connected    configured   ok
sbd/slot11.1   CPU       connected    configured   ok
sbd/slot15.0   CPU       connected    configured   ok
sbd/slot15.1   hpci      connected    configured   ok
```

NAME

clbconfig – Content Load Balancer Configuration Script

SYNOPSIS

/opt/SUNWclb/bin/clbconfig [add <interface> | remove <interface> | list]

DESCRIPTION

This script is used to add or remove interfaces for content load balancing. It is also used to list the interfaces participating in content load balancing.

EXAMPLES

The following examples show how to add, remove and list interfaces.

Example 1: Add interface

```
example% clbconfig add ce0
```

This adds the interface for content load balancing.

Example 2: Remove interface

```
example% clbconfig remove ce0
```

This removes the interface from content load balancing list.

Example 3: List interfaces participating in load balancing.

```
example% clbconfig list
```

This lists the interfaces participating in content load balancing.

SEE ALSO

clb.conf(4)

NAME

envmond - environmental monitor daemon

SYNOPSIS

/usr/platform/SUNW,UltraSPARC–IIi–Netract/lib/envmond/sparcv9/envmond [**–d**] [**–f** *file*] [**–g** *granularity*]

AVAILABILITY

SUNWcteux

DESCRIPTION

The **envmond** daemon polls system environment monitoring devices to check for conditions that may require corrective action. In order to do this, the daemon reads a configuration file on startup, during the initial Solaris boot process, to find out which environmental devices will be monitored. Each configuration file entry describing an environmental device is referred to as a policy, and the supported policy entries are described in **envmond.conf**(4)**.**

The **envmond** daemon logs appropriate messages to a system log file via **syslogd**(1M)**.**

The **envmond** daemon will reread its configuration information file whenever it receives a hang-up signal, SIGHUP.

OPTIONS

–d Sets Debug mode option. The **envmond** will not run as a daemon, and will instead run in the foreground, inheriting standard input and output. Error and warning messages will be written to the standard output instead of being logged via **syslogd**(1M)**.**

–f *file* Provides an alternate file path for the configuration file.

–g *granularity*
Defines the finest granularity for the poll interval. The default value is 10 seconds.

FILES

/usr/platform/SUNW,UltraSPARC–IIi–Netract/lib/envmond/sparcv9/envmond
The executable daemon

/usr/platform/SUNW,UltraSPARC–IIi–Netract/lib/envmond/sparcv9/*.so
The **envmond** policies

/platform/SUNW,UltraSPARC–IIi–Netract/lib/envmond.conf
The **envmond** configuration file

SEE ALSO

syslogd(1M), **envmond.conf**(4)

NOTES

The **envmond** policies retrieve their environmental information via I2C devices in the system.

This daemon is in the PROTOTYPE stage, and is therefore subject to CHANGE WITHOUT NOTICE.

NAME

hsi_init – set high speed serial line interface operating parameters.

SYNOPSIS

/opt/SUNWconn/bin/hsi_init *device* **[[** *baud_rate* **] | [** *keyword*=*value*, ... **] | [** *single-word option* **]]**

DESCRIPTION

The hsi_init utility allows the user to modify some of the hardware operating modes common to high speed synchronous serial lines. This may be useful in troubleshooting a link, or necessary to the operation of a communications package.

If run without options, hsi_init reports the options as presently set on the port. If options are specified, the new settings are reported after they have been made.

OPTIONS

Options to hsi_init normally take the form of a keyword, followed by an equal sign and a value. The exception is that a baud rate may be specified as a decimal integer by itself. Keywords must begin with the value shown in the options table, but may contain additional letters up to the equal sign. For example, "loop=" and "loopback=" are equivalent.

Recognized options are listed in the table below.

Keyword	**Value**	**Effect**
loopback	no	Disable internal loopback mode. If no other clocking options have been specified, perform the equivalent of **txc=txc** and **rxc=rxc**.
	yes	Set the port to operate in **internal loopback** mode. The receiver is electrically disconnected from the DCE receive data input and tied to the outgoing transmit data line. Transmit data is available to the DCE. If no other clocking options have been specified, perform the equivalent of **txc=baud** and **rxc=baud**.
nrzi	no	Set the port to operate with **NRZ** data encoding. NRZ encoding maintains a constant voltage level when data is present (1) and does not return to a zero voltage (0) until data is absent. The data is decoded as an absolute value based on the voltage level (0 or 1).
	yes	Set the port to operate with **NRZI** data encoding. **NRZI** encoding does a voltage transition when data is absent (0) and no voltage transition (no return to zero) when data is present (1). Hence, the name non-return to zero inverted. The data is decoded using relational decoding.
txc	txc	Transmit clock source will be the **TxCI** signal.
	-txc	Transmit clock source will be the inverted **TxCI** signal.
	rxc	Transmit clock source will be the **RxC** signal.
	baud	Transmit clock source will be the internal **baud rate generator**.

rxc	rxc	Receive clock source will be the **RxC** signal.
	-rxc	Receive clock source will be the inverted **RxC** signal.
	baud	Receive clock source will be the internal **baud rate generator**.
mode	fdx	HDLC Full Duplex mode (Default mode).
	ibm-fdx	IBM Full Duplex mode (SDLC).
	ibm-hdx	IBM Half Duplex mode (SDLC).
	ibm-mpt	IBM Multipoint mode (SDLC).
signal	yes	Notify application of modem signal (RTS and CTS) changes.
	no	Don't notify application of modem signal (RTS and CTS) changes.
speed	*integer*	Set the baud rate to *integer* bits per second. The speed can be set from 300 bps to 2048000 bps.
mtu		Set the Maximum Transmission Unit. This is the packet size that is transmitted. The maximum mtu is 1600 bytes.
mru		Set the Maximum Receive Unit. This is the packet size that is received. The maximum mru is 1600 bytes.
txd		This flags is used for inverting transmit data on serial lines. You can switch the polarity of a link by setting this flag to be negative, i.e. -txd.
rxd		This flags is used for inverting receive data on serial lines. You can switch the polarity of a link by setting this flag to be negative, i.e. -rxd.
reset		Resets the board. Terminates all incoming and outgoing traffic.

There are also several single-word options that set one or more paramaters at a time:

Keyword	**Equivalent to Options:**
external	txc=txc rxc=rxc loop=no
sender	txc=baud rxc=rxc loop=no
stop	speed=0

EXAMPLES

The following command sets the first CPU port to loop internally, use internal clocking and operate at 38400 bps:

```
example# hsi_init hih0 38400 loop=yes
port=hih0 speed=38309, mode=fdx, loopback=yes, nrzi=no, mtu=1600,
mru=1600, txc=baud, rxc=baud, txd=txd, rxd=rxd, signal=no.
```

The following command sets the same port's clocking, local loopback and bit rate settings to their default values:

```
example# hsi_init hih0 1536000 loop=no
port=hih0 speed=1536000, mode=fdx, loopback=no, nrzi=no, mtu=1600,
mru=1600, txc=txc, rxc=rxc, txd=txd, rxd=rxd, signal=no.
```

SEE ALSO

hsi_loop(1M), hsi_stat(1M), hsi_trace(1M), Intro(2), hsi(7D)

DIAGNOSTICS

device **missing minor device number**
: The name *device* does not end in a decimal number that can be used as a minor device number.

bad speed: *arg*
: The string *arg* that accompanied the "speed=" option could not be interpreted as a decimal integer.

Bad arg: *arg*
: The string *arg* did not make sense as an option.

ioctl failure code = *errno*
: An **ioctl(2)** system called failed. The meaning of the value of *errno* may be found in the **Intro(2)** manual page.

WARNINGS

hsi_init should not be used on an active serial link, unless needed to resolve an error condition. It should not be run casually, or if the user is unsure of the consequences of its use.

NAME

hsi_loop – high speed synchronous serial loopback test program for high speed serial interface.

SYNOPSIS

/opt/SUNWconn/bin/hsi_loop [–cdlsvt] *device*

DESCRIPTION

The hsi_loop command performs several loopback tests that are useful in exercising the various components of a serial communications link.

Before running a test, hsi_loop opens the designated port and configures it according to command line options and the specified test type. It announces the names of the devices being used to control the hardware channel, the channel number (ppa) corresponding to the *device* argument, and the parameters it has set for that channel. It then runs the loopback test in three phases.

The first phase is to listen on the port for any activity. If no activity is seen for at least four seconds, hsi_loop proceeds to the next phase. Otherwise, the user is informed that the line is active and that the test cannot proceed, and the program exits.

In the second phase, called the "first-packet" phase, hsi_loop attempts to send and receive one packet. The program will wait for up to four seconds for the returned packet. If no packets are seen after five attempts, the test fails with an error message. If a packet is returned, the result is compared with the original. If the length and content do not match exactly, the test fails.

The final phase, known as the "multiple-packet" phase, attempts to send many packets through the loop. Because the program has verified the integrity of the link in the first-packet phase, the test will not fail after a particular number of timeouts. If a packet is not seen after four seconds, a message is displayed. Otherwise, a count of the number of packets received is updated on the display once per second. If it becomes obvious that the test is not receiving packets during this phase, the user may wish to stop the program manually. The number and size of the packets sent during this phase is determined by default values, or by command line options. Each returned packet is compared with its original for length and content. If a mismatch is detected, the test fails. The test completes when the required number of packets have been sent, regardless of errors.

After the multiple-packet phase has completed, the program displays a summary of the hardware event statistics for the channel that was tested. The display takes the following form:

```
Port    CRC errors    Aborts    Overruns    Underruns    In <-Drops-> Out
hih0             0         0           0            0     0           0
```

This is followed by an estimated line speed, which is an approximation of the bit rate of the line, based on the number of bytes sent and the actual time that it took to send them. This is a very rough approximation and should not be used in bechmarking, because elapsed time includes time to print to the display.

OPTIONS

The options for hsi_loop are described in the following table:

Option	Parameter	Default	Description
–c	*packet_count*	100	Specifies the number of packets to be sent in the multiple-packet phase.
–d	*hex_data_byte*	*random*	Specifies that each packet will be filled with bytes with the value of *hex_data_byte*.
–l	*packet_length*	100	Specifies the length of each packet in bytes.
–s	*line_speed*	9600	Bit rate in bits per second.
–v			Sets verbose mode. If data errors occur, the expected and received data is displayed.
–t	*test_type*	*none*	A number, from 1 to 4, that specifies which test to perform. The values for *test_type* are as follows:

1 Internal loopback test. Port loopback is on. Transmit and receive clock sources are internal (baud rate generator).

2 External loopback test. Port loopback is off. Transmit and receive clock sources are internal. Requires a loopback plug suitable to the port under test.

3 External loopback test. Port loopback is off. Transmit and receive clock sources are external (modem). Requires that one of the local modem, the remote modem, or the remote system (not a Sun) be set in a loopback configuration.

4 Test using predefined parameters. User defines hardware configuration and may select port parameters using the **hsi_init(1M)** command.

All numeric options except **–d** are entered as decimal numbers (for example, **–s 19200**). If you do not provide the **–t** *test_type* option, hsi_loop prompts for it.

EXAMPLES

The following command causes hsi_loop to use a packet length of 512 bytes over the first CPU port:

```
example# hsi_loop –l 512 hih0
```

In response to the above command, hsi_loop prompts you for the test option you want.

The following command performs an internal loopback test on the first CPU port, using 5000 packets and a bit rate of 56Kbps :

```
example# hsi_loop –t 1 –s 56000 –c 5000 hih0
```

SEE ALSO

hsi_init(1M), hsi_stat(1M), hsi_trace(1M), hsi(7d)

DIAGNOSTICS

device **missing minor device number**
: The name *device* does not end in a decimal number that can be used as a minor device number.

invalid packet length: *nnn*
: The packet length was specified to be less than zero or greater than 1600.

poll: nothing to read
poll: nothing to read or write.
: The **poll(2)** system call indicates that there is no input pending and/or that output would be blocked if attempted.

len *xxx* **should be** *yyy*
: The packet that was sent had a length of *yyy*, but was received with a length of *xxx*.

nnn **packets lost in outbound queueing**
nnn **packets lost in inbound queueing**
: A discrepancy has been found between the number of packets sent by hsi_loop and the number of packets the driver counted as transmitted, or between the number counted as received and the number read by the program.

WARNINGS

To allow its tests to run properly, as well as prevent disturbance of normal operations, hsi_loop should only be run on a port that is not being used for any other purpose at that time.

NAME

hsi_stat – report driver statistics from a high speed synchronous serial link port.

SYNOPSIS

/opt/SUNWconn/bin/hsi_stat [-f] -a|*num_of_ports*
/opt/SUNWconn/bin/hsi_stat -c [-f] -a|*num_of_ports*
/opt/SUNWconn/bin/hsi_stat [-f] *device* **[***period***]**
/opt/SUNWconn/bin/hsi_stat -c [-f] *device*

DESCRIPTION

The hsi_stat command reports the event statistics maintained by a high speed synchronous serial device driver. The report may be a single snapshot of the accumulated totals, or a series of samples showing incremental changes.

Event statistics are maintained by a driver for each physical channel that it supports. They are initialized to zero at the time the driver module is loaded into the system when one of the driver's entry points is first called.

The **device** argument is the name of the high speed serial device as it appears in the /dev directory. For example, **hih0** specifies the first on-board high speed serial device.

As an alternative, you can display or clear the statistics for multiple physical channels using **num_of_ports** argument. The hsi_stat program will then display statistics accumulated from device **hih0** to **hih(num_of_ports - 1)**. Additionally, statistics for all ports can be displayed or cleared by the use of the -a option. In this case, the command will be issued for all the ports on the system. This option is not available for sampling purposes.

The following is a breakdown of hsi_stat output:

speed	The line speed the device has been set to operate at. It is the user's responsibility to make this value correspond to the modem clocking speed when clocking is provided by the modem.
ipkts	The total number of input packets.
opkts	The total number of output packets.
undrun	The number of transmitter underrun errors.
ovrrun	The number of receiver overrun errors.
abort	The number of aborted received frames.
crc	The number of received frames with CRC errors.
isize	The average size (in bytes) of input packets.
osize	The average size (in bytes) of output packets.
iutil	Reports the input line utilization expressed as a percentage.
outil	Reports the output line utilization expressed as a percentage.

Additional fields for the 'f' flag are listed below.

ierror	Reports the input error count. Errors can be incomplete frames, empty frames, or receive clock (RxC) problems.

inactiv	Reports the number of input packets received when receive is inactive.
ishort	Reports the number of short input packets. This is the number of input packets with lengths less than the number of CRC bytes.
ilong	Reports the number of long input packets. This is the number of input packets with lengths larger than the MRU.
oerror	Reports the output error count. Errors that can be lost are clear to send (CTS) signals or transmit clock (TxC) problems.
olong	Reports the number of long output packets. This is the number of output packets with lengths with lengths larger than the MTU.
ohung	Reports the number of times the transmitter hangs, which is usually due to a missing clock.

OPTIONS

–f Select full set of accumulated statistics for the device specified. This is useful while debugging the **hsi** driver.

–c Clear the accumulated statistics for the device specified. This may be useful when it is not desirable to unload a particular driver, or when the driver is not capable of being unloaded.

num_of_ports
Specify the number of devices that you want to dump the statistics.

–a Specify all of the ports in the system, regardless of the number of HSI boards.

interval Cause hsi_stat to sample the statistics every *interval* seconds and report incremental changes. The output reports line utilization for input and output in place of average packet sizes. These are the relationships between bytes transferred and the baud rate, expressed as percentages. The loop repeats indefinitely, with a column heading printed every twenty lines for convenience.

EXAMPLES

```
example# hsi_stat hih0
speed    ipkts    opkts    undrun    ovrrun    abort    crc    isize
 9600    15716    17121       0         0        1       3       98

example# hsi_stat 5
speed    ipkts    opkts    undrun    ovrrun    abort    crc    isize
 hih0     9600    15716     10100       0        0       1       3
 hih1     9600    15234     20100       0        0       1       3
 hih2     9600    15123     18254       0        0       1       3
 hih3     9600    15378     18234       0        0       1       3

example# hsi_stat -a
speed    ipkts    opkts    undrun    ovrrun    abort    crc    isize    osize
```

```
 hih0    9600   15716   10100       0        0      1      3       98
 hih1    9600   15234   20100       0        0      1      3       98
 hih2    9600   15123   18254       0        0      1      3       98
 hih3    9600   15378   18234       0        0      1      3       98
 hih4    9600   13900   13000       0        0      1      3       98
 hih5    9600   15218   13100       0        0      1      3       98
 hih6    9600   15737   22100       0        0      1      3       98
 hih7    9600   15143   11254       0        0      1      3       98
```

```
example# hsi_stat -c hih0
speed   ipkts   opkts   undrun   ovrrun   abort   crc   isize   osize
 9600     0       0       0        0        0      0      0       0
```

```
example# hsi_stat hih0 5
ipkts   opkts   undrun   ovrrun   abort   crc   iutil   outil
  12      10      0        0        0      0     5%      4%
  22      60      0        0        0      0     3%     90%
  36      14      0        0        0      1    51%      2%
```

(In this final example a new line of output is generated every five seconds.)

SEE ALSO

hsi_init(1M), hsi_loop(1M), hsi_trace(1M), hsi(7D)

DIAGNOSTICS

device **missing minor device number**

The name *device* does not end in a decimal number that can be used as a minor device number.

hsi_stat: Can't sample multiple ports simultaneously.

Sampling is only available with one specified port, i.e. hsi_stat hih0 10.

WARNINGS

Underrun, overrun, frame-abort and CRC errors have a variety of causes. Communication protocols are typically able to handle such errors and initiate recovery of the transmission in which the error occurred. Small numbers of such errors are not a significant problem for most protocols. However, because the overhead involved in recovering from a link error can be much greater than that of normal operation, high error rates can greatly degrade overall link throughput. High error rates are often caused by problems in the link hardware, such as cables, connectors, interface electronics or telephone lines. They may also be related to excessive load on the link or the supporting system.

The percentages for input and output line utilization reported when using the *interval* option may occasionally be reported as slightly greater than 100% because of inexact sampling times and differences in the accuracy between the system clock and the modem clock. If the percentage of use greatly exceeds 100%, or never exceeds 50%, then the baud rate set for the device probably does not reflect the speed of the modem.

NAME

hsi_trace – Dump and Parse the HSI/S driver trace buffer. This is a development/field support only diagnostic utility.

SYNOPSIS

/opt/SUNWconn/bin/hsi_trace

DESCRIPTION

hsi_trace utility id for support and field personnel only. This utility prints out the trace of the incoming and outgoing packets at the hsi driver level.

There are two levels of traces that can be captured. This is controlled by setting a variable in the driver in the /etc/system file.

set HSI:hsi_trace=1

The driver maintains an internal circular buffer to store 24K frames (both in and out).

Then run hsi_trace on the driver to collect the trace data.

# hsi_trace > hsi_trace.log

This trace is useful when the problem occurs rarely (typically a week or so) and we do not have enough file system space.

This trace collects the last 24K of frame data.

Then there is another trace ‘strace‘ which can be used to collect all the data from the driver. This can be enabled by setting ‘hsi_trace‘ as

set HSI:hsi_trace=2

Then run

#strace 18515 all all > hsi_trace.log

This collects all the data from the driver. This trace is useful when we know that the problem occurs within a short time.

The trace output is as follows

In the first case (‘hsi_trace‘ utility)

13:26:38 0000004f hih9 len=0100 R: 31323334 35363738 fm: I-FR P/F=1 Nr=1 Ns=1

The fields are as follows

1 st field: Time stamp

2 nd field: time difference in microsecs between the last frame and current frame.

3 rd field: port

4 th field: length of the frame.

5 th field: R: received data T: transmitted data

6 th and 7 th field: First 8 bytes of the data transmitted or received.

7 th field: The frame type (SABM, TEST, XID, RR, RNR....)

Some of the frame types are described below.

Keyword	**Value**	**Effect**
RR	Receive Ready	This frame is used as a polling command by the primary station to solicit information frames from the secondary station.
RNR	Receive Not Ready	This frame is used as a flow control command or response to indicate that the station transmitting the Receive Not Ready frame is not able to accept any information frames at this time.
REJ	Reject	This frame is sent by a station to indicate that it has received a frame out of the normal sequence. This may indicate the loss of an information frame containing user data.
SABM	Set Async Balanced Mode	An LLC non-data frame requesting the establishment of a connection over which numbered information frames may be sent.
SNRM	Set Normal Response Mode	This command is sent from the primary station to a secondary station to place the secondary in the initialized normal SDLC operating mode.
SNRME	SNRM Extended	SNRM with two more bytes in the control field. Used in SDLC.
DISC	Disconnect	This command is sent from the primary station to the secondary station to place the secondary station in the off-line disconnected mode.
SIM	Set Initialization Mode	This command is sent from the primary station to the secondary station to being the initialization process.
UA	Unnumbered Ack	This response is sent from the secondary station to the primary station in response to an SNRM, DISC, or SIM command.
DM	Disconnect Mode	This response is sent from the secondary station to the primary station in response to any command other than SNRM or DISC.
RD	Request Disconnect	This response is sent from the secondary to the primary station to request that the secondary station be placed in the off-line or disconnect mode.
RIM	Req Init Mode	This response is sent from the secondary to the primary station to request initialization.

FRMR	Frame Reject	This response is sent from the secondary station to the primary station to indicate that an abnormal condition has been detected or that an invalid frame has been received. It contains bits which indicate the reason for the rejection of the frame.
XID	Exchange Identification	This frame may be either a command sent by the primary station or a response sent by the secondary station. It contains information that is used to identify the secondary station.
TEST	TEST	This command is sent from the primary station to the secondary station and may contain some form of a message that may be used to test the seconary's ability to receive data and transmit the data back to the primary station.
UI	Unnumbered Inforation	This command allows the primary station to send data to the secondary station and the unnumbered information response allows the secondary station to send data to the primary station.
INFO	Information	This frame contains the information and data relevant to the higher SNA architecture layers. INFO frames consist of several variable-length or optional fields, depending upon the implementation.
UP	unnumbered Poll frame	Used by a primary to poll a secondary.
BCN	Beacon	This is a beacon frame which is usually an indication of a problem.
CFGR	Configure	This is a configuration frame.

'strace' is the normal unix strace output.

```
020809 13:34:31 001c1330  0 ... 18515 0 hih8 len=0100 T: 31323334 35363738 fm: I-FR
P/F=1 Nr=1 Ns=1
```

SEE ALSO

hsi_init(1M), hsi_stat(1M), hsi_loop(1M), hsi(7d)

DIAGNOSTICS

NAME

hsip_init – set high speed serial line interface operating parameters.

SYNOPSIS

/opt/SUNWconn/bin/hsip_init *device* **[[** *baud_rate* **] | [** *keyword***=***value***, ...] | [** *single-word option* **]]**

DESCRIPTION

The hsip_init utility allows the user to modify some of the hardware operating modes common to high speed synchronous serial lines. This may be useful in troubleshooting a link, or necessary to the operation of a communications package.

If run without options, hsip_init reports the options as presently set on the port. If options are specified, the new settings are reported after they have been made.

OPTIONS

Options to hsip_init normally take the form of a keyword, followed by an equal sign and a value. The exception is that a baud rate may be specified as a decimal integer by itself. Keywords must begin with the value shown in the options table, but may contain additional letters up to the equal sign. For example, "loop=" and "loopback=" are equivalent.

Recognized options are listed in the table below.

Keyword	**Value**	**Effect**
loopback	yes	Set the port to operate in **internal loopback** mode. The receiver is electrically disconnected from the DCE receive data input and tied to the outgoing transmit data line. Transmit data is available to the DCE. If no other clocking options have been specified, perform the equivalent of **txc=baud** and **rxc=baud**.
	no	Disable internal loopback mode. If no other clocking options have been specified, perform the equivalent of **txc=txc** and **rxc=rxc**.
	echo	Set the port to operate in **auto-echo** mode. The port will echo incoming receive data on the transmit data pin. When the loopback is set for echo and no clocking option is given the clocking is set txc=txc and rxc=rxc. Other clocking options can be used but line errors may occur due to the loopback=echo implementation.
nrzi	no	Set the port to operate with **NRZ** data encoding. **NRZ** encoding maintains a constant voltage level when data is present (1) and does not not return to a zero voltage (0) until data is absent. The data is decoded as an absolute value based on the voltage level (0 or 1).

	yes	Set the port to operate with **NRZI** data encoding. **NRZI** encoding does a voltage transition when data is absent (0) and no voltage transition (no return to zero) when data is present (1). Hence, the name non-return to zero inverted. The data is decoded using relational decoding.
txc	txc	Transmit clock source will be the **TxCI** signal.
	rxc	Transmit clock source will be the **RxC** signal.
	baud	Transmit clock source will be the internal **baud rate generator**.
	pll	Transmit clock source will be the output of the **DPLL** circuit. This can only be set with NRZI data encoding.
	-txc	Transmit clock source will be the inverted **TxCI** signal.
rxc	rxc	Receive clock source will be the **RxC** signal.
	txc	Receive clock source will be the **TxCI** signal. This can only be used with transmit clock option txc=txc.
	baud	Receive clock source will be the internal **baud rate generator**.
	pll	Receive clock source will be the output of the **DPLL** circuit. This can only be set with NRZI data encoding.
	-rxc	Receive clock source will be the inverted **RxC** signal.
txd	txd	Transmit data is not inverted.
	-txd	Transmit data is inverted.
rxd	rxd	Receive data is not inverted.
	-rxd	Receive data is inverted.
mode	fdx	HDLC Full Duplex mode (Default mode).
	ibm-fdx	IBM Full Duplex mode (SDLC).
	ibm-hdx	IBM Half Duplex mode (SDLC).
	ibm-mpt	IBM Multipoint mode (SDLC).
signal	yes	Notify application of modem signal (RTS and CTS) changes.
	no	Do not notify application of modem signal (RTS and CTS) changes.
mtu	*integer*	Set the maximum transmit unit to *integer* bytes with 2064 bytes maximum.
mru	*integer*	Set the maximum receive unit to *integer* bytes with 2064 bytes maximum.
speed	*integer*	Set the baud rate to *integer* bits per second with a minimum rate of 9600 bps and a maximum of 2048000 bps. Zero is also valid when txc is set to txc or -txc.

There are also several single-word options that set one or more paramaters at a time:

Keyword	**Equivalent to Options:**
external	txc=txc rxc=rxc loop=no
sender	txc=baud rxc=rxc loop=no
internal	txc=pll rxc=pll loop=no
stop	speed=0

EXAMPLES

The following command sets the first port to loop internally, use internal clocking and operate at 38400 baud:

```
example# hsip_init hihp0 38400 loop=yes
port=hihp0
speed=38400,
mode=fdx, signal=no, loopback=yes, nrzi=no, mtu=2064, mru=2064,
txc=baud, rxc=baud, txd=txd, rxd=rxd
```

The following command sets the same port’s clocking, local loopback and baud rate settings to their default values:

```
example# hsip_init hihp0 speed=1536000 loopback=no txc=txc rxc=rxc
port=hihp0
speed=1536000,
mode=fdx, signal=no, loopback=no, nrzi=no, mtu=2064, mru=2064,
txc=txc, rxc=rxc, txd=txd, rxd=rxd
```

SEE ALSO

hsip_loop(1M), hsip_stat(1M), Intro(2), hsip(7D)

DIAGNOSTICS

device **missing minor device number**
The name *device* does not end in a decimal number that can be used as a minor device number.

bad speed: *arg*
The string *arg* that accompanied the "speed=" option could not be interpreted as a decimal integer.

Bad arg: *arg*
The string *arg* did not make sense as an option.

ioctl failure code = *errno*
An **ioctl(2)** system called failed. The meaning of the value of *errno* may be found in the **Intro(2)** manual page.

WARNINGS

hsip_init should not be used on an active serial link, unless needed to resolve an error condition. It should not be run casually, or if the user is unsure of the consequences of its use.

NAME

hsip_loop – high speed synchronous serial loopback test program for high speed serial interface.

SYNOPSIS

/opt/SUNWconn/bin/hsip_loop [–cdlsvt] *device*

DESCRIPTION

The hsip_loop command performs several loopback tests that are useful in exercising the various components of a serial communications link.

Before running a test, hsip_loop opens the designated port and configures it according to command line options and the specified test type. It announces the names of the devices being used to control the hardware channel, the channel number (ppa) corresponding to the *device* argument, and the parameters it has set for that channel. It then runs the loopback test in three phases.

The first phase is to listen on the port for any activity. If no activity is seen for at least four seconds, hsip_loop proceeds to the next phase. Otherwise, the user is informed that the line is active and that the test cannot proceed, and the program exits.

In the second phase, called the "first-packet" phase, hsip_loop attempts to send and receive one packet. The program will wait for up to four seconds for the returned packet. If no packets are seen after five attempts, the test fails with an error message. If a packet is returned, the result is compared with the original. If the length and content do not match exactly, the test fails.

The final phase, known as the "multiple-packet" phase, attempts to send many packets through the loop. Because the program has verified the integrity of the link in the first-packet phase, the test will not fail after a particular number of timeouts. If a packet is not seen after four seconds, a message is displayed. Otherwise, a count of the number of packets received is updated on the display once per second. If it becomes obvious that the test is not receiving packets during this phase, the user may wish to stop the program manually. The number and size of the packets sent during this phase is determined by default values, or by command line options. Each returned packet is compared with its original for length and content. If a mismatch is detected, the test fails. The test completes when the required number of packets have been sent, regardless of errors.

After the multiple-packet phase has completed, the program displays a summary of the hardware event statistics for the channel that was tested. The display takes the following form:

```
Port    CRC errors   Aborts   Overruns   Underruns   In    <-Drops-> Out
hihp0            0        0          0           0    0                0
```

This is followed by an estimated line speed, which is an approximation of the bit rate of the line, based on the number of bytes sent and the actual time that it took to send them. This is a very rough approximation and should not be used in bechmarking, because elapsed time includes time to print to the display.

OPTIONS

The options for hsip_loop are described in the following table:

Option	Parameter	Default	Description
–c	*packet_count*	100	Specifies the number of packets to be sent in the multiple-packet phase.
–d	*hex_data_byte*	*random*	Specifies that each packet will be filled with bytes with the value of *hex_data_byte*.
–l	*packet_length*	100	Specifies the length of each packet in bytes with a maximum of 2064 bytes.
–s	*line_speed*	9600	Bit rate in bits per second, minimum of 9600 bps and a maximum of 2048000 bps.
–v			Sets verbose mode. If data errors occur, the expected and received data is displayed.
–t	*test_type*	*none*	A number, from 1 to 4, that specifies which test to perform. The values for *test_type* are as follows: **1** Internal loopback test. Port loopback is on. Transmit and receive clock sources are internal (baud rate generator). **2** External loopback test. Port loopback is off. Transmit and receive clock sources are internal. Requires a loopback plug suitable to the port under test. **3** External loopback test. Port loopback is off. Transmit and receive clock sources are external (modem). Requires that one of the local modem or the remote modem be set in a loopback configuration. **4** Test using predefined parameters. User defines hardware configuration and may select port parameters using the **hsip_init(1M)** command.

All numeric options except **–d** are entered as decimal numbers (for example, **–s 19200**). If you do not provide the **–t** *test_type* option, hsip_loop prompts for it.

EXAMPLES

The following command causes hsip_loop to use a packet length of 512 bytes over the first CPU port:

```
example# hsip_loop –l 512 hihp0
```

In response to the above command, hsip_loop prompts you for the test option you want.

The following command performs an internal loopback test on the first CPU port, using 5000 packets and a bit rate of 56000 bps :

```
example# hsip_loop –t 1 –s 56000 –c 5000 hihp0
```

SEE ALSO

hsip_init(1M), hsip_stat(1M), hsip(7D)

DIAGNOSTICS

device **missing minor device number**
The name *device* does not end in a decimal number that can be used as a minor device number.

invalid packet length: *nnn*
The packet length was specified to be less than zero or greater than 2064.

poll: nothing to read
poll: nothing to read or write.
The **poll(2)** system call indicates that there is no input pending and/or that output would be blocked if attempted.

len *xxx* **should be** *yyy*
The packet that was sent had a length of *yyy*, but was received with a length of *xxx*.

nnn **packets lost in outbound queueing**
nnn **packets lost in inbound queueing**
A discrepancy has been found between the number of packets sent by hsip_loop and the number of packets the driver counted as transmitted, or between the number counted as received and the number read by the program.

WARNINGS

To allow its tests to run properly, as well as prevent disturbance of normal operations, hsip_loop should only be run on a port that is not being used for any other purpose at that time.

NAME

hsip_stat – report driver statistics from a high speed synchronous serial link port.

SYNOPSIS

/opt/SUNWconn/bin/hsip_stat [-f] -a|*num_of_ports*
/opt/SUNWconn/bin/hsip_stat [-f] *device* **[***period***]**
/opt/SUNWconn/bin/hsip_stat -c [-f] -a|*num_of_ports*
/opt/SUNWconn/bin/hsip_stat -c [-f] *device*

DESCRIPTION

The hsip_stat command reports the event statistics maintained by a high speed synchronous serial device driver. The report may be a single snapshot of the accumulated totals, or a series of samples showing incremental changes.

Event statistics are maintained by a driver for each physical channel that it supports. They are initialized to zero at the time the driver module is loaded into the system when one of the driver's entry points is first called.

The **device** argument is the name of the high speed serial device as it appears in the /dev directory. For example, **hihp0** specifies the first on-board high speed serial device.

As an alternative, you can display or clear the statistics for multiple physical channels using **num_of_ports** argument. The hsip_stat program will then display statistics accumulated for the first **n** number of ports, where **n** is **num_of_ports**.

The following is a breakdown of hsip_stat output:

speed	The line speed the device has been set to operate at. It is the user's responsibility to make this value correspond to the modem clocking speed when clocking is provided by the modem.
ipkts	The total number of input packets.
opkts	The total number of output packets.
undrun	The number of transmitter underrun errors.
ovrrun	The number of receiver overrun errors.
abort	The number of aborted received frames.
crc	The number of received frames with CRC errors.
isize	The average size (in bytes) of input packets.
osize	The average size (in bytes) of output packets.
ierror	Input error count (errors: Incomplete Frame, Empty frame, Glitch on RxC).
oerror	Output error count (errors: CTS lost, Glitch on TxC).
iutil	Input line utilization expressed as a percentage.
outil	Output line utilization expressed as a percentage.

OPTIONS

−f Select a complete set of accumulated statistics for the device specified. This is useful while debugging the **hsip** driver.

−a Select all devices.

−c Clear the accumulated statistics for the device specified. This may be useful when it is not desirable to unload a particular driver, or when the driver is not capable of being unloaded.

num_of_ports
Specify the number of devices that you want to dump the statistics.

period Cause hsip_stat to sample the statistics every *period* seconds and report incremental changes. The output reports line utilization for input and output in place of average packet sizes. These are the relationships between bytes transferred and the speed, expressed as percentages. The loop repeats indefinitely, with a column heading printed every twenty lines for convenience.

EXAMPLES

```
example# hsip_stat hihp0
speed   ipkts   opkts   undrun   ovrrun   abort   crc   isize   osize
 9600   15716   17121      0        0       1      3     98      89

example# hsip_stat 5
        speed   ipkts   opkts   undrun   ovrrun   abort   crc   isize   osize
hihp0    9600   15716   10100      0        0       1      3     98      89
hihp1    9600   15234   20100      0        0       1      3     98      89
hihp2    9600   15123   18254      0        0       1      3     98      89
hihp3    9600   15378   18234      0        0       1      3     98      89
hihp4    9600   13900   13000      0        0       1      3     98      89

example# hsip_stat -c hihp0
speed   ipkts   opkts   undrun   ovrrun   abort   crc   isize   osize
 9600     0       0       0        0        0      0      0       0

example# hsip_stat hihp0 5
ipkts   opkts   undrun   ovrrun   abort   crc   iutil   outil
  12      10       0        0       0      0     5%      4%
  22      60       0        0       0      0     3%     90%
  36      14       0        0       0      1    51%      2%
```

(In this final example a new line of output is generated every five seconds.)

SEE ALSO

hsip_init(1M), hsip_loop(1M), hsip(7D)

DIAGNOSTICS

bad interval: *arg*
The argument *arg* is expected to be an interval and could not be understood.

device **missing minor device number**

The name *device* does not end in a decimal number that can be used as a minor device number.

WARNINGS

Underrun, overrun, frame-abort and CRC errors have a variety of causes. Communication protocols are typically able to handle such errors and initiate recovery of the transmission in which the error occurred. Small numbers of such errors are not a significant problem for most protocols. However, because the overhead involved in recovering from a link error can be much greater than that of normal operation, high error rates can greatly degrade overall link throughput. High error rates are often caused by problems in the link hardware, such as cables, connectors, interface electronics or telephone lines. They may also be related to excessive load on the link or the supporting system.

The percentages for input and output line utilization reported when using the *interval* option may occasionally be reported as slightly greater than 100% because of inexact sampling times and differences in the accuracy between the system clock and the modem clock. If the percentage of use greatly exceeds 100%, or never exceeds 50%, then the baud rate set for the device probably does not reflect the speed of the modem.

NAME

nf_fddidaemon – start/stop the NF FDDI SMT/SNM daemon and its associated processes.

SYNOPSIS

nf_fddidaemon start | stop

AVAILABILITY

This command is available with the *SunFDDI* product.

DESCRIPTION

The **nf_fddidaemon** script starts/stops the SNM daemon and its associated processes.

OPTIONS

start	Starts the SNM daemon
stop	Stops the SNM daemon

You must be root to run this command.

SEE ALSO

nf_snmd (1M)

NAME

nf_install_agents – install SunNet Manager agents for SunFDDI

SYNOPSIS

nf_install_agents

AVAILABILITY

This command is available with the *SunFDDI* product.

DESCRIPTION

The **nf_install_agents** script copies the FDDI schema files to the directory in which the standard agents are installed and updates the configuration files for SunNet Manager

The **nf_install_agents** command takes no arguments.

You must be root to run this command.

SEE ALSO

nf_snmd (1M)

NAME

nf_macid – obtain MAC address from specified **nf** (SunFDDI) interface.

SYNOPSIS

nf_macid *interface*

AVAILABILITY

This command is available only with the *SunFDDI* product.

DESCRIPTION

This command queries the IDPROM on the SunFDDI SBus card associated with a **nf** interface to obtain the MAC address resident there. This address is a globally unique, 48-bit address that is drawn from the same pool from which Ethernet addresses are taken.

The **nf_macid** command does not allow you to set a MAC address, either on the SBus card or for an interface. Use **ifconfig** with the **ether** argument to assign the MAC address you obtain with **nf_macid** to an SunFDDI interface.

Normally, you use the host-resident MAC address for all network interfaces on a machine. You would only use the MAC address obtained with **nf_macid** under unusual circumstances.

You can be normal user (not root) to run this command.

OPTIONS

interface	Specifies the FDDI interface (**nf**<*num*>). The default (which you can omit) is **nf0.**

EXAMPLE

Obtain the MAC address for **nf0:**

```
% nf_macid
8:0:20:3e:da:5
```

Set the **nf0** interface to have the MAC address in the SBus card IDPROM:

```
# ifconfig nf0 ether ‘nf_macid‘
```

You would follow the preceding command with an **ifconfig** command to assign an IP address to **nf0** and bring up that interface. Normally, such **ifconfig** commands would be run from a startup file.

SEE ALSO

ifconfig (1M)

NAME

nf_smtmon – the SMT monitor.

SYNOPSIS

nf_smtmon [–**i** *interface*] [**-x**] [**-h**] [*frametype*]

AVAILABILITY

This command is available with the *SunFDDI* product.

DESCRIPTION

nf_smtmon is used to display received SMT frames. You should run this command on the FDDI proxy system if the Console does not receive a response from a request for SMT MIB information.

You must be root to run this command.

OPTIONS

–**i** *interface*	Specifies the FDDI interface (**nf***num* for SunFDDI). If this option is not specified, frames for all FDDI interfaces are displayed.
-x	Displays the received frames in hex.
-h	Displays the usage of this command.
frametype	Specifies one or more types of SMT frames to be displayed. If this option is not specified, all types of frames are displayed. You can specify the following types of frames to be displayed:

ecf	Echo Frame. Request and response frames are used for SMT-to-SMT loopback testing on an FDDI ring.
esf	Extended Service Frame. Request, response, and announcement frames are used to extend new SMT services.
nif	Neighborhood Information Frame. Request, response, and announcement frames are used to communicate station addresses and descriptions.
pmf_get	Parameter Management Frame (PMF) Get Request. Request and response frames are used to retrieve SMT Management Information Base (MIB) attribute values.
rdf	Request Denied Frame (response only). Sent in response to an unsupported or unknown request.
sifconfig	Status Information Frame (SIF) Configuration. Request and response frames are used to retrieve configuration parameters for one or more stations on the ring.
sifoperation	Status Information Frame (SIF) Operation. Request and response frames are used to retrieve operation information for one or more stations on the ring.
srf	Status Report Frame. Announcement frame used to report Station Status. The current version of the SMT

daemon does not send out SRFs; however, any received SRFs are passed on to SNM as traps.

EXAMPLES

nf_smtmon -i nf0 nif sifconfig
displays the NIF and SIF configuration frames received in non-hex format on the **nf0** (SunFDDI) interface.

nf_smtmon -i nf1 -x ecf
displays, in hex, ECF frames received on the **nf1** (SunFDDI) interface.

SEE ALSO

smtd (1M)

NAME

nf_snmd – start the station management (SMT) to SunNet Manager daemon.

SYNOPSIS

nf_snmd [–**d**] [**-v5**]

AVAILABILITY

This command is available with the *SunFDDI* product.

DESCRIPTION

Upon invocation, the SNM daemon starts up station management processes that allow the station to communicate with other stations using the SMT protocol, and collect and return FDDI statistics to a SunNet Manager (SNM) Console. The daemon also receives SMT requests and SMT responses. The daemon also sends out SMT requests to other stations on the ring on behalf of SNM. The SMT daemon also forwards received Status Report Frames (SRFs) to the SNM management station in the form of traps.

The processes started by the SNM daemon include two SNM agents: a local agent (fddi) and a proxy agent (fddismt). Like other SNM agents, the local agent and proxy agent communicate with the SNM management station using RPC. The local agent responds to SNM requests with FDDI statistics gathered on the local machine. These statistics are equivalent to those displayed with the **nf_stat** and **nf_stat -m** commands.

The proxy agent can return two types of SMT information to the SNM Console: actual SMT frames (ECF, ESF, NIF, SIF Configuration, or SIF Operation), and attribute values for selected SMT MIB groups. The proxy agent gathers information from target stations by issuing SMT request frames and receiving SMT response frames. The proxy uses PMF Get request and response frames to retrieve MIB attribute values from the target station.

If the target station does not support PMF Get frames, it returns an RDF response to the proxy system. If a Console request for MIB attributes values is not successful, run the SMT monitor on the proxy system to see if an RDF frame has been received from the target station. If PMF Get frames are not supported by the target station, you may be able to use NIF, SIF Configuration or SIF Operation frames to return the desired attribute values.

The SMT MIB attributes groups MAC, PATH, and PORT contain index parameters. If you send a Quick Dump request from the Console for attribute values from one of these groups, only the values associated with the first index are returned (from the Console's point of view, the key value associated with the request is 1). If you want to see attribute values associated with other indexes, you must send a Data Report request with the Key field in the request set to the desired index.

If you make any changes to the **/etc/opt/snm/snm.conf** file on the station (for example, you add an additional hostname to the **na.fddi.trap-rendez** entry), you must kill the SNM daemon with **nf_snmd_kill** and then restart it in order for the change(s) to take effect.

You must be root to run this command.

OPTIONS

-d (debug mode) Displays a one-line entry in the window where **nf_snmd** is started for each frame that the station sends or receives. If this option is not specified, you are returned to the system prompt and there is no display. Use of this option is not recommended if the **nf_snmd** command is included in **/etc/rc2.d/S98nf_fddidaemon .**

SEE ALSO

nf_snmd_kill (1M), nf_stat (1M)

NAME

nf_snmd_kill – kill the station management (SMT) to SunNet Manager daemon and its associated processes.

SYNOPSIS

nf_snmd_kill

AVAILABILITY

This command is available with the *SunFDDI* product.

DESCRIPTION

The **nf_snmd_kill** script kills the SNM daemon and its associated processes. This command also kills the two SNM agents which are started by the SNM daemon: the local agent (fddi) and the proxy agent (fddismt). This command should not be used if the SNM daemon is not already running.

The **nf_snmd_kill** command takes no arguments.

You must be root to run this command.

SEE ALSO

nf_snmd (1M)

NAME

nf_stat – display SunFDDI interface statistics.

SYNOPSIS

nf_stat [–**m**][*interface*][*interval*][*count*]

AVAILABILITY

This command is available with the *SunFDDI* product.

DESCRIPTION

The **nf_stat** utility displays statistics for the SunFDDI interface. Some statistics relate to the SunFDDI implementation of the ANSI FDDI Connection Management standard (CMT), while others contain packet throughput, or station neighbor information.

This utility can report, on a periodic basis, packet throughput statistics, reconfiguration events, and interface exceptions. It also reports the identity of neighboring stations, information on its PHYs, and some FORMAC error counters. Several of the counters and status variables are periodically passed to the host from the hardware during the heartbeat signal. These statistics are available when invoking the command without the *-m* option. Issuing the command without an *interval* value displays the accumulated statistics; issuing the command with an *interval* value displays any differences between values since the previous display.

OPTIONS

–**m**	Dumps the current nearest neighbor information and FDDI/S timer settings (described below). The *interval* and *count* arguments have no effect when used with this option. Note that you must be root to invoke **nf_stat** with the **-m** option.
interface	Specifies which SunFDDI interface, **nf***num.*
interval	Specifies the interval in seconds at which to display the statistics.
count	Specifies the number of times to display the statistics. If no count is provided, the utility runs forever. It can be terminated by typing ˆC (Control-C).

USAGE

You invoke **nf_stat** *with* the **-m** option to display information about neighboring stations. It generates a columnar display containing the following categories of data:

PhyA	On a machine running SunFDDI Dual, shows the PHY type of the neighboring station that is connected to PHYA. Values are **A**, **B**, **S**, **M**, and **None** (if no connection). This column does not appear on a machine running SunFDDI SAS - Single Attached Station. (See Chapter 9 of the document ANSI/FDDI Station Management (SMT) Rev7.2 (25 June 1992)).
PhyB	On a machine running SunFDDI Dual, shows the PHY type of the neighboring station that is connected to PHYB. Values are **A**, **B**, **S**, **M**, and **None** (if no connection). This column does not appear on a machine running SunFDDI SAS. (See Chapter 7 of the document ANSI/FDDI Station Management (SMT) Rev7.2 (25 June 1992)).
PhyS	On a machine running SunFDDI SAS, shows the PHY type of the neighboring station that is connected to PHYS. Values are **A**, **B**, **S**, **M**, and **None** (if no connection). If connected to a concentrator, this will be **M.** This column

does not appear on a machine running SunFDDI Dual.

Frame	FDDI MAC standard counter, frames received.
Error	FDDI MAC standard counter, frame with the E bit first detected at this station.
Lost	Frames whose reception is aborted.
SA	MAC address; the unique 48-bit address of the SunFDDI interface. Where an IP hostname exists, it is displayed; otherwise, the 48-bit MAC address is used.
UNA	The address of this station's upstream neighbor, using the SMT NIF protocol.
DNA	The address of this station's downstream neighbor, using the SMT NIF protocol.

Display status information : You invoke **nf_stat** without the *-m* option, or with values for *interface* or *interval*, to display status information. Issuing the command without an *interval* value displays the accumulated statistics; issuing the command with an *interval* value displays any differences between values since the previous display.

One use of **nf_stat** without the **-m** option is to monitor the **Ring_OP** (Ring Operational) column; if it indicates more than one ring_op per second, there are media problems that must be fixed.

When invoked without the **-m** option, **nf_stat** generates a columnar display containing the following categories of data:

Ring	Indicates whether the ring is up or down (that is, the Claim has succeeded). **Note:** The following five fields use terms described in the SMT document, Chapter 9.
ECM	(**ec_state**). Shows the current state of the ECM state machine. Valid values are: **Out**, **In**, **Trace**, **Leave**, **Path_Test**, **Insert**, **Check**, and **Deinsert**.
RMT	(**rmt_state**). Shows the current state of the RMT state machine. Valid values are: **Isolated**, **Non_Op**, **Ring_Op**, **Detect**, **Non_Op_Dup**, **Ring_Op_Dup**, **Directed**, and **Rm_Trace**.
PCMA/PCMB (for SunFDDI Dual) **PCMS** (for SunFDDI SAS)	(**pc_state**). Is a variable from PCM to other management entities containing the current state of the PCM state machine. Current valid values are: **Off (O)**, **Break (B)**, **Reject (R)**, **Connect (C)**, **Next (N)**, **Signal (S)**, **Join (J)**, **Verify (V)**, **Active (A)**, and **Maint (M)**.
Ring_OP	(**Ring Operational**). Indicates the number of times the ring has come up (and therefore implies the number of times the ring has gone down).
XmitP	The number of packets transmitted.
RecvP	The number of packets received.

SEE ALSO **netstat (1M)**

NAME

nf_sync – configure SunFDDI interface to operate in synchronous mode.

SYNOPSIS

nf_sync *nf<inst>* [*tsync sap*]

AVAILABILITY

This command is available with the *SunFDDI* product.

DESCRIPTION

The **nf_sync** utility is used to configure SunFDDI interfaces to operate in synchronous mode. By default, the SunFDDI interface configure to carry asynchronous traffic only.

OPTIONS

nf<inst>	Specifies the FDDI interface,
tsync	Specifies synchronous timer in nanoseconds, 400000 nanoseconds minimum,
sap	Specifies the service access point (SAP) for synchronous operation.

USAGE

Running nf_sync without specifying values for *tsync* and *sap* returns current configuration of the interface.

To reconfigure SAP for asynchronous operations, specify tsync=0

EXAMPLES

nf_sync nf0
displays current configuration on the **nf0** (SunFDDI) interface.

nf_sync nf0 1000000 800
configures SAP 800 for synchronous operation with a clock rate 1000000 nanoseconds (1ms)

NAME

pf_fddidaemon – start/stop the PF FDDI SMT/SNM daemon and its associated processes.

SYNOPSIS

pf_fddidaemon start | stop

AVAILABILITY

This command is available with the *SunFDDI* product.

DESCRIPTION

The **pf_fddidaemon** script starts/stops the SNM daemon and its associated processes.

OPTIONS

start	Starts the SNM daemon
stop	Stops the SNM daemon

You must be root to run this command.

SEE ALSO

pf_snmd (1M)

NAME

pf_install_agents – install SunNet Manager agents for SunFDDI

SYNOPSIS

pf_install_agents

AVAILABILITY

This command is available with the *SunFDDI* product.

DESCRIPTION

The **pf_install_agents** script copies the FDDI schema files to the directory in which the standard agents are installed and updates the configuration files for SunNet Manager

The **pf_install_agents** command takes no arguments.

You must be root to run this command.

SEE ALSO

pf_snmd (1M)

NAME

pf_macid – obtain MAC address from specified **pf** (SunFDDI/P) interface.

SYNOPSIS

pf_macid *interface*

AVAILABILITY

This command is available only with the *SunFDDI* product.

DESCRIPTION

This command queries the IDPROM on the SunFDDI card associated with a **pf** interface to obtain the MAC address resident there. This address is a globally unique, 48-bit address that is drawn from the same pool from which Ethernet addresses are taken.

The **pf_macid** command does not allow you to set a MAC address, either on the PCI card or for an interface. Use **ifconfig** with the **ether** argument to assign the MAC address you obtain with **pf_macid** to an SunFDDI interface.

Normally, you use the host-resident MAC address for all network interfaces on a machine. You would only use the MAC address obtained with **pf_macid** under unusual circumstances.

You can be normal user (not root) to run this command.

OPTIONS

interface	Specifies the FDDI interface (**pf**<*num*>). The default (which you can omit) is **pf0.**

EXAMPLE

Obtain the MAC address for **pf0:**

```
% pf_macid
8:0:20:3e:da:5
```

Set the **pf0** interface to have the MAC address in the PCI card IDPROM:

```
# ifconfig pf0 ether ‘pf_macid‘
```

You would follow the preceding command with an **ifconfig** command to assign an IP address to **pf0** and bring up that interface. Normally, such **ifconfig** commands would be run from a startup file.

SEE ALSO

ifconfig (1M)

NAME

pf_smtmon – the SMT monitor.

SYNOPSIS

pf_smtmon [–**i** *interface*] [**-x**] [**-h**] [*frametype*]

AVAILABILITY

This command is available with the *SunFDDI/P* product.

DESCRIPTION

pf_smtmon is used to display received SMT frames. You should run this command on the FDDI proxy system if the Console does not receive a response from a request for SMT MIB information.

You must be root to run this command.

OPTIONS

–**i** *interface*	Specifies the FDDI interface (**pf***num* for SunFDDI / P). If this option is not specified, frames for all FDDI interfaces are displayed.
-x	Displays the received frames in hex.
-h	Displays the usage of this command.
frametype	Specifies one or more types of SMT frames to be displayed. If this option is not specified, all types of frames are displayed. You can specify the following types of frames to be displayed:

ecf	Echo Frame. Request and response frames are used for SMT-to-SMT loopback testing on an FDDI ring.
esf	Extended Service Frame. Request, response, and announcement frames are used to extend new SMT services.
nif	Neighborhood Information Frame. Request, response, and announcement frames are used to communicate station addresses and descriptions.
pmf_get	Parameter Management Frame (PMF) Get Request. Request and response frames are used to retrieve SMT Management Information Base (MIB) attribute values.
rdf	Request Denied Frame (response only). Sent in response to an unsupported or unknown request.
sifconfig	Status Information Frame (SIF) Configuration. Request and response frames are used to retrieve configuration parameters for one or more stations on the ring.
sifoperation	Status Information Frame (SIF) Operation. Request and response frames are used to retrieve operation information for one or more stations on the ring.
srf	Status Report Frame. Announcement frame used to report Station Status. The current version of the SMT

daemon does not send out SRFs; however, any received SRFs are passed on to SNM as traps.

EXAMPLES

pf_smtmon -i pf0 nif sifconfig
displays the NIF and SIF configuration frames received in non-hex format on the **pf0** (SunFDDI/P) interface.

pf_smtmon -i pf1 -x ecf
displays, in hex, ECF frames received on the **pf1** (SunFDDI/P) interface.

SEE ALSO

smtd (1M)

NAME

pf_snmd – start the station management (SMT) to SunNet Manager daemon.

SYNOPSIS

pf_snmd [–**d**] [**-v5**]

AVAILABILITY

This command is available with the *SunFDDI/P* product.

DESCRIPTION

Upon invocation, the SNM daemon starts up station management processes that allow the station to communicate with other stations using the SMT protocol, and collect and return FDDI statistics to a SunNet Manager (SNM) Console. The daemon also receives SMT requests and SMT responses. The daemon also sends out SMT requests to other stations on the ring on behalf of SNM. The SMT daemon also forwards received Status Report Frames (SRFs) to the SNM management station in the form of traps.

The processes started by the SNM daemon include two SNM agents: a local agent (fddi) and a proxy agent (fddismt). Like other SNM agents, the local agent and proxy agent communicate with the SNM management station using RPC. The local agent responds to SNM requests with FDDI statistics gathered on the local machine. These statistics are equivalent to those displayed with the **pf_stat** and **pf_stat -m** commands.

The proxy agent can return two types of SMT information to the SNM Console: actual SMT frames (ECF, ESF, NIF, SIF Configuration, or SIF Operation), and attribute values for selected SMT MIB groups. The proxy agent gathers information from target stations by issuing SMT request frames and receiving SMT response frames. The proxy uses PMF Get request and response frames to retrieve MIB attribute values from the target station.

If the target station does not support PMF Get frames, it returns an RDF response to the proxy system. If a Console request for MIB attributes values is not successful, run the SMT monitor on the proxy system to see if an RDF frame has been received from the target station. If PMF Get frames are not supported by the target station, you may be able to use NIF, SIF Configuration or SIF Operation frames to return the desired attribute values.

The SMT MIB attributes groups MAC, PATH, and PORT contain index parameters. If you send a Quick Dump request from the Console for attribute values from one of these groups, only the values associated with the first index are returned (from the Console's point of view, the key value associated with the request is 1). If you want to see attribute values associated with other indexes, you must send a Data Report request with the Key field in the request set to the desired index.

If you make any changes to the **/etc/opt/snm/snm.conf** file on the station (for example, you add an additional hostname to the **na.fddi.trap-rendez** entry), you must kill the SNM daemon with **pf_snmd_kill** and then restart it in order for the change(s) to take effect.

You must be root to run this command.

OPTIONS

-d (debug mode) Displays a one-line entry in the window where **pf_snmd** is started for each frame that the station sends or receives. If this option is not specified, you are returned to the system prompt and there is no display. Use of this option is not recommended if the **pf_snmd** command is included in **/etc/rc2.d/S98pf_fddidaemon .**

SEE ALSO

pf_snmd_kill (1M), pf_stat (1M)

NAME

pf_snmd_kill – kill the station management (SMT) to SunNet Manager daemon and its associated processes.

SYNOPSIS

pf_snmd_kill

AVAILABILITY

This command is available with the *SunFDDI/P* product.

DESCRIPTION

The **pf_snmd_kill** script kills the SNM daemon and its associated processes. This command also kills the two SNM agents which are started by the SNM daemon: the local agent (fddi) and the proxy agent (fddismt). This command should not be used if the SNM daemon is not already running.

The **pf_snmd_kill** command takes no arguments.

You must be root to run this command.

SEE ALSO

pf_snmd (1M)

NAME

pf_stat – display SunFDDI/P interface statistics.

SYNOPSIS

pf_stat [–**m**][*interface*][*interval*][*count*]

AVAILABILITY

This command is available with the *SunFDDI/P* product.

DESCRIPTION

The **pf_stat** utility displays statistics for the SunFDDI/P interface. Some statistics relate to the SunFDDI/P implementation of the ANSI FDDI Connection Management standard (CMT), while others contain packet throughput, or station neighbor information.

This utility can report, on a periodic basis, packet throughput statistics, reconfiguration events, and interface exceptions. It also reports the identity of neighboring stations, information on its PHYs, and some FORMAC error counters. Several of the counters and status variables are periodically passed to the host from the hardware during the heartbeat signal. These statistics are available when invoking the command without the *-m* option. Issuing the command without an *interval* value displays the accumulated statistics; issuing the command with an *interval* value displays any differences between values since the previous display.

OPTIONS

–**m**	Dumps the current nearest neighbor information and FDDI/S timer settings (described below). The *interval* and *count* arguments have no effect when used with this option. Note that you must be root to invoke **pf_stat** with the **-m** option.
interface	Specifies which SunFDDI/P interface, **pf***num.*
interval	Specifies the interval in seconds at which to display the statistics.
count	Specifies the number of times to display the statistics. If no count is provided, the utility runs forever. It can be terminated by typing ^C (Control-C).

USAGE

You invoke **pf_stat** *with* the **-m** option to display information about neighboring stations. It generates a columnar display containing the following categories of data:

PhyA On a machine running SunFDDI/P Dual, shows the PHY type of the neighboring station that is connected to PHYA. Values are **A**, **B**, **S**, **M**, and **None** (if no connection). This column does not appear on a machine running SunFDDI/P SAS - Single Attached Station. (See Chapter 9 of the document ANSI/FDDI Station Management (SMT) Rev7.2 (25 June 1992)).

PhyB On a machine running SunFDDI/P Dual, shows the PHY type of the neighboring station that is connected to PHYB. Values are **A**, **B**, **S**, **M**, and **None** (if no connection). This column does not appear on a machine running SunFDDI/P SAS. (See Chapter 7 of the document ANSI/FDDI Station Management (SMT) Rev7.2 (25 June 1992)).

PhyS On a machine running SunFDDI/P SAS, shows the PHY type of the neighboring station that is connected to PHYS. Values are **A**, **B**, **S**, **M**, and **None**

(if no connection). If connected to a concentrator, this will be **M.** This column does not appear on a machine running SunFDDI/P Dual.

Frame FDDI MAC standard counter, frames received.

Error FDDI MAC standard counter, frame with the E bit first detected at this station.

Lost Frames whose reception is aborted.

SA MAC address; the unique 48-bit address of the SunFDDI/P interface. Where an IP hostname exists, it is displayed; otherwise, the 48-bit MAC address is used.

UNA The address of this station's upstream neighbor, using the SMT NIF protocol.

DNA The address of this station's downstream neighbor, using the SMT NIF protocol.

Display status information : You invoke **pf_stat** without the *-m* option, or with values for *interface* or *interval*, to display status information. Issuing the command without an *interval* value displays the accumulated statistics; issuing the command with an *interval* value displays any differences between values since the previous display.

One use of **pf_stat** without the **-m** option is to monitor the **Ring_OP** (Ring Operational) column; if it indicates more than one ring_op per second, there are media problems that must be fixed.

When invoked without the **-m** option, **pf_stat** generates a columnar display containing the following categories of data:

Ring Indicates whether the ring is up or down (that is, the Claim has succeeded).

Note: The following five fields use terms described in the SMT document, Chapter 9.

ECM (**ec_state**). Shows the current state of the ECM state machine. Valid values are: **Out**, **In**, **Trace**, **Leave**, **Path_Test**, **Insert**, **Check**, and **Deinsert**.

RMT (**rmt_state**). Shows the current state of the RMT state machine. Valid values are: **Isolated**, **Non_Op**, **Ring_Op**, **Detect**, **Non_Op_Dup**, **Ring_Op_Dup**, **Directed**, and **Rm_Trace**.

PCMA/PCMB (for SunFDDI/P Dual) **PCMS** (for SunFDDI/P SAS)
(**pc_state**). Is a variable from PCM to other management entities containing the current state of the PCM state machine. Current valid values are: **Off (O)**, **Break (B)**, **Reject (R)**, **Connect (C)**, **Next (N)**, **Signal (S)**, **Join (J)**, **Verify (V)**, **Active (A)**, and **Maint (M)**.

Ring_OP (**Ring Operational**). Indicates the number of times the ring has come up (and therefore implies the number of times the ring has gone down).

XmitP The number of packets transmitted.

RecvP The number of packets received.

SEE ALSO **netstat (1M)**

NAME

rscadm – administer SUN(tm) Remote System Control (RSC)

SYNOPSIS

rscadm help
rscadm resetrsc [-s]
rscadm set *variable value*
rscadm download [boot] *file*
rscadm show [variable]
rscadm date [-s] | [[mmdd]HHMM | mmddHHMM[cc]yy][.SS]
rscadm send_event [-c] *message*
rscadm modem_setup
rscadm useradd *username*
rscadm userdel *username*
rscadm usershow [username]
rscadm userpassword *username*
rscadm userperm *username* [cuar]

DESCRIPTION

rscadm administers the SUN(tm) Remote System Console (RSC). It allows the host server to interact with the RSC. The following operations are supported:

rscadm help
: Displays a usage screen.

rscadm resetrsc
: Reset the RSC. There are two types of reset allowed, a "hard" reset and a "soft" reset. The hard reset is done by default. The soft reset can be selected by using the -s option.

rscadm set
: Set RSC configuration variables. Examples of RSC configuration variables include RSC IP address and RSC hostname. See the RSC documentation for a complete list of RSC configuration variables.

rscadm download
: Program the RSC's firmware. There are two parts to the firmware, the boot monitor and the main image. By default, rscadm download programs the main firmware image. The boot option selects programming of the boot monitor.

rscadm show
: View the current RSC configuration variable settings. If no variable is specified, rscadm shows all variable settings.

rscadm date
: Show or set RSC's time and date. The -s options can be used to set RSC's time and date to the hosts time and date.

rscadm send_event
: Send a text based event to RSC. RSC may forward the event based on its event configuration.

rscadm modem_setup
: Direct connection to the RSC modem. This allows the user to enter AT commands to configure the modem. "~." returns to prompt.

rscadm useradd
: Add user account to RSC. RSC can support up to four separate users.

rscadm userdel
: Delete a user account from RSC.

rscadm usershow
: Show details on the specified user account. If a username is not specified, all user accounts will be shown.

rscadm userpassword
: Set a password for the user account specified. This password overrides any existing password currently set. There is no verification of the old password before setting the new password. See the RSC documentation on valid password formats.

rscadm userperm
: Set the authorization profile for the user. See the userperm options section in this man page for more detail.

OPTIONS

The following options are supported for rscadm:

rscadm resetrsc

[-s] Perform a "soft" reset instead of a "hard" reset. A hard reset physically resets the RSC hardware. The RSC software jumps to the boot firmware, simulating a reset, for a soft reset.

rscadm download

[boot] Program the boot monitor portion of the flash. The main portion of the flash is usually programmed.

rscadm show

[variable] Show the value of that particular variable.

rscadm date

[-s] Set the date to the hosts time and date.

[[mmdd]HHMM | mmddHHMM[cc]yy][.SS]
: the date.

mm - month
dd - day
HH - hour
MM - minute
cc - the first two digits of the four digit year
yy - last 2 digits of the year number
SS - seconds

rscadm send_event

[-c]	Send a critical event. Without the -c, send_event sends a warning. Warnings are only logged in the RSC event log and not forwarded further.

rscadm usershow

[username]
RSC account name to display info on. If no username is given, all accounts will be displayed.

rscadm userperm

[cuar]	Set permissions for RSC account. If no permissions are specified, all four permissions will be disabled. The options are to; allow user to connect to (c)onsole, allow user to use the (u)ser commands to modify RSC accounts, allow user to (a)dminister/change the RSC configuration variables, allow the user to (r)eset RSC and to power on/off the host.

OPERANDS

The following operands are supported for rscadm:

rscadm set

variable	RSC configuration variable to set. See the RSC documentation for a list of configuration variables.
value	Value to set RSC configuration variable to. See the RSC documentation for a list of valid values.

rscadm download

file	Firmware file to download. The file should contain the RSC boot monitor image or RSC main image.

rscadm send_event

message	Text message to describe event. Should be enclosed in quotes.

rscadm useradd

username	Username for new RSC account.

rscadm userdel

username	RSC account to be removed.

rscadm userpassword

username	RSC account to have password set.

rscadm userperm

username	RSC account to have permissions changed.

EXIT STATUS

```
 = 0   on success
!= 0   on failure (with status message)
```

EXAMPLES

```
# rscadm date
# rscadm date -s
# rscadm date 050113101998
# rscadm set hostname rsc15
# rscadm show
# rscadm show hostname
# rscadm send_event -c "The UPS signaled a loss in power!"
# rscadm send_event "The disk is close to full capacity"
# rscadm useradd rscroot
# rscadm userdel olduser
# rscadm usershow
# rscadm usershow rscroot
# rscadm userperm rscroot cuar
# rscadm userperm newuser c
# rscadm userperm newuser
```

NOTES

rscadm modem_setup - "~." will only work after a new line.

rscadm MUST be run as root.

BUGS

None known.

NAME

scadm – administer System Controller (SC)

SYNOPSIS

scadm *subcommand* [*option*] [*argument*...]

DESCRIPTION

The **scadm** utility administers the System Controller (**SC**). This utility allows the host server to interact with the **SC**.

The **scadm** utility *must* be run as root.

The **scadm** utility has fifteen subcommands. Some subcommands have specific options and arguments associated with them. See **SUBCOMMANDS**, **OPTIONS**, **OPERANDS**, and **USAGE**.

SUBCOMMANDS

Subcommands immediately follow the **scadm** command on the command line, and are separated from the command by a <**SPACE**>.

The following subcommands are supported

date Display the **SC**'s time and date

The format for the **date** subcommand is:

scadm date

download
Program the **SC**'s firmware.

There are two parts to the firmware, the boot monitor and the main image.

By default, The **scadm** command's download programs the main firmware image. The **boot** argument selects programming of the boot monitor.

The format for the **download** subcommand is:

scadm download [boot] *file*

help Display a list of commands.

The format for the **help** subcommand is:

scadm help

loghistory
Display the most recent entries in the **SC** event log.

The format for the **loghistory** subcommand is:

scadm loghistory

resetrsc
: Reset the **SC**. There are two types of resets allowed, a **hard** reset and a **soft** reset.The **hard** reset is done by default. The **soft** reset can be selected by using the **-s** option.

  The format for the **resetrsc** subcommand is:

scadm resetrsc [**-s**]

send_event
: Manually send a text based event. The **SC** can forward the event to the **SC** event log. You can configure the **-c** option to send a critical warning to email, alert to logged in SC users, and **syslog**. Critical events are logged to **syslog**(3C). There is an **80** character limit to the length of the associated text message.

  The format for the **send_event** subcommand is:

scadm send_event [**-c**] "*message*"

set
: Set SC configuration variables to a value.

  Examples of SC configuration variables include: SC IP address **netsc_ipaddr** and SC Customer Information **sc_customerinfo**. See the output from the **scadm help** command for a complete list of SC configuration variables.

  The format for the **set** subcommand is:

scadm set *variable value*

show
: Display the current SC configuration variable settings. If no variable is specified, **scadm** shows all variable settings.

  The format for the **show** subcommand is:

scadm show [*variable*]

shownetwork
: Display the current network configuration parameters for SC.

  The format for the **shownetwork** subcommand is:

scadm shownetwork

useradd

Add user accounts to the **SC**. The **SC** supports up to sixteen separate users.

The format for the **useradd** subcommand is:

scadm useradd *username*

userdel

Delete a user account from **SC**.

The format for the **userdel** subcommand is:

scadm userdel *username*

userpassword

Set a password for the user account specified. This password overrides any existing password currently set. There is no verification of the old password before setting the new password.

The format for the **userpassword** subcommand is:

scadm userpassword *username*

userperm

Set the permission level for the user.

The format for the **userperm** subcommand is:

scadm userperm *username* [aucr]

usershow

Display details on the specified user account. If a username is not specified, all user accounts are displayed.

The format for the **usershow** subcommand is:

scadm usershow *username*

version

Display the version numbers of the **SC** and its components.

The format for the **version** subcommand is:

scadm version [**-v**]

OPTIONS

The **resetrsc**, **send_event**, and **version** subcommands have associated options. Options follow subcommands on the command line and are separated from the subcommand by a <**SPACE**>.

The **resetrsc** subcommand supports the following options:

-s Perform a soft reset instead of a hard reset. A hard reset physically resets the SC hardware. The SC software jumps to the boot firmware, simulating a reset, for a soft reset.

The **send_event** subcommand supports the following options:

-c Send a critical event. Without the **-c**, -**send_event** sends a warning.

The **version** subcommand supports the following options:

-v Display a verbose output of version numbers and associated information.

OPERANDS

The **download**, **send_event**, **set**, **show**, **useradd**, **userdel**, **userperm**, **usershow**, **userpassword**, and **userperm** subcommands have associated arguments (operands).

If the subcommand has an option, the arguments follow the option on the command line and is separated from the option by a <**SPACE**>. If the subcommand does not have an option, the arguments follow the subcommand on the command line and are separated from the subcommand by a <**SPACE**>. If there are more than one arguments, they are separated from each other by a <**SPACE**>.

The **download** subcommand supports the following arguments:

boot Program the boot monitor portion of the flash. The main portion of the flash is programmed without any arguments

file Specify *file* as the path to where the boot or main firmware image resides for download.

Examples of *file* are:

/usr/platform/*platform_type*/lib/image/alommainfw

or

/usr/platform/*platform_type*/lib/image/alombootfw

The **send_event** subcommand supports the following arguments: "" .nr)I *message*""n

Describe event using the test contained in *message*. Enclose *message* in quotation marks.

The **set** subcommand supports the following arguments:

variable
Set SC configuration *variable*.

value Set SC configuration variable to *value*.

The **show** subcommand supports the following arguments:

variable
: Display the value of that particular variable.

The **useradd** subcommand supports the following arguments:

username
: Add new SC account *username*.

The **userdel** subcommand supports the following arguments:

username
: Remove SC account *username*.

The **userperm** subcommand supports the following arguments:

-aucr Set permissions for SC user accounts. If no permissions are specified, all four permissions are disabled and read only access is assigned.

The following are the definitions for permissions:

a	Allow user to administer or change the SC configuration variables
u	Allow user to use the user commands to modify SC accounts
c	Allow user to connect to console.
r	Allow user to reset SC and to power on and off the host.

username
: Change permissions on SC account *username*.

The **-usershow** subcommand supports the following arguments:

username
: Display information on SC account *username* . If *username* is not specified, all accounts are displayed.

The **userpassword** subcommand supports the following arguments:

username
: Set SC password for *username*.

The **userperm** subcommand supports the following arguments:

username
: Change SC permissions for *username*.

EXAMPLES

Example 1: Displaying the SC's Date and Time

The following command displays the SC's date and time.

```
scadm date
```

Example 2: Setting the SC's Configuration Variables

The following command sets the SC's configuration variable **netsc_ipaddr** to **192.168.1.2**:

```
scadm set netsc_ipaddr 192.168.1.2
```

Example 3: Displaying the Current SC's Configuration Settings:

The following command displays the current SC configuration settings:

```
scadm show
```

Example 4: Displaying the Current Settings for a Variable

The following command displays the current settings for the **sys_hostname** variable:

```
scadm show sys_hostname
```

Example 5: Sending a Text-Based Critical Event

The following command sends a critical event to the SC logs, alerts the current SC users, and sends an event to **syslog**(3C):

```
scadm send_event -c "The UPS signaled a loss in power"
```

Example 6: Sending an Informational Text-Based Event

The following command sends an non-critical informational text based event to the SC event log:

```
scadm send_event "The disk is close to full capacity"
```

Example 7: Adding a User To the SC

The following command adds user **rscroot** to the SC:

```
scadm useradd rscroot
```

Example 8: Deleting a User From the SC

The following command deletes user **olduser** from the SC:

```
scadm userdel olduser
```

Example 9: Displaying User Details

The following command displays details of all user accounts:

```
scadm usershow
```

Example 10: Displaying Details for a Specific User

The following command displays details of user account **rscroot**:

```
scadm usershow rscroot
```

Example 11: Setting the User Permission Level

The following command sets the full permission level for user **rscroot** to **aucr**:

```
scadm userperm rscroot aucr
```

Example 12: Setting the User Permission Level

The following command sets only console access for user **newuser** to **c** scadm userperm newuser c

Example 13: Setting the User Permission Level

The following command sets the permission level for user **newuser** to read only access:

scadm userperm newuser

Example 14: Displaying the Current Network Parameters

The following command displays the current network configuation parameters for the SC:

scadm shownetwork

Example 15: Viewing the Loghistory

The following command displays the most recent entries in the SC event log:

scadm loghistory

Example 16: Displaying Verbose Information

The following command displays verbose version information on the SC and its components:

scadm version **-v**

EXIT STATUS

The following exit values are returned:

0 Successful completion.

non-zero
An error occurred.

ATTRIBUTES

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWkvm

SEE ALSO

syslog(3C), **attributes**(5)

NAME

sunvts – Invokes the SunVTS kernel and its user interface

SYNOPSIS

sunvts [–**lepqstv**] [–**o** *option_file*] [–**f** *log_dir*] [–**h** *hostname*]

AVAILABILITY

SUNWvts

DESCRIPTION

The **sunvts** command is used to invoke the SunVTS user interface and kernel on the same system. It could be used to start the user interface on the local system and connect to the SunVTS kernel on the remote system. By default, it displays CDE Motif graphic interface for CDE environment, OpenLook graphic interface for OpenWindows environment, or TTY interface for non-windowing system.

OPTIONS

–**l** Displays SunVTS OpenLook graphic interface.

–**e** Disables the security checking feature.

–**f** *log_dir*
Specifies an alternative log_file directory. The default log_file directory is **/var/opt/SUNWvts/logs.**

–**h** *hostname*
Starts the SunVTS user interface on the local system, which connects to or invokes the SunVTS kernel on the specified host after security checking succeeds.

–**o** *option_file*
Starts the SunVTS kernel with the test options loaded from the specified *option_file*, which by default is located in **/var/opt/SUNWvts/options.**

–**p** Starts the SunVTS kernel **vtsk (1M)** such that it does not probe the test system's devices.

–**q** Automatically quits both the SunVTS kernel and the user interface when testing stops.

–**s** Automatically starts testing from a selected group of tests. The flag must be used with the –**o** *option_file* flag.

–**t** Starts **vtstty (1M),** a TTY based interface, instead of CDE or OpenLook interface.

–**v** Displays version information from **vtsui**(1M) and **vtsk**(1M).

NOTES

If **vtsk (1M)** is already running on the test system, the **sunvts** command ignores the –**e,** –**o,** –**f,** –**q,** –**p,** and –**s** options.

SEE ALSO

vtsk(1M), **vtstty**(1M), **vtsui**(1M), **vtsprobe**(1M)

NAME

update_drv – modify device driver attributes

SYNOPSIS

update_drv [**-v**] *device_driver*

update_drv [**-b** *basedir*] [**-v**] **-a -i** '*identify-name*' *device_driver*

update_drv [**-b** *basedir*] [**-v**] **-a -m** '*permission*' *device_driver*

update_drv [**-b** *basedir*] [**-v**] **-a -i** '*identify-name*' **-m** '*permission*' *device_driver*

update_drv [**-b** *basedir*] [**-v**] **-d -i** '*identify-name*' *device_driver*

update_drv [**-b** *basedir*] [**-v**] **-d -m** '*permission*' *device_driver*

update_drv [**-b** *basedir*] [**-v**] **-d -i** '*identify-name*' **-m** '*permission*' *device_driver*

The **update_drv** command informs the system about attribute changes to an installed device driver. It can be used to re-read the **driver.conf**(4) file, or to add, modify, or delete a driver's minor node permissions or aliases.

Without options, **update_drv** reloads the **driver.conf** file.

Upon successfully updating the aliases, the driver binding takes effect upon reconfig boot or hotplug of the device.

Upon successfully updating the permissions, only the new driver minor nodes get created with the modified set of file permissions. Existing driver minor nodes do not get modified.

The following options are supported:

-a Add a *permission* or an *aliases* entry.

With the **-a** option specified, a permission entry (using the **-m** option), or a driver's aliases entry (using the **-i** option) can be added or updated. If a matching minor node permissions entry is encountered (having the same driver name and the minor node), it is replaced. If a matching aliases entry is encountered (having a different driver name and the same alias), an error is reported.

The **-a** and **-d** options are mutually exclusive.

-b *basedir*

Installs or modifies the driver on the system with a root directory of basedir rather than installing on the system executing **update_drv**.

-d Deletes a permission or an aliases entry.

Either the **-m** *permission* or **-i** *identify-name* option needs to be specified with the **-d** option.The **-d** and **-a** options are mutually exclusive.

If the entry doesn't exist **update_drv** returns an error.

-i '*identify-name*'

A white-space separated list of aliases for the driver. If **-a** or **-d** option is not specified then this option is ignored. The *identify-name* string is mandatory. If all aliases need to be removed, **rem_drv**(1M) is recommended.

-m '*permission*'
Specify a white-space separated list of file system permissions for the device node of the device driver. If **-a** or **-d** option is not specified then, this option is ignored. The permission string is mandatory.

Example 1: Adding or Modifying an Existing Minor Permissions Entry

The following command adds or modifies the exisitng minor permissions entry of the **clone** driver:

```
example# update_drv -a -m 'llc1 777 joe staff' clone
```

Example 2: Removing All Minor Permissions Entries

The following command removes all minor permission entries of the **usbprn** driver, the USB printer driver:

```
example# update_drv -d -m '* 0666 root sys' usbprn
```

Example 3: Adding a Driver Aliases Entry

The following command adds a driver aliases entry of the **ugen** driver with the identity string of **usb459,20**:

```
example# update_drv -a -i '"usb459,20"' ugen
```

Example 4: Re-reading the driver.conf File For the **ohci** Driver

The following command re-reads the **driver.conf**(4) file.

```
example# update_drv ohci
```

The following exit values are returned:

0	Successful completion.
>**0**	An error occurred.

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu

add_drv(1M), **modunload**(1M), **rem_drv**(1M), **driver.conf**(4), **attributes**(5)

If **-b** option is specified, **update_drv** does not re-read the **driver.conf** file.

It is possible to add an alias , which changes the driver binding of a device already being managed by a different driver.

NAME

vts_cmd – Send a command to the SunVTS kernel (vtsk)

SYNOPSIS

vts_cmd [*command*] [*argument*]

AVAILABILITY

SUNWvts

DESCRIPTION

vts_cmd is a UNIX shell application that allows you to send a single command to the SunVTS kernel (vtsk). The test machine's SunVTS kernel will send the response to the standard output.

The SunVTS application programming interface (API) is character based, which means that a string of characters (in the form of a *command*) can be sent to the SunVTS kernel, which then returns a reply back in the form of a string of characters.

vts_cmd(1M) allows the user to send commands and receive replies from a UNIX command line.

OPTIONS

vts_cmd uses the commands listed below. In all cases, the commands (and any of the command's arguments) must follow vts_cmd. See the EXAMPLES section for reference.

Some of the command descriptions listed below refer to a testnode. In the SunVTS API, there is a hierarchy of testnodes, with the system being on the top, the test groups below the system, and the tests themselves at the bottom. In the commands below, use a slash "/" to refer to the system. A test group can be one of the following: Processor(s), Memory, Network, SCSI-Devices(esp0), Comm.Ports, Graphics, OtherDevices, or any user specified group. When referring to a test, you must mention the device name and the test name [for example, sound0(audio)].

list testnode
Displays all the testnodes under the specified testnode.

config testnode
Displays the configuration information of the testnode.

status [testnode] [-r]
Displays the testing status information of the system. If a testnode is specified, status will display the status information of that testnode. If you use the -r argument, the status information of all of testnodes recursive to the testnode will be displayed.

option [testnode] [-l] [-h|n|s|t|a]
Either displays all the options associated with the specified testnode, or sets a specific option in a testnode.

To display a testnode's options, type option followed by the testnode and one of the categories:

-h	**Threshold**
-n	**Notify category**
-s	**Scheduling category**

-t Test execution category
-a Advanced category

vts_cmd will print all options, as well as the setting of each option. Use the **-l** option to display the options in long form. In long form, the options will be displayed with all their settings.

option [testnode] [test_option] [-g|s|x|y|z]

-g is used to pass all of the current option settings, for a given instance of a given test, to all of the same instances and tests that are in the same group (will not affect the same tests that are in different groups).

-s is used to pass all of the current option settings for a given instance of a given test, to all of the same instances for all of the same tests on the system (rather than for a group, as with -g).

-x is used to pass all of the current option settings for a given instance of a given test, to all the instances of that test.

-y is used to pass all of the current option settings for a given instance of a given test, to all the instances of all the same tests in a particular group.

-z is used to pass all of the current option settings for a given instance of a given test, to all the instances of all the same tests in the whole system.

To set an option, you must state the testnode immediately followed by the option and the new setting. You must use this format when setting an option:

vts_cmd option testnode[option:setting]

Once the option has been successfully changed, vts_cmd will display the word "DONE".

select testnode
Selects a testnode. If a testnode is selected, all the tests associated with the testnode will be enabled and run when testing begins.

For example, if you select the Graphics testnode, all the tests in Graphics will be enabled for testing. If you select just the "fpu(fputest)" test, then you will only enable this test.

deselect testnode
Deselects a testnode. If a testnode is deselected, all the tests associated with the testnode will be disabled and will not be run when testing begins.

For example, if you deselect the OtherDevices testnode, all the tests in the OtherDevices will be disabled. If you select just the "cgsix0(cg6)" test, then you will only enable this test.

start
Starts all enabled (selected) SunVTS tests.

stop
Stops all running SunVTS tests.

suspend
Suspends (or pauses) all running SunVTS tests. When you are ready to resume testing, type "resume".

resume
Resumes any suspended tests.

reset
Resets all the SunVTS pass and error counts to zero.

probe
Probes all the devices on the test machine and updates the SunVTS kernel's device list.

If a device is listed in the device list, but it is not found during the probe, it will be removed from the list. Conversely, if a device does not exist in a previous device list and is found during the probe, it will be added to the list.

load option_file
Loads an option file. Once loaded, the system and test options will be changed to reflect the settings listed in the option file.

Option files are stored in the /var/opt/SUNWvts/options directory.

store option_file
Creates an option file, listing all the system and test options, and save it in the /var/opt/SUNWvts/options directory.

quit
: Terminates the SunVTS kernel (vtsk).

invokeds
: Starts the deterministic scheduler.

quitds
: Terminates the deterministic scheduler.

loadseq sequence_file
: Loads a sequence file. Once loaded, the deterministic scheduler UI will reflect the tasks in the loaded sequence file.

storeseq sequence_file
: Creates sequence_file, listing all the tasks in the directory /var/opt/SUNWvts/sequences.

statusseq
: Returns a string containing the status information of the currently running sequence. The string consists of four fields separated by commas (","). The fields are: current status of SunVTS, current loop count of the sequence, total loop count of the sequence, and currently running task's position.

startseq
: Starts the execution of the deterministic scheduler.

stopseq
: Stops the execution of the currently running task in the sequence file. Upon starting again, the execution will start from the tast that was stopped.

resumeseq
: Restarts the execution of the sequence file. Execution will start at the point where the sequence was stopped, unless the sequence was reset, in which case it would start at the beginning of the sequence file.

resetseq
: Sets the starting point of the execution to the start of the sequence file. Will also reset the passes and error count.

suspendseq
: Suspends the execution of the currently running task in the sequence file.

removeseq sequence_file
: Removes sequence_file from the list of sequence files in the directory /var/opt/SUNWvts/sequences.

listtask
: Lists the tasks that are present in the currently loaded

sequence file.

addtask task_name [i]
Adds task_name at the ith position in the sequence file. If no index is passed then the task would be added to the end of the list.

deletetask [i]
Removes the task at the specified index from the selected sequence.

loadtask task_name
Loads a task file. Once loaded, the system and test options will be changed to reflect the settings listed in the task file.

setloopcount count
Sets the number of loops to run in the current sequence to count.

getvtsmode
Gets the current mode of SunVTS kernel.

EXAMPLES

To list out the configuration information of the test machine, you would use the config command:

```
sample% vts_cmd config /
/[Hostname:sample,Model:SPARCstation-10,SunVTS version:1.0]:idle
```

To load an option file, you would use the load command:

```
sample% ls /var/adm/sunvtslog/options
CPU_options        sample              options
sbus_standard
sample% vts_cmd load sbus_standard
DONE
```

To print all the system options in the Comm.Ports testnode, you would use the option command and pipe the output to your local printer:

```
sample% vts_cmd option Comm.Ports -l | lp
request id is printer-213 (standard input)
```

ENVIRONMENT

VTS_CMD_HOST=hostname
The hostname of the test machine running the SunVTS kernel **(vtsk).** If this environment variable is not set, **vts_cmd** will attempt to send the commands to

the local machine's SunVTS kernel.

SEE ALSO

SunVTS User's Guide

NAME

vtsk – SunVTS diagnostic kernel

SYNOPSIS

vtsk [–**epqsv**] [–**o** *options_file*] [–**f** *logfile_directory*]

AVAILABILITY

SUNWvts

DESCRIPTION

The **vtsk** command starts up the SunVTS diagnostic kernel as a background process. There can only be one copy of **vtsk** running at a time. Only the superuser can execute this command.

Normally, **vtsk** is automatically started up by the **sunvts (1M)** command if it is not already running. **vtsk** will also be invoked by **inetd (1M)** when there is a connection request from vtsui. In that case, the security file, **.sunvts_sec,** will be checked for the permission before running vtsk on the target host specified by **vtsui**(1M).

OPTIONS

–**e** Enables the security checking for all connection requests.

–**p** Starts SunVTS diagnostic kernel, but does not probe system configuration.

–**q** Quits both the SunVTS diagnostic kernel and the attached User Interfaces when the testing is completed.

–**s** Runs enabled tests immediately after started.

–**v** Display SunVTS diagnostic kernel's version information only.

–**o** *options_file*
Starts the SunVTS diagnostic kernel and sets the test options according to the option file named *options_file.*

–**f** *logfile_directory*
Specifies an alternative logfile directory, other than the default.

EXIT STATUS

The following exit values are returned:

0 Successful completion.

–**1** An error occurred.

FILES

/var/opt/SUNWvts/options	default option file directory.
/var/opt/SUNWvts/logs	default log file directory.

SEE ALSO

sunvts(1M), **vtsui**(1M), **vtstty**(1M), **vtsprobe**(1M)

NAME

vtsprobe – prints the device probe information from the SunVTS kernel

SYNOPSIS

vtsprobe [–**m**] [–**h** *hostname*]

AVAILABILITY

SUNWvts

DESCRIPTION

vtsprobe is a utility that displays the device and configuration information contained in the SunVTS kernel. The output includes the SunVTS assigned group for the device, the device name, the device instance, the testname attached to this device, and the configuration information obtained from the device-specific test probe.

OPTIONS

–**m** Specifies manufacturing mode, which displays the probe information in a format that is easy to read using script files.

–**h** *hostname*
Specifies the *hostname* to connect to and get the device and configuration information. If not specified, the current host will be used.

USAGE

After the SunVTS kernel is up and running, you may type **vtsprobe** at the shell prompt to get the probe output. (See the **sunvts (1M)** man page for more information on how to start up SunVTS.

EXAMPLE

Running **vtsprobe** on a sun4m SPARCclassic produces the following output:

```
% vtsprobe

    Processor(s)
        system(systest)
            System Configuration=sun4m SPARCclassic
            System clock frequency=50 MHz
            SBUS clock frequency=25 MHz
        fpu(fputest)
            Architecture=sparc
            Type=TI TMS390S10 or TMS390S15 microSPARC chip
    Memory
        kmem(vmem)
            Total: 143120KB
        mem(pmem)
            Physical Memory size=24 Mb
    SCSI-Devices(esp0)
        c0t2d0(rawtest)
            Capacity: 638.35MB
            Controller: esp0
            Vendor: MICROP
            SUN Id: 1588-15MBSUN0669
            Firmware Rev: SN0C
```

```
                Serial Number: 1588-15MB103
        c0t2d0(fstest)
                Controller: esp0
        c0t3d0(rawtest)
                Capacity: 404.65MB
                Controller: esp0
                Vendor: SEAGATE
                SUN Id: ST1480   SUN0424
                Firmware Rev: 8628
                Serial Number: 00836508
        c0t3d0(fstest)
                Capacity: 404.65MB
                Controller: esp0
                Vendor: SEAGATE
                SUN Id: ST1480   SUN0424
                Firmware Rev: 8628
                Serial Number: 00836508
        c0t3d0(fstest)
                Controller: esp0
        c0t6d0(cdtest)
                Controller: esp0
        tape1(tapetest)
                Drive Type: Exabyte EXB-8500 8mm Helical Scan
Network
        isdn0(isdntest)
                NT Port  TE Port
        le0(nettest)
                Host_Name: ctech84
                Host Address: 129.146.210.84
                Host ID: 8001784b
                Domain Name: scsict.Eng.Sun.COM
Comm.Ports
        zs0(sptest)
                Port a -- zs0  /dev/term/a : /devices/ ... a
                Port b -- zs1  /dev/term/b : /devices/ ... b
Graphics
        cgthree0(fbtest)

OtherDevices
        bpp0(bpptest)
                Logical name: bpp0
        sound0(audio)
                Audio Device Type: AMD79C30
        sound1(audio)
                Audio Device Type: DBRI Speakerbox
```

spd0(spdtest)
Logical name: spd0

NOTES

The output of **vtsprobe** is highly dependent on the device being correctly configured into the system (so that a SunVTS probe for the device can be run successfully on it) and on the availability of a device-specific test probe.

If the device is improperly configured or if there is no probing function associated with this device, **vtsprobe** cannot print any information associated with it.

SEE ALSO

sunvts(1M), **vtsk**(1M), **vtsui**(1M), **vtstty**(1M)

NAME

vtstty – TTY interface for SunVTS

SYNOPSIS

vtstty [–**qv**] [–**h** *hostname*]

AVAILABILITY

SUNWvts

DESCRIPTION

vtstty is the default interface for SunVTS in the absence of a windowing environment. It can be used in a non-windowing environment such as a terminal connected to the serial port of the system. However, its use is not restricted to this; **vtstty** can also be used from shell window.

OPTIONS

–**q** The "auto-quit" option automatically quits when the conditions for SunVTS to quit are met.

–**v** Prints the **vtstty** version. The interface is not started when you include this option.

–**h** *hostname*
Connects to the SunVTS kernel running on the host identified by *hostname.*

USAGE

The **vtstty** screen consists of four panels: main control, status, test groups, and console. The panels are used to display choices that the user can select to perform some function and/or to display information. A panel is said to be "in focus" or in a "selected" state when it is surrounded by asterisks and the current item is highlighted. In order to choose from the items in a panel, the focus should be shifted to that panel first.

The following are the different types of selection items that can be present in a panel:

Text string	Describes a choice that, when selected, either pops up another panel or performs a function. For example, "stop" will stop the SunVTS testing.
Data entry field	To enter or edit numeric or textual data.
Checkbox	Represented as "[ ]". Checkboxes are associated with items and indicate whether the associated item is selected or not. A checkbox can be in one of the following two states: Deselected [ ] or Selected [∗].

The key assignments given below describe the keys for shifting focus, making a selection, and performing other functions:

TAB or <CTRL>W	Shift focus to another panel
RETURN	Select current item
Spacebar	Toggle checkbox
Up arrow or <CTRL>U	Move up one item
Down arrow or <CTRL>N	Move down one item

Left arrow or <CTRL>P	Move left one item
Right arrow or <CTRL>R	Move right one item
Backspace	Delete text in a data entry field
ESC	Dismiss a pop-up
<CTRL>F	Scroll forward in a scrollable panel
<CTRL>B	Scroll backward in a scrollable panel
<CTRL>X	Quit **vtstty** but leave the SunVTS kernel running
<CTRL>L	Refresh the **vtstty** screen

NOTES

1. To run **vtstty** from a telnet session, carry out the following steps:

   a. Before telnet-ing, determine the values for "rows and "columns". (See **stty**(1)).

   b. Set term to the appropriate type after telnet-ing(for example, **set term=vt100**

   c. Set the values of columns and rows to the value noted above. (See **stty**(1)).

2. Before running **vtstty** ensure that the environment variable describing the terminal type is set correctly.

SEE ALSO

sunvts(1M), **vtsk**(1M), **vtsui**(1M), **vtsprobe**(1M)

NAME

vtsui – SunVTS Graphic User Interface (CDE)

SYNOPSIS

vtsui [–**qv**] [–**h** *hostname*]

AVAILABILITY

SUNWvts

DESCRIPTION

The **vtsui** command starts up the CDE Motif version of SunVTS graphic user interface. There can be multiple instances of **vtsui** running at the same time, all connected to one SunVTS diagnostic kernel, **vtsk**(1M). The name of the host machine running the diagnostic kernel, **vtsk**(1M), will be displayed in the title bar of the graphical user interface window.

vtsui is automatically started up by the **sunvts (1M)** command. **vtsui can** be also used to start **vtsk (1M)** if **inetd (1M)** is in operation. In that case, the security file, **sunvts_sec,** will be checked for the permission before running **vtsk** on the target host.

See the "SunVTS User's Guide" for a complete description on using the graphical user interface.

OPTIONS

–**q** Quits the SunVTS graphic user interface when testing has terminated.

–**v** Displays graphic user interface version information only.

–**h** *hostname*
Starts the SunVTS graphic user interface and connects to the SunVTS diagnostic kernel running on *hostname*, or invokes the kernel if not running, after security checking succeeds. If *hostname* not specified, the local host is assumed.

EXIT STATUS

The following exit values are returned:

0 Successful completion.

1 An error occurred.

SEE ALSO

sunvts(1M), **vtsk**(1M), **vtstty**(1M), **vtsprobe**(1M)

NAME

SCF_Session_close, SCF_Terminal_close, SCF_Card_close – close a smartcard session, terminal, or card

SYNOPSIS

cc [*flag...*] *file...* -**lsmartcard** [*library...*]
#include <**smartcard/scf.h**>

SCF_Status_t SCF_Session_close(**SCF_Session_t** *session*);

SCF_Status_t SCF_Terminal_close(**SCF_Terminal_t** *terminal*);

SCF_Status_t SCF_Card_close(**SCF_Card_t** *card*);

card An object that was returned from **SCF_Terminal_getCard**(3SMARTCARD)

session
An object that was returned from **SCF_Session_getSession**(3SMARTCARD)

terminal
An object that was returned from **SCF_Session_getTerminal**(3SMARTCARD)

These functions release the resources (memory, threads, and others) that were allocated within the library when the session, terminal, or card was opened. Any storage allocated by calls to **SCF_Session_getInfo**(3SMARTCARD), **SCF_Terminal_getInfo**(3SMARTCARD), or **SCF_Card_getInfo**(3SMARTCARD) is deallocated when the associated object is closed. Attempts to access results from these interfaces after the object has been closed results in undefined behavior.

If a card that was locked by **SCF_Card_lock**(3SMARTCARD) is closed, the lock is automatically released. When a terminal is closed, any event listeners on that terminal object are removed and any cards that were obtained with the terminal are closed. Similarly, closing a session will close any terminals or cards obtained with that session. These are the only cases where the library will automatically perform a close.

Once closed, a session, terminal, or card object can no longer be used by an SCF function. Any attempt to do so results in an **SCF_STATUS_BADHANDLE** error. The sole exception is that closing an object, even if already closed, is always a successful operation.

Closing a handle is always a successful operation that returns **SCF_STATUS_SUCCESS**. The library can safely detect handles that are invalid or already closed.

Example 1: Close each object explicitly.

```
SCF_Status_t status;
SCF_Session_t mySession;
SCF_Terminal_t myTerminal;
SCF_Card_t myCard;

status = SCF_Session_getSession(&mySession);
if (status != SCF_STATUS_SUCCESS) exit(1);
status = SCF_Session_getTerminal(mySession, NULL, &myTerminal);
```

```
if (status != SCF_STATUS_SUCCESS) exit(1);
status = SCF_Terminal_getCard(myTerminal, &myCard);
if (status != SCF_STATUS_SUCCESS) exit(1);

/* (Do interesting things with smartcard...) */

SCF_Card_close(myCard);
SCF_Terminal_close(myTerminal);
SCF_Session_close(mySession);
```

Example 2: Allow the library to close objects.

```
SCF_Status_t status;
SCF_Session_t mySession;
SCF_Terminal_t myTerminal;
SCF_Card_t myCard;

status = SCF_Session_getSession(&mySession);
if (status != SCF_STATUS_SUCCESS) exit(1);
status = SCF_Session_getTerminal(mySession, NULL, &myTerminal);
if (status != SCF_STATUS_SUCCESS) exit(1);
status = SCF_Terminal_getCard(myTerminal, &myCard);
if (status != SCF_STATUS_SUCCESS) exit(1);

/* (Do interesting things with smartcard...) */

SCF_Session_close(mySession);
/* myTerminal and myCard have been closed by the library. */
```

See **attributes**(5) for descriptions of the following attributes:

tab() allbox; cw(2.750000i)| cw(2.750000i) lw(2.750000i)| lw(2.750000i). ATTRIBUTE TYPEATTRIBUTE VALUE Interface StabilityEvolving MT-LevelMT-Safe

libsmartcard(3LIB), **SCF_Card_getInfo**(3SMARTCARD), **SCF_Card_lock**(3SMARTCARD), **SCF_Session_getInfo**(3SMARTCARD), **SCF_Session_getSession**(3SMARTCARD), **SCF_Session_getTerminal**(3SMARTCARD), **SCF_Terminal_getCard**(3SMARTCARD), **SCF_Terminal_getInfo**(3SMARTCARD), **attributes**(5)

NAME

SCF_Card_exchangeAPDU – send a command APDU to a card and read the card's response

SYNOPSIS

cc [*flag*...] *file*... **-lsmartcard** [*library*...]
#include <**smartcard/scf.h**>

SCF_Status_t SCF_Card_exchangeAPDU(**SCF_Card_t** *card*, **const uint8_t** **sendBuffer*, **size_t** *sendLength*, **uint8_t** **recvBuffer*, **size_t** **recvLength*);

card The card (from **SCF_Terminal_getCard**(3SMARTCARD)) to communicate with.

sendBuffer
A pointer to a buffer containing the command APDU.

sendLength
The number of bytes in the sendBuffer (that is, the size of the command APDU).

recvBuffer
A pointer to a buffer in which the card's reply APDU should be stored. This buffer can be the same as the *sendBuffer* to allow the application to conserve memory usage. The buffer must be large enough to store the expected reply.

recvLength
The caller specifies the maximum size of the recvBuffer in *recvLength*. The library uses this value to prevent overflowing the buffer. When the reply is received, the library sets *recvLength* to the actual size of the reply APDU that was stored in the *recvBuffer*.

The **SCF_Card_exchangeAPDU( )** function sends a binary command to the card and reads the reply. The application is responsible for constructing a valid command and providing a receive buffer large enough to hold the reply. Generally, the command and reply will be ISO7816-formatted APDUs (Application Protocol Data Units), but the SCF library does not examine or verify the contents of the buffers.

If the caller needs to perform a multi-step transaction that must not be interrupted, **SCF_Card_lock**(3SMARTCARD) should be used to prevent other applications from communicating with the card during the transaction. Similarly, calls to **SCF_Card_exchangeAPDU( )** must be prepared to retry the call if **SCF_STATUS_CARDLOCKED** is returned.

An ISO7816-formatted command APDU always begins with a mandatory 4 byte header (CLA, INS, P1, and P2), followed by a variable length body (zero or more bytes). For details on the APDUs supported by a specific card, consult the documentation provided by the card manufacturer or applet vendor.

An ISO7816-formatted reply APDU consists of zero or more bytes of data, followed by a manditory 2 byte status trailer (SW1 and SW2).

If the APDU is successfully sent and a reply APDU is successfully read, **SCF_STATUS_SUCCESS** is returned with *recvBuffer* and *recvLength* set appropriately. Otherwise, an error value is returned and both *recvBuffer* and *recvLength* remain unaltered.

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

SCF_STATUS_BADARGS
: Neither *sendBuffer*, *recvBuffer*, nor *recvLength* can be null pointers. The value of recvLength must be at least 2.

SCF_STATUS_BADHANDLE
: The card has been closed or is invalid.

SCF_STATUS_CARDLOCKED
: The APDU cannot be sent because the card is locked by another application.

SCF_STATUS_CARDREMOVED
: The card object cannot be used because the card represented by the **SCF_Card_t** has been removed

SCF_STATUS_COMMERROR
: The connection to the server was closed.

SCF_STATUS_FAILED
: An internal error occurred.

SCF_STATUS_NOSPACE
: The specified size of *recvBuffer* is too small to hold the complete reply APDU.

Example 1: Send a command to the card.

```
SCF_Status_t status;
SCF_Card_t myCard;
uint8_t commandAPDU[] = {0x00, 0xa4, 0x00, 0x00, 0x02, 0x3f, 0x00};
uint8_t replyAPDU[256];
uint32_t commandSize = sizeof(commandAPDU);
uint32_t replySize = sizeof(replyAPDU);
/* (...call SCF_Terminal_getCard to open myCard...) */

/* Send the ISO7816 command to select the card's MF. */
status = SCF_Card_exchangeAPDU(myCard, commandAPDU, commandSize,
    replyAPDU, &replySize);
if (status != SCF_STATUS_SUCCESS) exit(1);

printf("Received a %d byte reply.\n", replySize);
printf("SW1=0x%02.2x SW2=0x%02.2x\n",
    replyAPDU[replySize-2], replyAPDU[replySize-1]);

/* ... */
```

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

libsmartcard(3LIB), **SCF_Card_lock**(3SMARTCARD), **SCF_Terminal_getCard**(3SMARTCARD), **attributes**(5)

NAME

SCF_Session_freeInfo, SCF_Terminal_freeInfo, SCF_Card_freeInfo – deallocate information storage

SYNOPSIS

cc [*flag*...] *file*... **-lsmartcard** [*library*...]
#include <**smartcard/scf.h**>

SCF_Status_t SCF_Session_freeInfo(**SCF_Session_t** *session*, **void** **value*);

SCF_Status_t SCF_Terminal_freeInfo(**SCF_Terminal_t** *terminal*, **void** **value*);

SCF_Status_t SCF_Card_freeInfo(**SCF_Card_t** *card*, **void** **value*);

card An object that was returned from **SCF_Terminal_getCard**(3SMARTCARD). This object must be associated with the information value being freed.

session
An object that was returned from **SCF_Session_getSession**(3SMARTCARD). This object must be associated with the information value being freed.

terminal
An object that was returned from **SCF_Session_getTerminal**(3SMARTCARD). This object must be associated with the information value being freed.

value A pointer that was returned from a call to **SCF_Session_getInfo**(3SMARTCARD), **SCF_Terminal_getInfo**(3SMARTCARD), or **SCF_Card_getInfo**(3SMARTCARD).

When information is requested for an object (for example, by using **SCF_Session_getInfo( )**), the result is placed in memory allocated for that request. This memory must eventually be deallocated, or a memory leak will result. The deallocation of memory can occur in one of two ways.

- The simplest method is to allow the **smartcard** library to automatically deallocate memory when the object associated with the information is closed. For example, when **SCF_Card_close**(3SMARTCARD) is called, any information obtained from **SCF_Card_getInfo( )** for that card object is deallocated. The application is not required to call **SCF_Card_freeInfo( )** at all.
- If the object persists for a long period of time, the application can explicitly request the information to be deallocated without closing the object, so that memory is not wasted on unneeded storage. Similarly, if an application repeatedly requests information about an object (even the same information), the application can explicitly request deallocation as needed, so that memory usage does not continue to increase until the object is closed. In general, requesting information to be deallocated can be used to reduce runtime memory bloat.

Attempts to access deallocated memory result in undefined behavior.

If the information is successfully deallocated, **SCF_STATUS_SUCCESS** is returned. Otherwise, an error value is returned.

These functions will fail if:

SCF_STATUS_BADARGS
The specified value cannot be deallocated, possibly because of an invalid pointer, a value already deallocated, or because the value is not associated with the specified session, terminal, or card.

SCF_STATUS_BADHANDLE
The specified session, terminal, or card has been closed or is invalid.

SCF_STATUS_FAILED
An internal error occured.

Example 1: Free information.

```
char *terminalName;
SCF_Status_t status;
SCF_Terminal_t myTerminal;

/* (...call SCF_Session_getTerminal to open myTerminal...) */

status = SCF_Terminal_getInfo(myTerminal, "name", &terminalName);
if (status != SCF_STATUS_SUCCESS) exit(1);

printf("The terminal name is %s\n", terminalName);

status = SCF_Terminal_freeInfo(myTerminal, terminalName);
if (status != SCF_STATUS_SUCCESS) exit(1);
```

See **attributes**(5) for descriptions of the following attributes:

tab() allbox; cw(2.750000i)| cw(2.750000i) lw(2.750000i)| lw(2.750000i). ATTRIBUTE TYPEATTRIBUTE VALUE Interface StabilityEvolving MT-LevelMT-Safe

libsmartcard(3LIB), **SCF_Session_getInfo**(3SMARTCARD), **SCF_Session_getSession**(3SMARTCARD), **SCF_Session_getTerminal**(3SMARTCARD), **SCF_Terminal_getCard**(3SMARTCARD), **attributes**(5)

NAME

SCF_Session_freeInfo, SCF_Terminal_freeInfo, SCF_Card_freeInfo – deallocate information storage

SYNOPSIS

cc [*flag*...] *file*... **-lsmartcard** [*library*...]
#include <**smartcard/scf.h**>

SCF_Status_t SCF_Session_freeInfo(**SCF_Session_t** *session*, **void** **value*);

SCF_Status_t SCF_Terminal_freeInfo(**SCF_Terminal_t** *terminal*, **void** **value*);

SCF_Status_t SCF_Card_freeInfo(**SCF_Card_t** *card*, **void** **value*);

card An object that was returned from **SCF_Terminal_getCard**(3SMARTCARD). This object must be associated with the information value being freed.

session
An object that was returned from **SCF_Session_getSession**(3SMARTCARD). This object must be associated with the information value being freed.

terminal
An object that was returned from **SCF_Session_getTerminal**(3SMARTCARD). This object must be associated with the information value being freed.

value A pointer that was returned from a call to **SCF_Session_getInfo**(3SMARTCARD), **SCF_Terminal_getInfo**(3SMARTCARD), or **SCF_Card_getInfo**(3SMARTCARD).

When information is requested for an object (for example, by using **SCF_Session_getInfo( )**), the result is placed in memory allocated for that request. This memory must eventually be deallocated, or a memory leak will result. The deallocation of memory can occur in one of two ways.

- The simplest method is to allow the **smartcard** library to automatically deallocate memory when the object associated with the information is closed. For example, when **SCF_Card_close**(3SMARTCARD) is called, any information obtained from **SCF_Card_getInfo( )** for that card object is deallocated. The application is not required to call **SCF_Card_freeInfo( )** at all.
- If the object persists for a long period of time, the application can explicitly request the information to be deallocated without closing the object, so that memory is not wasted on unneeded storage. Similarly, if an application repeatedly requests information about an object (even the same information), the application can explicitly request deallocation as needed, so that memory usage does not continue to increase until the object is closed. In general, requesting information to be deallocated can be used to reduce runtime memory bloat.

Attempts to access deallocated memory result in undefined behavior.

If the information is successfully deallocated, **SCF_STATUS_SUCCESS** is returned. Otherwise, an error value is returned.

These functions will fail if:

SCF_STATUS_BADARGS
: The specified value cannot be deallocated, possibly because of an invalid pointer, a value already deallocated, or because the value is not associated with the specified session, terminal, or card.

SCF_STATUS_BADHANDLE
: The specified session, terminal, or card has been closed or is invalid.

SCF_STATUS_FAILED
: An internal error occured.

Example 1: Free information.

```
char *terminalName;
SCF_Status_t status;
SCF_Terminal_t myTerminal;

/* (...call SCF_Session_getTerminal to open myTerminal...) */

status = SCF_Terminal_getInfo(myTerminal, "name", &terminalName);
if (status != SCF_STATUS_SUCCESS) exit(1);

printf("The terminal name is %s\n", terminalName);

status = SCF_Terminal_freeInfo(myTerminal, terminalName);
```
if (status != SCF_STATUS_SUCCESS) exit(1);

See **attributes**(5) for descriptions of the following attributes:

tab() allbox; cw(2.750000i)| cw(2.750000i) lw(2.750000i)| lw(2.750000i). ATTRIBUTE TYPEATTRIBUTE VALUE Interface StabilityEvolving MT-LevelMT-Safe

libsmartcard(3LIB), **SCF_Session_getInfo**(3SMARTCARD), **SCF_Session_getSession**(3SMARTCARD), **SCF_Session_getTerminal**(3SMARTCARD), **SCF_Terminal_getCard**(3SMARTCARD), **attributes**(5)

NAME

SCF_Session_getInfo, SCF_Terminal_getInfo, SCF_Card_getInfo – retrieve information about a session, terminal, or card

SYNOPSIS

cc [*flag*...] *file*... **-lsmartcard** [*library*...]
#include <**smartcard/scf.h**>

SCF_Status_t SCF_Session_getInfo(**SCF_Session_t** *session*, **const char** **name*, **void** **value*);

SCF_Status_t SCF_Terminal_getInfo(**SCF_Terminal_t** *terminal*, **const char** **name*, **void** **value*);

SCF_Status_t SCF_Card_getInfo(**SCF_Card_t** *card*, **const char** **name*, **void** **value*);

card An object that was returned from **SCF_Terminal_getCard**(3SMARTCARD).

name The name of a property for which a value is to be returned. The name is case-sensitive.

session
An object that was returned from **SCF_Session_getSession**(3SMARTCARD).

terminal
An object that was returned from **SCF_Session_getTerminal**(3SMARTCARD).

value The value of the property. The actual type of the value depends on what property was being queried.

These functions obtain information about a session, terminal, or card. The information returned represents the current state of the object and can change between calls.

Each call allocates new storage for the returned result. This storage is tracked internally and is deallocated when the object is closed. An application repeatedly asking for information can cause memory bloat until the object is closed. The application can optionally call **SCF_Session_freeInfo**(3SMARTCARD), **SCF_Terminal_freeInfo**(3SMARTCARD), or **SCF_Card_freeInfo**(3SMARTCARD) to cause immediate deallocation of the value. Applications must not use other means such as**free**(3C) to deallocate the memory.

Applications must not access values that have been deallocated. For example, accessing a Card's ATR after the card has been closed results in undefined behavior.

For a session, the valid property names and value types are:

terminalnames (pointer to **char** **)
The list of terminal names that can currently be used in this session. The returned value is an array of **char** *, each element of the list is a pointer to a terminal name. The end of the array is denoted by a null pointer. The first element of the list is the default terminal for the session, which will be used when **SCF_Session_getTerminal( )** is called with a null pointer for the terminal name.

For a terminal, the standard property names and value types are as follows. Some terminal drivers can define additional driver-specific properties.

name (pointer to **char** *)

The name of the terminal. If the default terminal was used (a null pointer was passed to **SCF_Session_getTerminal( )**), the value will contain the actual name of the default terminal. For example, "MyInternalCardReader".

type (pointer to **char** ∗)
: The type of the terminal. For example, "SunISCRI".

devname (pointer to **char** ∗)
: Information about how the device is attached to the system. This can be a UNIX device name (for example, "/dev/scmi2c0") or some other terminal-specific string describing its relation to the system.

For a card, the valid property names and value types are:

type (pointer to **char** ∗)
: The type of the smartcard, as recognized by the framework (For example, "Cyberflex"). If the framework does not recognize the card type, "UnknownCard" is returned.

atr (pointer to **struct SCF_BinaryData_t** ∗)
: The Answer To Reset (ATR) data returned by the card when it was last inserted or reset. The structure member **length** denotes how many bytes are in the ATR. The structure member **data** is a pointer to the actual ATR bytes.

Upon success, **SCF_STATUS_SUCCESS** is returned and *value* will contain the the requested information. Otherwise, an error value is returned and *value* remains unaltered.

These functions will fail if:

SCF_STATUS_BADARGS
: Either *name* or *value* is a null pointer.

SCF_STATUS_BADHANDLE
: The session, terminal, or card has been closed or is invalid.

SCF_STATUS_FAILED
: An internal error occurred.

SCF_STATUS_UNKNOWNPROPERTY
: The property specified by *name* was not found.

Example 1: Simple string information.

```
SCF_Status_t status;
SCF_Terminal_t myTerminal;
const char *myName, *myType;

/* (...call SCF_Session_getTerminal to open myTerminal...) */

status = SCF_Terminal_getInfo(myTerminal, "name", &myName);
if (status != SCF_STATUS_SUCCESS) exit(1);
status = SCF_Terminal_getInfo(myTerminal, "type", &myType);
if (status != SCF_STATUS_SUCCESS) exit(1);
```

printf("The terminal called %s is a %s\n", myName, myType);

Example 2: Display the names of all terminals available in the session.

```
SCF_Status_t status;
SCF_Session_t mySession;
const char **myList;  /* Technically "const char * const *". */
int i;

/* (...call SCF_Session_getSession to open mySession...) */

status = SCF_Session_getInfo(mySession, "terminalnames", &myList);
if (status != SCF_STATUS_SUCCESS) exit(1);

printf("The following terminals are available:\n");
for (i=0; myList[i] != NULL; i++) {
    printf("%d: %s\n", i, myList[i]);
}
```

Example 3: Display the card's ATR.

```
SCF_Status_t status;
SCF_Card_t myCard;
struct SCF_BinaryData_t *myATR;
int i;

/* (...call SCF_Terminal_getCard to open myCard...) */

status = SCF_Card_getInfo(myCard, "atr", &myATR);
if (status != SCF_STATUS_SUCCESS) exit(1);

printf("The card's ATR is: 0x");
for(i=0; i < myATR->length; i++) {
    printf("%02.2x", myATR->data[i]);
}
printf("\n");
```

See **attributes**(5) for descriptions of the following attributes:

tab() allbox; cw(2.750000i)| cw(2.750000i) lw(2.750000i)| lw(2.750000i). ATTRIBUTE TYPEATTRIBUTE VALUE Interface StabilityEvolving MT-LevelMT-Safe

libsmartcard(3LIB), **SCF_Session_freeInfo**(3SMARTCARD), **SCF_Session_getSession**(3SMARTCARD), **SCF_Session_getTerminal**(3SMARTCARD), **SCF_Terminal_getCard**(3SMARTCARD), **attributes**(5)

NAME

SCF_Card_lock, SCF_Card_unlock – perform mutex locking on a card

SYNOPSIS

cc [*flag*...] *file*... **-lsmartcard** [*library*...]
#include <**smartcard/scf.h**>

SCF_Status_t SCF_Card_lock(**SCF_Card_t** *card*, **unsigned int** *timeout*);

SCF_Status_t SCF_Card_unlock(**SCF_Card_t** *card*);

card The card (from **SCF_Terminal_getCard**(3SMARTCARD)) to be locked.

timeout
The maximum number of seconds **SCF_Card_lock( )** should wait for a card locked by another application to become unlocked. A value of 0 results in **SCF_Card_lock( )** returning immediately if a lock cannot be immediately obtained. A value of **SCF_TIMEOUT_MAX** results in **SCF_Card_lock( )** waiting forever to obtain a lock.

Locking a card allows an application to perform a multi-APDU transaction (that is, multiple calls to **SCF_Card_exchangeAPDU**(3SMARTCARD)) without interference from other smartcard applications. The lock is enforced by the server, so that other applications that attempt to call **SCF_Card_exchangeAPDU( )** or **SCF_Card_reset**(3SMARTCARD) will be denied access to the card. Applications should restrict use of locks only to brief critical sections. Otherwise it becomes difficult for multiple applications to share the same card.

When a lock is granted to a specific **SCF_Card_t** card object, only that object can be used to access the card and subsquently release the lock. If a misbehaving application holds a lock for an extended period, the lock can be broken by having the user remove and reinsert the smartcard.

It is an error to attempt to lock a card when the caller already holds a lock on the card (that is, calling **SCF_Card_lock( )** twice in a succession). Unlocking a card that is not locked (or was already unlocked) can be performed without causing an error.

An application might find that it is unable to lock the card, or communicate with it because **SCF_Card_exchangeAPDU( )** keeps returning **SCF_STATUS_CARDLOCKED**. If this situation persists, it might indicate that another application has not released its lock on the card. The user is able to forcably break a lock by removing the card and reinserting it, after which the application must call **SCF_Terminal_getCard**(3SMARTCARD) to access the "new" card. In this situation an application should retry for a reasonable period of time, and then alert the user that the operation could not be completed because the card is in use by another application and that removing or reinserting the card will resolve the problem.

If the card is successfully locked or unlocked, **SCF_STATUS_SUCCESS** is returned. Otherwise, the lock status of the card remains unchanged and an error value is returned.

The **SCF_Card_lock( )** and **SCF_Card_lock( )** functions will fail if:

SCF_STATUS_BADHANDLE
The specified card has been closed or is invalid.

SCF_STATUS_CARDLOCKED
There is a lock present on the card, but it is not held by the specified card object. For example, the caller is attempting to unlock a card locked by another application.

SCF_STATUS_CARDREMOVED
The card object cannot be used because the card represented by the **SCF_Card_t** has been removed.

SCF_STATUS_COMMERROR
The connection to the server was lost.

SCF_STATUS_DOUBLELOCK
The caller has already locked this card and is attempting to lock it again.

SCF_STATUS_FAILED
An internal error occured.

SCF_STATUS_TIMEOUT
The *timeout* expired before the call was able to obtain the lock.

Example 1: Use a card lock.

```
SCF_Status_t status;
SCF_Card_t myCard;

/* (...call SCF_Terminal_getCard to open myCard...) */

status = SCF_Card_lock(myCard, 15);
if (status == SCF_STATUS_TIMEOUT) {
    printf("Unable to get a card lock, someone else has a lock.\n");
    exit(0);
}
else if (status != SCF_STATUS_SUCCESS) exit(1);

/* Send the first APDU */
SCF_Card_exchangeAPDU(myCard, ...);

/* Send the second APDU */
SCF_Card_exchangeAPDU(myCard, ...);

status = SCF_Card_unlock(myCard);
```

/* ... */

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

libsmartcard(3LIB), **SCF_Card_exchangeAPDU**(3SMARTCARD), **SCF_Card_reset**(3SMARTCARD), **SCF_Terminal_getCard**(3SMARTCARD), **attributes**(5)

NAME

SCF_Card_reset – perform a reset of a smartcard

SYNOPSIS

cc [*flag*...] *file*... **-lsmartcard** [*library*...]
#include <**smartcard/scf.h**>

SCF_Status_t SCF_Card_reset(**SCF_Card_t** *card*);

card The card (from **SCF_Terminal_getCard**(3SMARTCARD)) to be reset

The **SCF_Card_reset( )** function causes the specified smartcard to be reset by the terminal.

A card can be reset only if it has not been locked (with **SCF_Card_lock**(3SMARTCARD)) by another client. A client wishing to reset a card should either first call **SCF_Card_lock( )** to obtain the card lock, or be prepared to retry the reset operation if it fails because another client holds the card lock.

When the card is reset, any **SCF_Card_t** object representing the card will continue to remain valid after the reset. When the reset occurs, an **SCF_EVENT_CARDRESET** event will be sent to all registered event listeners for the terminal (assuming they registered for this event). This is the only notification of a reset provided to SCF clients. When a client receives this event, it should be prepared to reinitialize any state on the card that might have been interrupted by the reset. New information about the card (for example, ATR, if it changed) can also be available from **SCF_Card_getInfo**(3SMARTCARD).

If the card is successfully reset, **SCF_STATUS_SUCCESS** is returned. Otherwise, the status of the card remains unchanged and an error value is returned.

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

SCF_STATUS_BADHANDLE
The specified card has been closed or is invalid.

SCF_STATUS_CARDLOCKED
The card cannot be reset because another client holds a lock on the card.

SCF_STATUS_CARDREMOVED
The card cannot be reset because the card represented by the **SCF_Card_t** has been removed.

SCF_STATUS_COMMERROR
The connection to the server was lost.

SCF_STATUS_FAILED
An internal error occured.

Example 1: Reset a card.

```
SCF_Status_t status;
SCF_Card_t myCard;

/* (...call SCF_Terminal_getCard to open myCard...) */
```

```
status = SCF_Card_lock(myCard, SCF_TIMEOUT_MAX);
if (status != SCF_STATUS_SUCCESS) exit(1);

status = SCF_Card_reset(myCard);
if (status != SCF_STATUS_SUCCESS) exit(1);

status = SCF_Card_unlock(myCard);
if (status != SCF_STATUS_SUCCESS) exit(1);

/* ... */
```

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

libsmartcard(3LIB), **SCF_Card_getInfo**(3SMARTCARD), **SCF_Card_lock**(3SMARTCARD), **SCF_Terminal_addEventListener**(3SMARTCARD), **SCF_Terminal_getCard**(3SMARTCARD), **attributes**(5)

NAME

SCF_Card_lock, SCF_Card_unlock – perform mutex locking on a card

SYNOPSIS

cc [*flag*...] *file*... **-lsmartcard** [*library*...]
#include <**smartcard/scf.h**>

SCF_Status_t SCF_Card_lock(**SCF_Card_t** *card*, **unsigned int** *timeout*);

SCF_Status_t SCF_Card_unlock(**SCF_Card_t** *card*);

card The card (from **SCF_Terminal_getCard**(3SMARTCARD)) to be locked.

timeout
The maximum number of seconds **SCF_Card_lock( )** should wait for a card locked by another application to become unlocked. A value of 0 results in **SCF_Card_lock( )** returning immediately if a lock cannot be immediately obtained. A value of **SCF_TIMEOUT_MAX** results in **SCF_Card_lock( )** waiting forever to obtain a lock.

Locking a card allows an application to perform a multi-APDU transaction (that is, multiple calls to **SCF_Card_exchangeAPDU**(3SMARTCARD)) without interference from other smartcard applications. The lock is enforced by the server, so that other applications that attempt to call **SCF_Card_exchangeAPDU( )** or **SCF_Card_reset**(3SMARTCARD) will be denied access to the card. Applications should restrict use of locks only to brief critical sections. Otherwise it becomes difficult for multiple applications to share the same card.

When a lock is granted to a specific **SCF_Card_t** card object, only that object can be used to access the card and subsquently release the lock. If a misbehaving application holds a lock for an extended period, the lock can be broken by having the user remove and reinsert the smartcard.

It is an error to attempt to lock a card when the caller already holds a lock on the card (that is, calling **SCF_Card_lock( )** twice in a succession). Unlocking a card that is not locked (or was already unlocked) can be performed without causing an error.

An application might find that it is unable to lock the card, or communicate with it because **SCF_Card_exchangeAPDU( )** keeps returning **SCF_STATUS_CARDLOCKED**. If this situation persists, it might indicate that another application has not released its lock on the card. The user is able to forcably break a lock by removing the card and reinserting it, after which the application must call **SCF_Terminal_getCard**(3SMARTCARD) to access the "new" card. In this situation an application should retry for a reasonable period of time, and then alert the user that the operation could not be completed because the card is in use by another application and that removing or reinserting the card will resolve the problem.

If the card is successfully locked or unlocked, **SCF_STATUS_SUCCESS** is returned. Otherwise, the lock status of the card remains unchanged and an error value is returned.

The **SCF_Card_lock( )** and **SCF_Card_lock( )** functions will fail if:

SCF_STATUS_BADHANDLE
The specified card has been closed or is invalid.

SCF_STATUS_CARDLOCKED
: There is a lock present on the card, but it is not held by the specified card object. For example, the caller is attempting to unlock a card locked by another application.

SCF_STATUS_CARDREMOVED
: The card object cannot be used because the card represented by the **SCF_Card_t** has been removed.

SCF_STATUS_COMMERROR
: The connection to the server was lost.

SCF_STATUS_DOUBLELOCK
: The caller has already locked this card and is attempting to lock it again.

SCF_STATUS_FAILED
: An internal error occured.

SCF_STATUS_TIMEOUT
: The *timeout* expired before the call was able to obtain the lock.

Example 1: Use a card lock.

```
SCF_Status_t status;
SCF_Card_t myCard;

/* (...call SCF_Terminal_getCard to open myCard...) */

status = SCF_Card_lock(myCard, 15);
if (status == SCF_STATUS_TIMEOUT) {
    printf("Unable to get a card lock, someone else has a lock.\n");
    exit(0);
}
else if (status != SCF_STATUS_SUCCESS) exit(1);

/* Send the first APDU */
SCF_Card_exchangeAPDU(myCard, ...);

/* Send the second APDU */
SCF_Card_exchangeAPDU(myCard, ...);

status = SCF_Card_unlock(myCard);
```

/* ... */

See **attributes**(5) for descriptions of the following attributes:

tab() allbox; cw(2.750000i)| cw(2.750000i) lw(2.750000i)| lw(2.750000i). ATTRIBUTE TYPEATTRIBUTE VALUE Interface StabilityEvolving MT-LevelMT-Safe

libsmartcard(3LIB), **SCF_Card_exchangeAPDU**(3SMARTCARD),
SCF_Card_reset(3SMARTCARD), **SCF_Terminal_getCard**(3SMARTCARD), **attributes**(5)

NAME

SCF_Terminal_waitForCardPresent, SCF_Terminal_waitForCardAbsent, SCF_Card_waitForCardRemoved – wait for a card to be inserted or removed

SYNOPSIS

cc [*flag...*] *file...* **-lsmartcard** [*library*...]
#include <**smartcard/scf.h**>

SCF_Status_t SCF_Terminal_waitForCardPresent(**SCF_Terminal_t** *terminal*, **unsigned int** *timeout*);

SCF_Status_t SCF_Terminal_waitForCardAbsent(**SCF_Terminal_t** *terminal*, **unsigned int** *timeout*);

SCF_Status_t SCF_Card_waitForCardRemoved(**SCF_Card_t** *card*, **unsigned int** *timeout*);

card A card that was returned from **SCF_Terminal_getCard**(3SMARTCARD).

terminal
A terminal that was returned from **SCF_Session_getTerminal**(3SMARTCARD).

timeout
The maximum number or seconds to wait for the desired state to be reached. If the timeout is 0, the function will immediately return **SCF_STATUS_TIMEOUT** if the terminal or card is not in the desired state. A timeout of **SCF_TIMEOUT_MAX** can be specified to indicate that the function should never timeout.

These functions determine if a card is currently available in the specified terminal.

The **SCF_Card_waitForCardRemoved( )** function differs from **SCF_Terminal_waitForCardAbsent( )** in that it checks to see if a specific card has been removed. If another card (or even the same card) has since been reinserted, **SCF_Card_waitForCardRemoved( )** will report that the old card was removed, while the **SCF_Terminal_waitForCardAbsent( )** will instead report that there is a card present.

If the desired state is already true, the function will immediately return **SCF_STATUS_SUCCESS**. Otherwise it will wait for a change to the desired state, or for the timeout to expire, whichever occurs first.

Unlike an event listener (**SCF_Terminal_addEventListener**(3SMARTCARD)), these functions return the state of the terminal, not just events. To use an electronics analogy, event listeners are edge-triggered, while these functions are level-triggered.

If the desired state is reached before the timeout expires, **SCF_STATUS_SUCCESS** is returned. If the timeout expires, **SCF_STATUS_TIMEOUT** is returned. Otherwise, an error value is returned.

These functions will fail if:

SCF_STATUS_BADHANDLE
The specified *terminal* or *card* has been closed or is invalid.

SCF_STATUS_COMMERROR
The server closed the connection.

SCF_STATUS_FAILED
An internal error occured.

Example 1: Determine if a card is currently inserted.

```
int isCardCurrentlyPresent(SCF_Terminal_t myTerminal) {
    SCF_Status_t status;

    /*
     * The timeout of zero makes sure this call will always
     * return immediately.
     */
    status = SCF_Terminal_waitForCardPresent(myTerminal, 0);

    if (status == SCF_STATUS_SUCCESS) return (TRUE);
    else if (status == SCF_STATUS_TIMEOUT) return (FALSE);

    /*
     * For other errors, this example just assumes no card
     * is present. We don’t really know.
     */
    return (FALSE);
}
```

Example 2: Remind the user every 5 seconds to remove their card.

```
SCF_Status_t status;
SCF_Terminal_t myTerminal;

/* (...call SCF_Session_getTerminal to open myTerminal...) */

status = SCF_Terminal_waitForCardAbsent(myTerminal, 0);
while (status == SCF_STATUS_TIMEOUT) {
    printf("Please remove the card from the terminal!\n");
    status = SCF_Terminal_waitForCardAbsent(myTerminal, 5);
}

if (status == SCF_STATUS_SUCCESS)
    printf("Thank you.\n");
else
    exit(1);
```

/* ... */

Example 3: Demonstrate the difference between the card-specific and terminal-specific calls.

```
SCF_Status_t status;
SCF_Terminal_t myTerminal;
SCF_Card_t myCard;

/* (...call SCF_Session_getTerminal to open myTerminal...) */

status = SCF_Terminal_getCard(myTerminal, &myCard);
if (status != SCF_STATUS_SUCCESS) exit(1);

/*
 * While we sleep, assume user removes the card
 * and inserts another card.
 */
sleep(10);

status = SCF_Terminal_waitForCardAbsent(myTerminal, 0);
/*
 * In this case, status is expected to be SCF_STATUS_TIMEOUT, as there
 * is a card present.
 */

status = SCF_Card_waitForCardRemoved(myCard, 0);
/*
 * In this case, status is expected to be SCF_STATUS_SUCCESS, as the
 * card returned from SCF_Terminal_getCard was indeed removed (even
 * though another card is currently in the terminal).
 */

/* ... */
```

See **attributes**(5) for descriptions of the following attributes:

tab() allbox; cw(2.750000i)| cw(2.750000i) lw(2.750000i)| lw(2.750000i). ATTRIBUTE TYPEATTRIBUTE VALUE Interface StabilityEvolving MT-LevelMT-Safe

libsmartcard(3LIB), **SCF_Session_getTerminal**(3SMARTCARD), **SCF_Terminal_addEventListener**(3SMARTCARD), **SCF_Terminal_getCard**(3SMARTCARD), **attributes**(5)

NAME

SCF_Session_close, SCF_Terminal_close, SCF_Card_close – close a smartcard session, terminal, or card

SYNOPSIS

cc [*flag...*] *file...* -**lsmartcard** [*library...*]
#include <**smartcard/scf.h**>

SCF_Status_t SCF_Session_close(**SCF_Session_t** *session*);

SCF_Status_t SCF_Terminal_close(**SCF_Terminal_t** *terminal*);

SCF_Status_t SCF_Card_close(**SCF_Card_t** *card*);

card An object that was returned from **SCF_Terminal_getCard**(3SMARTCARD)

session
An object that was returned from **SCF_Session_getSession**(3SMARTCARD)

terminal
An object that was returned from **SCF_Session_getTerminal**(3SMARTCARD)

These functions release the resources (memory, threads, and others) that were allocated within the library when the session, terminal, or card was opened. Any storage allocated by calls to **SCF_Session_getInfo**(3SMARTCARD), **SCF_Terminal_getInfo**(3SMARTCARD), or **SCF_Card_getInfo**(3SMARTCARD) is deallocated when the associated object is closed. Attempts to access results from these interfaces after the object has been closed results in undefined behavior.

If a card that was locked by **SCF_Card_lock**(3SMARTCARD) is closed, the lock is automatically released. When a terminal is closed, any event listeners on that terminal object are removed and any cards that were obtained with the terminal are closed. Similarly, closing a session will close any terminals or cards obtained with that session. These are the only cases where the library will automatically perform a close.

Once closed, a session, terminal, or card object can no longer be used by an SCF function. Any attempt to do so results in an **SCF_STATUS_BADHANDLE** error. The sole exception is that closing an object, even if already closed, is always a successful operation.

Closing a handle is always a successful operation that returns **SCF_STATUS_SUCCESS**. The library can safely detect handles that are invalid or already closed.

Example 1: Close each object explicitly.

```
SCF_Status_t status;
SCF_Session_t mySession;
SCF_Terminal_t myTerminal;
SCF_Card_t myCard;

status = SCF_Session_getSession(&mySession);
if (status != SCF_STATUS_SUCCESS) exit(1);
status = SCF_Session_getTerminal(mySession, NULL, &myTerminal);
```

```
if (status != SCF_STATUS_SUCCESS) exit(1);
status = SCF_Terminal_getCard(myTerminal, &myCard);
if (status != SCF_STATUS_SUCCESS) exit(1);

/* (Do interesting things with smartcard...) */

SCF_Card_close(myCard);
SCF_Terminal_close(myTerminal);
SCF_Session_close(mySession);
```

Example 2: Allow the library to close objects.

```
SCF_Status_t status;
SCF_Session_t mySession;
SCF_Terminal_t myTerminal;
SCF_Card_t myCard;

status = SCF_Session_getSession(&mySession);
if (status != SCF_STATUS_SUCCESS) exit(1);
status = SCF_Session_getTerminal(mySession, NULL, &myTerminal);
if (status != SCF_STATUS_SUCCESS) exit(1);
status = SCF_Terminal_getCard(myTerminal, &myCard);
if (status != SCF_STATUS_SUCCESS) exit(1);

/* (Do interesting things with smartcard...) */

SCF_Session_close(mySession);
/* myTerminal and myCard have been closed by the library. */
```

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

libsmartcard(3LIB), **SCF_Card_getInfo**(3SMARTCARD), **SCF_Card_lock**(3SMARTCARD), **SCF_Session_getInfo**(3SMARTCARD), **SCF_Session_getSession**(3SMARTCARD), **SCF_Session_getTerminal**(3SMARTCARD), **SCF_Terminal_getCard**(3SMARTCARD), **SCF_Terminal_getInfo**(3SMARTCARD), **attributes**(5)

NAME

SCF_Session_freeInfo, SCF_Terminal_freeInfo, SCF_Card_freeInfo – deallocate information storage

SYNOPSIS

cc [*flag...*] *file...* **-lsmartcard** [*library...*]
#include <**smartcard/scf.h**>

SCF_Status_t SCF_Session_freeInfo(**SCF_Session_t** *session*, **void** **value*);

SCF_Status_t SCF_Terminal_freeInfo(**SCF_Terminal_t** *terminal*, **void** **value*);

SCF_Status_t SCF_Card_freeInfo(**SCF_Card_t** *card*, **void** **value*);

card An object that was returned from **SCF_Terminal_getCard**(3SMARTCARD). This object must be associated with the information value being freed.

session
An object that was returned from **SCF_Session_getSession**(3SMARTCARD). This object must be associated with the information value being freed.

terminal
An object that was returned from **SCF_Session_getTerminal**(3SMARTCARD). This object must be associated with the information value being freed.

value A pointer that was returned from a call to **SCF_Session_getInfo**(3SMARTCARD), **SCF_Terminal_getInfo**(3SMARTCARD), or **SCF_Card_getInfo**(3SMARTCARD).

When information is requested for an object (for example, by using **SCF_Session_getInfo( )**), the result is placed in memory allocated for that request. This memory must eventually be deallocated, or a memory leak will result. The deallocation of memory can occur in one of two ways.

- The simplest method is to allow the **smartcard** library to automatically deallocate memory when the object associated with the information is closed. For example, when **SCF_Card_close**(3SMARTCARD) is called, any information obtained from **SCF_Card_getInfo( )** for that card object is deallocated. The application is not required to call **SCF_Card_freeInfo( )** at all.
- If the object persists for a long period of time, the application can explicitly request the information to be deallocated without closing the object, so that memory is not wasted on unneeded storage. Similarly, if an application repeatedly requests information about an object (even the same information), the application can explicitly request deallocation as needed, so that memory usage does not continue to increase until the object is closed. In general, requesting information to be deallocated can be used to reduce runtime memory bloat.

Attempts to access deallocated memory result in undefined behavior.

If the information is successfully deallocated, **SCF_STATUS_SUCCESS** is returned. Otherwise, an error value is returned.

These functions will fail if:

SCF_STATUS_BADARGS
The specified value cannot be deallocated, possibly because of an invalid pointer, a value already deallocated, or because the value is not associated with the specified session, terminal, or card.

SCF_STATUS_BADHANDLE
The specified session, terminal, or card has been closed or is invalid.

SCF_STATUS_FAILED
An internal error occured.

Example 1: Free information.

```
char *terminalName;
SCF_Status_t status;
SCF_Terminal_t myTerminal;

/* (...call SCF_Session_getTerminal to open myTerminal...) */

status = SCF_Terminal_getInfo(myTerminal, "name", &terminalName);
if (status != SCF_STATUS_SUCCESS) exit(1);

printf("The terminal name is %s\n", terminalName);

status = SCF_Terminal_freeInfo(myTerminal, terminalName);
if (status != SCF_STATUS_SUCCESS) exit(1);
```

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

libsmartcard(3LIB), **SCF_Session_getInfo**(3SMARTCARD), **SCF_Session_getSession**(3SMARTCARD), **SCF_Session_getTerminal**(3SMARTCARD), **SCF_Terminal_getCard**(3SMARTCARD), **attributes**(5)

NAME

SCF_Session_getInfo, SCF_Terminal_getInfo, SCF_Card_getInfo – retrieve information about a session, terminal, or card

SYNOPSIS

cc [*flag...*] *file...* **-lsmartcard** [*library...*]
#include <**smartcard/scf.h**>

SCF_Status_t SCF_Session_getInfo(**SCF_Session_t** *session*, **const char** *∗name*, **void** *∗value*);

SCF_Status_t SCF_Terminal_getInfo(**SCF_Terminal_t** *terminal*, **const char** *∗name*, **void** *∗value*);

SCF_Status_t SCF_Card_getInfo(**SCF_Card_t** *card*, **const char** *∗name*, **void** *∗value*);

card An object that was returned from **SCF_Terminal_getCard**(3SMARTCARD).

name The name of a property for which a value is to be returned. The name is case-sensitive.

session
An object that was returned from **SCF_Session_getSession**(3SMARTCARD).

terminal
An object that was returned from **SCF_Session_getTerminal**(3SMARTCARD).

value The value of the property. The actual type of the value depends on what property was being queried.

These functions obtain information about a session, terminal, or card. The information returned represents the current state of the object and can change between calls.

Each call allocates new storage for the returned result. This storage is tracked internally and is deallocated when the object is closed. An application repeatedly asking for information can cause memory bloat until the object is closed. The application can optionally call **SCF_Session_freeInfo**(3SMARTCARD), **SCF_Terminal_freeInfo**(3SMARTCARD), or **SCF_Card_freeInfo**(3SMARTCARD) to cause immediate deallocation of the value. Applications must not use other means such as**free**(3C) to deallocate the memory.

Applications must not access values that have been deallocated. For example, accessing a Card’s ATR after the card has been closed results in undefined behavior.

For a session, the valid property names and value types are:

terminalnames (pointer to **char** ∗∗)
The list of terminal names that can currently be used in this session. The returned value is an array of **char** ∗, each element of the list is a pointer to a terminal name. The end of the array is denoted by a null pointer. The first element of the list is the default terminal for the session, which will be used when **SCF_Session_getTerminal( )** is called with a null pointer for the terminal name.

For a terminal, the standard property names and value types are as follows. Some terminal drivers can define additional driver-specific properties.

name (pointer to **char** ∗)

The name of the terminal. If the default terminal was used (a null pointer was passed to **SCF_Session_getTerminal( )**), the value will contain the actual name of the default terminal. For example, "MyInternalCardReader".

type (pointer to **char** ∗)
: The type of the terminal. For example, "SunISCRI".

devname (pointer to **char** ∗)
: Information about how the device is attached to the system. This can be a UNIX device name (for example, "/dev/scmi2c0") or some other terminal-specific string describing its relation to the system.

For a card, the valid property names and value types are:

type (pointer to **char** ∗)
: The type of the smartcard, as recognized by the framework (For example, "Cyberflex"). If the framework does not recognize the card type, "UnknownCard" is returned.

atr (pointer to **struct SCF_BinaryData_t** ∗)
: The Answer To Reset (ATR) data returned by the card when it was last inserted or reset. The structure member **length** denotes how many bytes are in the ATR. The structure member **data** is a pointer to the actual ATR bytes.

Upon success, **SCF_STATUS_SUCCESS** is returned and *value* will contain the the requested information. Otherwise, an error value is returned and *value* remains unaltered.

These functions will fail if:

SCF_STATUS_BADARGS
: Either *name* or *value* is a null pointer.

SCF_STATUS_BADHANDLE
: The session, terminal, or card has been closed or is invalid.

SCF_STATUS_FAILED
: An internal error occurred.

SCF_STATUS_UNKNOWNPROPERTY
: The property specified by *name* was not found.

Example 1: Simple string information.

```
SCF_Status_t status;
SCF_Terminal_t myTerminal;
const char *myName, *myType;

/* (...call SCF_Session_getTerminal to open myTerminal...) */

status = SCF_Terminal_getInfo(myTerminal, "name", &myName);
if (status != SCF_STATUS_SUCCESS) exit(1);
status = SCF_Terminal_getInfo(myTerminal, "type", &myType);
if (status != SCF_STATUS_SUCCESS) exit(1);
```

printf("The terminal called %s is a %s\n", myName, myType);

Example 2: Display the names of all terminals available in the session.

```
SCF_Status_t status;
SCF_Session_t mySession;
const char **myList;  /* Technically "const char * const *". */
int i;

/* (...call SCF_Session_getSession to open mySession...) */

status = SCF_Session_getInfo(mySession, "terminalnames", &myList);
if (status != SCF_STATUS_SUCCESS) exit(1);

printf("The following terminals are available:\n");
for (i=0; myList[i] != NULL; i++) {
    printf("%d: %s\n", i, myList[i]);
}
```

Example 3: Display the card's ATR.

```
SCF_Status_t status;
SCF_Card_t myCard;
struct SCF_BinaryData_t *myATR;
int i;

/* (...call SCF_Terminal_getCard to open myCard...) */

status = SCF_Card_getInfo(myCard, "atr", &myATR);
if (status != SCF_STATUS_SUCCESS) exit(1);

printf("The card's ATR is: 0x");
for(i=0; i < myATR->length; i++) {
    printf("%02.2x", myATR->data[i]);
}
printf("\n");
```

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

libsmartcard(3LIB), **SCF_Session_freeInfo**(3SMARTCARD), **SCF_Session_getSession**(3SMARTCARD), **SCF_Session_getTerminal**(3SMARTCARD), **SCF_Terminal_getCard**(3SMARTCARD), **attributes**(5)

NAME

SCF_Session_getSession – establish a context with a system's smartcard framework

SYNOPSIS

cc [*flag*...] *file*... **-lsmartcard** [*library*...]
#include <**smartcard/scf.h**>

SCF_Status_t SCF_Session_getSession(**SCF_Session_t** **session*);

session
A pointer to an **SCF_Session_t**. If a session is successfully established, the session will be returned through this parameter.

The **SCF_Session_getSession( )** function establishes a session with the Solaris Smart Card Framework (SCF). Once a session has been opened, the session can be used with **SCF_Session_getTerminal**(3SMARTCARD) to access a smartcard terminal (reader). Information about the session can be obtained by calling **SCF_Session_getInfo**(3SMARTCARD).

When the session is no longer needed, **SCF_Session_close**(3SMARTCARD) should be called to end the session and release session resources. Closing a session will also close any terminals and cards opened within the session.

An application usually needs to open only a single session. For example, multiple terminals can be opened from the same session. If an appication opens additional sessions, each call will return independent (different) sessions.

Upon success, **SCF_STATUS_SUCCESS** is returned and *session* contains a valid session. If a session could not be established, an error value is returned and *session* remains unaltered.

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

SCF_STATUS_BADARGS
The *session* argument is a null pointer.

SCF_STATUS_COMMERROR
The library was unable to contact the smartcard server daemon (**ocfserv**(1M)), or the library was unable to obtain a session from the server.

SCF_STATUS_FAILED
An internal error occurred.

Example 1: Establish a session with the framework.

```
SCF_Status_t status;
SCF_Session_t mySession;

status = SCF_Session_getSession(&mySession);
if (status != SCF_STATUS_SUCCESS) exit(1);
```

/* Proceed with other smartcard operations. */

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

libsmartcard(3LIB), **SCF_Session_close**(3SMARTCARD), **SCF_Session_getInfo**(3SMARTCARD), **SCF_Session_getTerminal**(3SMARTCARD), **attri-butes**(5)

NAME

SCF_Session_getTerminal – establish a context with a smartcard terminal (reader)

SYNOPSIS

cc [*flag*...] *file*... **-lsmartcard** [*library*...]
#include <**smartcard/scf.h**>

SCF_Status_t SCF_Session_getTerminal(**SCF_Session_t** *session*, **const char** **terminalName*, **SCF_Terminal_t** **terminal*);

session
The session (from **SCF_Session_getSession**(3SMARTCARD)) containing a terminal to be opened.

terminal
A pointer to an **SCF_Terminal_t**. If the terminal is successfully opened, a handle for the terminal will be returned through this parameter.

terminalName
Specifies the name of the terminal to access. If *terminalName* is a null pointer, it indicates that the library should connect with the default terminal for the session.

The **SCF_Session_getTerminal( )** function establishes a context with a specific smartcard terminal (also known as a reader) in the session. Terminal objects are used for detecting card movement (insertion or removal) and to create card objects for accessing a specific card.

The list of available terminal names can be retrieved by calling **SCF_Session_getInfo**(3SMARTCARD). Unless the user explicitly requests a specific terminal, applications should use the session's default terminal by calling **SCF_Session_getTerminal( )** with a null pointer for the terminal name. This eliminates the need to first process an available-terminal list with just one element on systems with only a single smartcard terminal. On multi-terminal systems, the user can preconfigure one of the terminals as the default (or preferred) terminal. See USAGE below.

If **SCF_Session_getTerminal( )** is called multiple times in the same session to access the same physical terminal, the same **SCF_Terminal_t** will be returned in each call. Multithreaded applications must take care to avoid having one thread close a terminal that is still needed by another thread. This can be accomplished by coordination within the application or by having each thread open a seperate session to avoid interference.

When the terminal is no longer needed, **SCF_Terminal_close**(3SMARTCARD) should be called to release terminal resources. Closing a terminal will also close any cards opened from the terminal.

Upon success, **SCF_STATUS_SUCCESS** is returned and *terminal* contains the opened terminal. Otherwise, an error value is returned and *terminal* remains unaltered.

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

SCF_STATUS_BADARGS
The *terminal* argument is a null pointer.

SCF_STATUS_BADHANDLE
The session was closed or is invalid.

SCF_STATUS_BADTERMINAL
The specified *terminalName* is not valid for this session, or the default terminal could not be opened because there are no terminals available in this session.

SCF_STATUS_COMMERROR
The connection to the server was lost.

SCF_STATUS_FAILED
An internal error occurred.

Example 1: Use the default terminal.

```
SCF_Status_t status;
SCF_Session_t mySession;
SCF_Terminal_t myTerminal;
char *myName;

/* (...call SCF_Session_getSession to open mySession...) */

status = SCF_Session_getTerminal(mySession, NULL, &myTerminal);
if (status != SCF_STATUS_SUCCESS) exit(1);

status = SCF_Terminal_getInfo(myTerminal, "name", &myName);
if (status != SCF_STATUS_SUCCESS) exit(1);

printf("Please insert a card into the terminal named %s\n", myName);
```

/* ... */

Example 2: Open a terminal by name.

```
SCF_Status_t status;
SCF_Session_t mySession;
SCF_Terminal_t myTerminal;
char *myName;

/* (...call SCF_Session_getSession to open mySession...) */

/*
 * The name should be selected from the list of terminal names
 * available from SCF_Session_getInfo, but it could also be
 * read from an appliation’s config file or from user input.
 */
myName = "SunInternalReader";

status = SCF_Session_getTerminal(mySession, myName, &myTerminal);
if (status == SCF_STATUS_BADTERMINAL) {
    printf("There is no terminal named %s.\n", myName);
    exit(1);
} else if (status != SCF_STATUS_SUCCESS) exit(2);

/* ... */
```

When using the Solaris OCF smartcard framework, the default reader is specified by the **ocf.client.default.defaultreader** property. If this property is not set, the first available reader is chosen as the default. Users can set the **SCF_DEFAULT_TERMINAL** environment variable to the name of a terminal to override the normal default. The **smartcard** utility can also be used to add terminals to or remove terminals from the system. See **smartcard**(1M) for information on how to add or modify the OCF property.

Terminals can be accessed only by the user who expected to have physical access to the terminal. By default, this user is assumed to be the owner of **/dev/console** and the superuser. Certain terminals such as Sun Ray appliances can use a different method to restrict access to the terminal.

The framework also uses the **DISPLAY** environment variable to further restrict which terminals are listed for a user. By default, terminals are associated with the ":0" display. Sun Ray terminals are associated with the display for that session, for example ":25". If the **DISPLAY** environment variable is not set or is a display on another host, it is treated as though it were set to ":0". Terminals not associated with the user's **DISPLAY** are not listed. To override this behaviour, the **SCF_FILTER_KEY** environment variable can be set to the desired display, for example ":0", ":25", and so on. To list all terminals to which a user has access, **SCF_FILTER_KEY** can be set to the special value of ":*".

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

smartcard(1M), **libsmartcard**(3LIB), **SCF_Session_getInfo**(3SMARTCARD), **SCF_Session_getSession**(3SMARTCARD), **SCF_Terminal_close**(3SMARTCARD), **attributes**(5)

NAME

SCF_Terminal_addEventListener, SCF_Terminal_updateEventListener, SCF_Terminal_removeEventListener – receive asychronous event notification

SYNOPSIS

cc [*flag...*] *file...* **-lsmartcard** [*library...*]
#include <**smartcard/scf.h**>

SCF_Status_t SCF_Terminal_addEventListener(**SCF_Terminal_t** *terminal*, **SCF_Event_t** *events*, **void(****callback)(SCF_Event_t, SCF_Terminal_t, void *)*, **void** **userData*, **SCF_ListenerHandle_t** **listenerHandle*);

SCF_Status_t SCF_Terminal_updateEventListener(**SCF_Terminal_t** *terminal*, **SCF_ListenerHandle_t** *listenerHandle*, **SCF_Event_t** *events*);

SCF_Status_t SCF_Terminal_removeEventListener(**SCF_Terminal_t** *terminal*, **SCF_ListenerHandle_t** *listenerHandle*);

terminal
A terminal (from **SCF_Session_getTerminal**(3SMARTCARD)) to which the event listener should be added or removed.

events Events to deliver to the callback. An event will not be delivered if it is not listed. The caller can register for multiple events by performing a bitwise OR of the desired events. The valid events are:

SCF_EVENT_ALL
All of the events listed below will be delivered.

SCF_EVENT_CARDINSERTED
A smartcard was inserted into the terminal.

SCF_EVENT_CARDREMOVED
A smartcard was removed from the terminal.

SCF_EVENT_CARDPRESENT
Indicates that a card was present in the terminal when the event listener was first added. This event allows event listeners to determine the initial state of the terminal before an insert or remove event occurs. Either this event or the **SCF_EVENT_CARDABSENT** (see below) event will be delivered only once upon adding an event listener and immediately before any other events are delivered. Future card movements will generate **SCF_EVENT_CARDINSERTED** and **SCF_EVENT_CARDREMOVED** events, but not **SCF_EVENT_CARDPRESENT** or **SCF_EVENT_CARDABSENT** events. An event listener can assume that if a **SCF_EVENT_CARDPRESENT** event is delivered, the next card movement event will be a **SCF_EVENT_CARDREMOVED**.

SCF_EVENT_CARDABSENT
Indicates that a card was not present in the terminal when the event listener was first added. This event allows event listeners to determine the initial state of the terminal before an insert or remove event occurs. Either this event or the **SCF_EVENT_CARDPRESENT** event (see above) will be

delivered only once upon adding an event listener and immediately before any other events are delivered. Future card movements will generate **SCF_EVENT_CARDINSERTED** and **SCF_EVENT_CARDREMOVED** events, but not **SCF_EVENT_CARDPRESENT** or **SCF_EVENT_CARDABSENT** events. An event listener can assume that if a **SCF_EVENT_CARDABSENT** event is delivered, the next card movement event will be a **SCF_EVENT_CARDINSERTED**.

SCF_EVENT_CARDRESET
The smartcard currently present has been reset (see **SCF_Card_reset**(3SMARTCARD)).

SCF_EVENT_TERMINALCLOSED
The terminal is in the process of being closed (due to a call to **SCF_Session_close**(3SMARTCARD) or **SCF_Terminal_close**(3SMARTCARD)), so no further events will be delivered. The *terminal* argument provided to the callback will still be valid.

SCF_EVENT_COMMERROR
The connection to the server has been lost. No further events will be delivered.

callback
A function pointer that will be executed when the desired event occurs. The function must take three arguments. The first is a **SCF_Event_t** containing the event that occured. The second argument is an **SCF_Terminal_t** containing the terminal on which the event occured. The third is a **void** ∗ that can be used to provide arbitrary data to the *callback* when it is executed.

userData
A pointer to arbitrary user data. The data is not accessed by the library. The pointer is simply provided to the callback when an event is issued. This argument can safely be set to *NULL* if not needed. The callback must be able to handle this case.

listenerHandle
A unique "key" that is provided by **SCF_Terminal_addEventListener( )** to refer to a specific event listener registration. This allows multiple event listeners to be selectivly updated or removed.

These functions allow an application to receive notification of events on a terminal as they occur. The concept is similar to a signal handler. When an event occurs, a thread in the SCF library will execute the provided *callback* function. Once added, the listener will receive events until it is removed or either the terminal or session is closed.

When the callback function is executed, the callback arguments specify the event that occured and the terminal on which it occurred. Additionally, each callback will receive the *userData* pointer that was provided when the listener was added. The library does not make a copy of the memory pointed to by *userData*, so applications must take care not to

deallocate that memory until it is known that the callback will no longer access it (for example, by removing the event listener). Each invocation of the callback will be for exactly one event. If the library needs to deliver multiple events, they will be dispatched one at a time. Because the callback is executed from a thread, any operations it performs must be thread safe. For each callback registration, the library creates a new thread to deliver events to that callback. The callback is expected to perform minimal work and return quickly.

An application can add multiple callbacks on a terminal. Any event that occurs will be delivered to all listeners that registered for that event type. The same callback can be registered multiple times. Each call to **SCF_Terminal_addEventListener( )** will result in a new **SCF_ListenerHandle_t**. The events a callback receives can be changed by calling **SCF_Session_updateEventListener( )** with the handle that was returned when the listener was initially added. If the listener is set to receive no events (that is, the events parameter has no bits set), the listener will remain registered but will not receive any events. To remove a listener and release allocated resources, use **SCF_Terminal_removeEventListener( )** or close the terminal.

If the event listener was successfully added or removed, **SCF_STATUS_SUCCESS** is returned. Otherwise, an error value is returned and the internal list of registered event listeners remains unaltered.

These functions will fail if:

SCF_STATUS_BADARGS
The callback function pointer and / or *listenerHandle* is null, or an unknown event was specified.

SCF_STATUS_BADHANDLE
The specified terminal has been closed or is invalid, or the event listener handle could not be found to update or remove.

SCF_STATUS_COMMERROR
The connection to the server was lost.

SCF_STATUS_FAILED
An internal error occurred.

Example 1: Register for card movements.

```
struct myState_t {
    int isStateKnown;
    int isCardPresent;
};

void myCallback(SCF_Event_t event, SCF_Terminal_t eventTerminal,
    void *data) {
    struct myState_t *state = data;
    if (event == SCF_EVENT_CARDINSERTED) {
        printf("--- Card inserted ---\n");
        state->isCardPresent = 1;
    }
    else if (event == SCF_EVENT_CARDREMOVED) {
        printf("--- Card removed ---\n");
        state->isCardPresent = 0;
    }
    state->isStateKnown = 1;
}

main() {
    SCF_Status_t status;
    SCF_Terminal_t myTerminal;
    SCF_ListenerHandle_t myListener;
    struct myState_t myState;

    /* (...call SCF_Session_getTerminal to open myTerminal...) */

    myState.isStateKnown = 0;
    status = SCF_Terminal_addEventListener(myTerminal,
        SCF_EVENT_CARDINSERTED|SCF_EVENT_CARDREMOVED, &myCallback,
        &myState, &myListener);
    if (status != SCF_STATUS_SUCCESS) exit(1);

    while(1) {
        if (!myState.isStateKnown)
            printf("Waiting for first event...\n");
        else {
            if (myState.isCardPresent)
                printf("Card is present.\n");
            else
                printf("Card is not present.\n");
        }
        sleep(1);
    }
}
```

Example 2: Use different callbacks for each event.

```
void myInsertCallback(SCF_Event_t event, SCF_Terminal_t eventTerminal,
    void *data) {

    /* ... */
}

void myRemoveCallback(SCF_Event_t event, SCF_Terminal_t eventTerminal,
    void *data) {
    /* ... */
}
```

```
main () {
    SCF_Status_t status;
    SCF_Terminal_t terminal;
    SCF_ListenerHandle_t myListener1, myListener2, myListener3;
    int foo, bar;

    /* (...call SCF_Session_getTerminal to open myTerminal...) */

    status = SCF_Terminal_addEventListener(myTerminal,
        SCF_EVENT_CARDINSERTED, &myInsertCallback, &foo,
        &myListener1);
    if (status != SCF_STATUS_SUCCESS) exit(1);

    status = SCF_Terminal_addEventListener(myTerminal,
        SCF_EVENT_CARDREMOVED, &myRemoveCallback, &foo,
        &myListener2);
    if (status != SCF_STATUS_SUCCESS) exit(1);

    status = SCF_Terminal_addEventListener(myTerminal,
        SCF_EVENT_CARDREMOVED, &myRemoveCallback, &bar,
        &myListener3);
    if (status != SCF_STATUS_SUCCESS) exit(1);

    /*
     * At this point, when each insertion occurs, myInsertCallback
     * will be called once (with a pointer to foo). When each removal
     * occurs, myRemoveCallback will be called twice. One call will
     * be given a pointer to foo, and the other will be given a
     * pointer to bar.
     */

    status = SCF_Terminal_removeEventListener(myTerminal,
        myListener2);
    if (status != SCF_STATUS_SUCCESS) exit(1);

    /*
     * Now, when a removal occurs, myRemoveCallback will only be
     * called once, with a pointer to bar.
     */

    /* ... */
}
```

Example 3: Use initial state events to show user the terminal state in a GUI.

```
void myCallback(SCF_Event_t event, SCF_Terminal_t eventTerminal,
    void *unused) {
    if (event == SCF_EVENT_CARDPRESENT) {
        /* Set initial icon to a terminal with a card present. */
    }
    else if (event == SCF_EVENT_CARDABSENT) {
        /* Set initial icon to a terminal without a card present. */
    }
    else if (event == SCF_EVENT_CARDINSERTED) {
        /* Show animation for card being inserted into a terminal. */
    }
    else if (event == SCF_EVENT_CARDREMOVED) {
        /* Show animation for card being removed from a terminal. */
    }
}

main() {
    SCF_Terminal_t myTerminal;
    SCF_ListenerHandle_t myListener;

    /* (...call SCF_Session_getTerminal to open myTerminal...) */

    status = SCF_Terminal_addEventListener(myTerminal,
        SCF_EVENT_ALL, &myCallback, NULL, &myListener);
    if (status != SCF_STATUS_SUCCESS) exit(1);


    /* ... */
}
```

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

libsmartcard(3LIB), **SCF_Card_reset**(3SMARTCARD), **SCF_Session_close**(3SMARTCARD), **SCF_Session_getTerminal**(3SMARTCARD), **SCF_Session_updateEventListener**(3SMARTCARD), **SCF_Terminal_close**(3SMARTCARD), **SCF_Terminal_removeEventListener**(3SMARTCARD), **attributes**(5)

NAME

SCF_Session_close, SCF_Terminal_close, SCF_Card_close – close a smartcard session, terminal, or card

SYNOPSIS

cc [*flag...*] *file...* **-lsmartcard** [*library...*]
#include <**smartcard/scf.h**>

SCF_Status_t SCF_Session_close(**SCF_Session_t** *session*);

SCF_Status_t SCF_Terminal_close(**SCF_Terminal_t** *terminal*);

SCF_Status_t SCF_Card_close(**SCF_Card_t** *card*);

card An object that was returned from **SCF_Terminal_getCard**(3SMARTCARD)

session
An object that was returned from **SCF_Session_getSession**(3SMARTCARD)

terminal
An object that was returned from **SCF_Session_getTerminal**(3SMARTCARD)

These functions release the resources (memory, threads, and others) that were allocated within the library when the session, terminal, or card was opened. Any storage allocated by calls to **SCF_Session_getInfo**(3SMARTCARD), **SCF_Terminal_getInfo**(3SMARTCARD), or **SCF_Card_getInfo**(3SMARTCARD) is deallocated when the associated object is closed. Attempts to access results from these interfaces after the object has been closed results in undefined behavior.

If a card that was locked by **SCF_Card_lock**(3SMARTCARD) is closed, the lock is automatically released. When a terminal is closed, any event listeners on that terminal object are removed and any cards that were obtained with the terminal are closed. Similarly, closing a session will close any terminals or cards obtained with that session. These are the only cases where the library will automatically perform a close.

Once closed, a session, terminal, or card object can no longer be used by an SCF function. Any attempt to do so results in an **SCF_STATUS_BADHANDLE** error. The sole exception is that closing an object, even if already closed, is always a successful operation.

Closing a handle is always a successful operation that returns **SCF_STATUS_SUCCESS**. The library can safely detect handles that are invalid or already closed.

Example 1: Close each object explicitly.

```
SCF_Status_t status;
SCF_Session_t mySession;
SCF_Terminal_t myTerminal;
SCF_Card_t myCard;

status = SCF_Session_getSession(&mySession);
if (status != SCF_STATUS_SUCCESS) exit(1);
status = SCF_Session_getTerminal(mySession, NULL, &myTerminal);
```

```
if (status != SCF_STATUS_SUCCESS) exit(1);
status = SCF_Terminal_getCard(myTerminal, &myCard);
if (status != SCF_STATUS_SUCCESS) exit(1);

/* (Do interesting things with smartcard...) */

SCF_Card_close(myCard);
SCF_Terminal_close(myTerminal);
SCF_Session_close(mySession);
```

Example 2: Allow the library to close objects.

```
SCF_Status_t status;
SCF_Session_t mySession;
SCF_Terminal_t myTerminal;
SCF_Card_t myCard;

status = SCF_Session_getSession(&mySession);
if (status != SCF_STATUS_SUCCESS) exit(1);
status = SCF_Session_getTerminal(mySession, NULL, &myTerminal);
if (status != SCF_STATUS_SUCCESS) exit(1);
status = SCF_Terminal_getCard(myTerminal, &myCard);
if (status != SCF_STATUS_SUCCESS) exit(1);

/* (Do interesting things with smartcard...) */

SCF_Session_close(mySession);
/* myTerminal and myCard have been closed by the library. */
```

See **attributes**(5) for descriptions of the following attributes:

tab() allbox; cw(2.750000i)| cw(2.750000i) lw(2.750000i)| lw(2.750000i). ATTRIBUTE TYPEATTRIBUTE VALUE Interface StabilityEvolving MT-LevelMT-Safe

libsmartcard(3LIB), **SCF_Card_getInfo**(3SMARTCARD), **SCF_Card_lock**(3SMARTCARD), **SCF_Session_getInfo**(3SMARTCARD), **SCF_Session_getSession**(3SMARTCARD), **SCF_Session_getTerminal**(3SMARTCARD), **SCF_Terminal_getCard**(3SMARTCARD), **SCF_Terminal_getInfo**(3SMARTCARD), **attributes**(5)

NAME

SCF_Session_freeInfo, SCF_Terminal_freeInfo, SCF_Card_freeInfo – deallocate information storage

SYNOPSIS

cc [*flag...*] *file...* **-lsmartcard** [*library...*]
#include <**smartcard/scf.h**>

SCF_Status_t SCF_Session_freeInfo(**SCF_Session_t** *session*, **void** **value*);

SCF_Status_t SCF_Terminal_freeInfo(**SCF_Terminal_t** *terminal*, **void** **value*);

SCF_Status_t SCF_Card_freeInfo(**SCF_Card_t** *card*, **void** **value*);

card	An object that was returned from **SCF_Terminal_getCard**(3SMARTCARD). This object must be associated with the information value being freed.
session	An object that was returned from **SCF_Session_getSession**(3SMARTCARD). This object must be associated with the information value being freed.
terminal	An object that was returned from **SCF_Session_getTerminal**(3SMARTCARD). This object must be associated with the information value being freed.
value	A pointer that was returned from a call to **SCF_Session_getInfo**(3SMARTCARD), **SCF_Terminal_getInfo**(3SMARTCARD), or **SCF_Card_getInfo**(3SMARTCARD).

When information is requested for an object (for example, by using **SCF_Session_getInfo( )**), the result is placed in memory allocated for that request. This memory must eventually be deallocated, or a memory leak will result. The deallocation of memory can occur in one of two ways.

- The simplest method is to allow the **smartcard** library to automatically deallocate memory when the object associated with the information is closed. For example, when **SCF_Card_close**(3SMARTCARD) is called, any information obtained from **SCF_Card_getInfo( )** for that card object is deallocated. The application is not required to call **SCF_Card_freeInfo( )** at all.
- If the object persists for a long period of time, the application can explicitly request the information to be deallocated without closing the object, so that memory is not wasted on unneeded storage. Similarly, if an application repeatedly requests information about an object (even the same information), the application can explicitly request deallocation as needed, so that memory usage does not continue to increase until the object is closed. In general, requesting information to be deallocated can be used to reduce runtime memory bloat.

Attempts to access deallocated memory result in undefined behavior.

If the information is successfully deallocated, **SCF_STATUS_SUCCESS** is returned. Otherwise, an error value is returned.

These functions will fail if:

SCF_STATUS_BADARGS
The specified value cannot be deallocated, possibly because of an invalid pointer, a value already deallocated, or because the value is not associated with the specified session, terminal, or card.

SCF_STATUS_BADHANDLE
The specified session, terminal, or card has been closed or is invalid.

SCF_STATUS_FAILED
An internal error occured.

Example 1: Free information.

```
char *terminalName;
SCF_Status_t status;
SCF_Terminal_t myTerminal;

/* (...call SCF_Session_getTerminal to open myTerminal...) */

status = SCF_Terminal_getInfo(myTerminal, "name", &terminalName);
if (status != SCF_STATUS_SUCCESS) exit(1);

printf("The terminal name is %s\n", terminalName);

status = SCF_Terminal_freeInfo(myTerminal, terminalName);
```
if (status != SCF_STATUS_SUCCESS) exit(1);

See **attributes**(5) for descriptions of the following attributes:

tab() allbox; cw(2.750000i)| cw(2.750000i) lw(2.750000i)| lw(2.750000i). ATTRIBUTE TYPEATTRIBUTE VALUE Interface StabilityEvolving MT-LevelMT-Safe

libsmartcard(3LIB), **SCF_Session_getInfo**(3SMARTCARD), **SCF_Session_getSession**(3SMARTCARD), **SCF_Session_getTerminal**(3SMARTCARD), **SCF_Terminal_getCard**(3SMARTCARD), **attributes**(5)

NAME

SCF_Terminal_getCard – establish a context with a smartcard

SYNOPSIS

cc [*flag*...] *file*... **-lsmartcard** [*library*...]
#include <**smartcard/scf.h**>

SCF_Status_t SCF_Terminal_getCard(**SCF_Terminal_t** *terminal*, **SCF_Card_t** **card*);

card A pointer to a **SCF_Card_t**. If the smartcard is successfully opened, a handle for the card will be returned through this parameter.

terminal
The terminal (from **SCF_Session_getTerminal**(3SMARTCARD)) containing a smartcard to open.

The **SCF_Terminal_getCard( )** function establishes a context with a specific smartcard in a terminal. Card objects can be used to send APDUs (Application Protocol Data Units) to the card with **SCF_Card_exchangeAPDU**(3SMARTCARD). When the card is no longer needed, **SCF_Card_close**(3SMARTCARD) should be called to release allocated resources.

If **SCF_Terminal_getCard( )** is called multiple times in the same session to access the same physical card (while the card remains inserted), the same **SCF_Card_t** will be returned in each call. The library cannot identifty specific cards, so when a card is reinserted it will be represented by a new **SCF_Card_t**. Multithreaded applications must take care to avoid having one thread close a card that is still needed by another thread. This can be accomplished by coordination within the application, or by having each thread open a seperate session to avoid interference.

If a working card is present in the reader, **SCF_STATUS_SUCCESS** is returned and *card* is a valid reference to the card. Otherwise, an error value is returned and card remains unaltered.

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

SCF_STATUS_BADARGS
The *card* argument is a null pointer.

SCF_STATUS_BADHANDLE
The specified terminal has been closed or is invalid.

SCF_STATUS_FAILED
An internal error occured.

SCF_STATUS_NOCARD
No card is present in the terminal.

Example 1: Access a smartcard.

```
SCF_Status_t status;
SCF_Terminal_t myTerminal;
SCF_Card_t myCard;
```

```
/* (...call SCF_Session_getTerminal to open myTerminal...) */

status = SCF_Terminal_getCard(myTerminal, &myCard);
if (status == SCF_STATUS_NOCARD) {
    printf("Please insert your smartcard and try again.\n");
    exit(0);
}
else if (status != SCF_STATUS_SUCCESS) exit(1);

/* (...go on to use the card with SCF_Card_exchangeAPDU()...) */
```

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

libsmartcard(3LIB), **SCF_Card_close**(3SMARTCARD), **SCF_Card_exchangeAPDU**(3SMARTCARD), **SCF_Card_getInfo**(3SMARTCARD), **SCF_Card_lock**(3SMARTCARD), **SCF_Session_getTerminal**(3SMARTCARD), **attributes**(5)

NAME

SCF_Session_getInfo, SCF_Terminal_getInfo, SCF_Card_getInfo – retrieve information about a session, terminal, or card

SYNOPSIS

cc [*flag...*] *file...* **-lsmartcard** [*library*...]
#include <**smartcard/scf.h**>

SCF_Status_t SCF_Session_getInfo(**SCF_Session_t** *session*, **const char** **name*, **void** **value*);

SCF_Status_t SCF_Terminal_getInfo(**SCF_Terminal_t** *terminal*, **const char** **name*, **void** **value*);

SCF_Status_t SCF_Card_getInfo(**SCF_Card_t** *card*, **const char** **name*, **void** **value*);

card An object that was returned from **SCF_Terminal_getCard**(3SMARTCARD).

name The name of a property for which a value is to be returned. The name is case-sensitive.

session
An object that was returned from **SCF_Session_getSession**(3SMARTCARD).

terminal
An object that was returned from **SCF_Session_getTerminal**(3SMARTCARD).

value The value of the property. The actual type of the value depends on what property was being queried.

These functions obtain information about a session, terminal, or card. The information returned represents the current state of the object and can change between calls.

Each call allocates new storage for the returned result. This storage is tracked internally and is deallocated when the object is closed. An application repeatedly asking for information can cause memory bloat until the object is closed. The application can optionally call **SCF_Session_freeInfo**(3SMARTCARD), **SCF_Terminal_freeInfo**(3SMARTCARD), or **SCF_Card_freeInfo**(3SMARTCARD) to cause immediate deallocation of the value. Applications must not use other means such as**free**(3C) to deallocate the memory.

Applications must not access values that have been deallocated. For example, accessing a Card's ATR after the card has been closed results in undefined behavior.

For a session, the valid property names and value types are:

terminalnames (pointer to **char** **)
The list of terminal names that can currently be used in this session. The returned value is an array of **char** *, each element of the list is a pointer to a terminal name. The end of the array is denoted by a null pointer. The first element of the list is the default terminal for the session, which will be used when **SCF_Session_getTerminal( )** is called with a null pointer for the terminal name.

For a terminal, the standard property names and value types are as follows. Some terminal drivers can define additional driver-specific properties.

name (pointer to **char** *)

The name of the terminal. If the default terminal was used (a null pointer was passed to **SCF_Session_getTerminal( )**), the value will contain the actual name of the default terminal. For example, "MyInternalCardReader".

type (pointer to **char** ∗)
: The type of the terminal. For example, "SunISCRI".

devname (pointer to **char** ∗)
: Information about how the device is attached to the system. This can be a UNIX device name (for example, "/dev/scmi2c0") or some other terminal-specific string describing its relation to the system.

For a card, the valid property names and value types are:

type (pointer to **char** ∗)
: The type of the smartcard, as recognized by the framework (For example, "Cyberflex"). If the framework does not recognize the card type, "UnknownCard" is returned.

atr (pointer to **struct SCF_BinaryData_t** ∗)
: The Answer To Reset (ATR) data returned by the card when it was last inserted or reset. The structure member **length** denotes how many bytes are in the ATR. The structure member **data** is a pointer to the actual ATR bytes.

Upon success, **SCF_STATUS_SUCCESS** is returned and *value* will contain the the requested information. Otherwise, an error value is returned and *value* remains unaltered.

These functions will fail if:

SCF_STATUS_BADARGS
: Either *name* or *value* is a null pointer.

SCF_STATUS_BADHANDLE
: The session, terminal, or card has been closed or is invalid.

SCF_STATUS_FAILED
: An internal error occurred.

SCF_STATUS_UNKNOWNPROPERTY
: The property specified by *name* was not found.

Example 1: Simple string information.

```
SCF_Status_t status;
SCF_Terminal_t myTerminal;
const char *myName, *myType;

/* (...call SCF_Session_getTerminal to open myTerminal...) */

status = SCF_Terminal_getInfo(myTerminal, "name", &myName);
if (status != SCF_STATUS_SUCCESS) exit(1);
status = SCF_Terminal_getInfo(myTerminal, "type", &myType);
if (status != SCF_STATUS_SUCCESS) exit(1);
```

printf("The terminal called %s is a %s\n", myName, myType);

Example 2: Display the names of all terminals available in the session.

```
SCF_Status_t status;
SCF_Session_t mySession;
const char **myList;  /* Technically "const char * const *". */
int i;

/* (...call SCF_Session_getSession to open mySession...) */

status = SCF_Session_getInfo(mySession, "terminalnames", &myList);
if (status != SCF_STATUS_SUCCESS) exit(1);

printf("The following terminals are available:\n");
for (i=0; myList[i] != NULL; i++) {
    printf("%d: %s\n", i, myList[i]);
}
```

Example 3: Display the card's ATR.

```
SCF_Status_t status;
SCF_Card_t myCard;
struct SCF_BinaryData_t *myATR;
int i;

/* (...call SCF_Terminal_getCard to open myCard...) */

status = SCF_Card_getInfo(myCard, "atr", &myATR);
if (status != SCF_STATUS_SUCCESS) exit(1);

printf("The card's ATR is: 0x");
for(i=0; i < myATR->length; i++) {
    printf("%02.2x", myATR->data[i]);
}
printf("\n");
```

See **attributes**(5) for descriptions of the following attributes:

tab() allbox; cw(2.750000i)| cw(2.750000i) lw(2.750000i)| lw(2.750000i). ATTRIBUTE TYPEATTRIBUTE VALUE Interface StabilityEvolving MT-LevelMT-Safe

libsmartcard(3LIB), **SCF_Session_freeInfo**(3SMARTCARD), **SCF_Session_getSession**(3SMARTCARD), **SCF_Session_getTerminal**(3SMARTCARD), **SCF_Terminal_getCard**(3SMARTCARD), **attributes**(5)

NAME

SCF_Terminal_addEventListener, SCF_Terminal_updateEventListener, SCF_Terminal_removeEventListener – receive asychronous event notification

SYNOPSIS

cc [*flag...*] *file...* **-lsmartcard** [*library...*]
#include <**smartcard/scf.h**>

SCF_Status_t SCF_Terminal_addEventListener(**SCF_Terminal_t** *terminal*, **SCF_Event_t** *events*, **void(****callback)(SCF_Event_t, SCF_Terminal_t, void *),* **void** **userData*, **SCF_ListenerHandle_t** **listenerHandle*);

SCF_Status_t SCF_Terminal_updateEventListener(**SCF_Terminal_t** *terminal*, **SCF_ListenerHandle_t** *listenerHandle*, **SCF_Event_t** *events*);

SCF_Status_t SCF_Terminal_removeEventListener(**SCF_Terminal_t** *terminal*, **SCF_ListenerHandle_t** *listenerHandle*);

terminal
A terminal (from **SCF_Session_getTerminal**(3SMARTCARD)) to which the event listener should be added or removed.

events Events to deliver to the callback. An event will not be delivered if it is not listed. The caller can register for multiple events by performing a bitwise OR of the desired events. The valid events are:

SCF_EVENT_ALL
All of the events listed below will be delivered.

SCF_EVENT_CARDINSERTED
A smartcard was inserted into the terminal.

SCF_EVENT_CARDREMOVED
A smartcard was removed from the terminal.

SCF_EVENT_CARDPRESENT
Indicates that a card was present in the terminal when the event listener was first added. This event allows event listeners to determine the initial state of the terminal before an insert or remove event occurs. Either this event or the **SCF_EVENT_CARDABSENT** (see below) event will be delivered only once upon adding an event listener and immediately before any other events are delivered. Future card movements will generate **SCF_EVENT_CARDINSERTED** and **SCF_EVENT_CARDREMOVED** events, but not **SCF_EVENT_CARDPRESENT** or **SCF_EVENT_CARDABSENT** events. An event listener can assume that if a **SCF_EVENT_CARDPRESENT** event is delivered, the next card movement event will be a **SCF_EVENT_CARDREMOVED**.

SCF_EVENT_CARDABSENT
Indicates that a card was not present in the terminal when the event listener was first added. This event allows event listeners to determine the initial state of the terminal before an insert or remove event occurs. Either this event or the **SCF_EVENT_CARDPRESENT** event (see above) will be

delivered only once upon adding an event listener and immediately before any other events are delivered. Future card movements will generate **SCF_EVENT_CARDINSERTED** and **SCF_EVENT_CARDREMOVED** events, but not **SCF_EVENT_CARDPRESENT** or **SCF_EVENT_CARDABSENT** events. An event listener can assume that if a **SCF_EVENT_CARDABSENT** event is delivered, the next card movement event will be a **SCF_EVENT_CARDINSERTED**.

SCF_EVENT_CARDRESET
The smartcard currently present has been reset (see **SCF_Card_reset**(3SMARTCARD)).

SCF_EVENT_TERMINALCLOSED
The terminal is in the process of being closed (due to a call to **SCF_Session_close**(3SMARTCARD) or **SCF_Terminal_close**(3SMARTCARD)), so no further events will be delivered. The *terminal* argument provided to the callback will still be valid.

SCF_EVENT_COMMERROR
The connection to the server has been lost. No further events will be delivered.

callback
A function pointer that will be executed when the desired event occurs. The function must take three arguments. The first is a **SCF_Event_t** containing the event that occured. The second argument is an **SCF_Terminal_t** containing the terminal on which the event occured. The third is a **void** * that can be used to provide arbitrary data to the *callback* when it is executed.

userData
A pointer to arbitrary user data. The data is not accessed by the library. The pointer is simply provided to the callback when an event is issued. This argument can safely be set to *NULL* if not needed. The callback must be able to handle this case.

listenerHandle
A unique "key" that is provided by **SCF_Terminal_addEventListener( )** to refer to a specific event listener registration. This allows multiple event listeners to be selectivly updated or removed.

These functions allow an application to receive notification of events on a terminal as they occur. The concept is similar to a signal handler. When an event occurs, a thread in the SCF library will execute the provided *callback* function. Once added, the listener will receive events until it is removed or either the terminal or session is closed.

When the callback function is executed, the callback arguments specify the event that occured and the terminal on which it occurred. Additionally, each callback will receive the *userData* pointer that was provided when the listener was added. The library does not make a copy of the memory pointed to by *userData*, so applications must take care not to

deallocate that memory until it is known that the callback will no longer access it (for example, by removing the event listener). Each invocation of the callback will be for exactly one event. If the library needs to deliver multiple events, they will be dispatched one at a time. Because the callback is executed from a thread, any operations it performs must be thread safe. For each callback registration, the library creates a new thread to deliver events to that callback. The callback is expected to perform minimal work and return quickly.

An application can add multiple callbacks on a terminal. Any event that occurs will be delivered to all listeners that registered for that event type. The same callback can be registered multiple times. Each call to **SCF_Terminal_addEventListener()** will result in a new **SCF_ListenerHandle_t**. The events a callback receives can be changed by calling **SCF_Session_updateEventListener()** with the handle that was returned when the listener was initially added. If the listener is set to receive no events (that is, the events parameter has no bits set), the listener will remain registered but will not receive any events. To remove a listener and release allocated resources, use **SCF_Terminal_removeEventListener()** or close the terminal.

If the event listener was successfully added or removed, **SCF_STATUS_SUCCESS** is returned. Otherwise, an error value is returned and the internal list of registered event listeners remains unaltered.

These functions will fail if:

SCF_STATUS_BADARGS
: The callback function pointer and/or *listenerHandle* is null, or an unknown event was specified.

SCF_STATUS_BADHANDLE
: The specified terminal has been closed or is invalid, or the event listener handle could not be found to update or remove.

SCF_STATUS_COMMERROR
: The connection to the server was lost.

SCF_STATUS_FAILED
: An internal error occurred.

Example 1: Register for card movements.

```
struct myState_t {
    int isStateKnown;
    int isCardPresent;
};

void myCallback(SCF_Event_t event, SCF_Terminal_t eventTerminal,
    void *data) {
    struct myState_t *state = data;
    if (event == SCF_EVENT_CARDINSERTED) {
        printf("--- Card inserted ---\n");
        state->isCardPresent = 1;
    }
    else if (event == SCF_EVENT_CARDREMOVED) {
        printf("--- Card removed ---\n");
        state->isCardPresent = 0;
    }
    state->isStateKnown = 1;
}

main() {
    SCF_Status_t status;
    SCF_Terminal_t myTerminal;
    SCF_ListenerHandle_t myListener;
    struct myState_t myState;

    /* (...call SCF_Session_getTerminal to open myTerminal...) */

    myState.isStateKnown = 0;
    status = SCF_Terminal_addEventListener(myTerminal,
        SCF_EVENT_CARDINSERTED|SCF_EVENT_CARDREMOVED, &myCallback,
        &myState, &myListener);
    if (status != SCF_STATUS_SUCCESS) exit(1);

    while(1) {
        if (!myState.isStateKnown)
            printf("Waiting for first event...\n");
        else {
            if (myState.isCardPresent)
                printf("Card is present.\n");
            else
                printf("Card is not present.\n");
        }
        sleep(1);
    }
}
```

Example 2: Use different callbacks for each event.

```
void myInsertCallback(SCF_Event_t event, SCF_Terminal_t eventTerminal,
    void *data) {

    /* ... */
}

void myRemoveCallback(SCF_Event_t event, SCF_Terminal_t eventTerminal,
    void *data) {
    /* ... */
}
```

```
main () {
    SCF_Status_t status;
    SCF_Terminal_t terminal;
    SCF_ListenerHandle_t myListener1, myListener2, myListener3;
    int foo, bar;

    /* (...call SCF_Session_getTerminal to open myTerminal...) */

    status = SCF_Terminal_addEventListener(myTerminal,
        SCF_EVENT_CARDINSERTED, &myInsertCallback, &foo,
        &myListener1);
    if (status != SCF_STATUS_SUCCESS) exit(1);

    status = SCF_Terminal_addEventListener(myTerminal,
        SCF_EVENT_CARDREMOVED, &myRemoveCallback, &foo,
        &myListener2);
    if (status != SCF_STATUS_SUCCESS) exit(1);

    status = SCF_Terminal_addEventListener(myTerminal,
        SCF_EVENT_CARDREMOVED, &myRemoveCallback, &bar,
        &myListener3);
    if (status != SCF_STATUS_SUCCESS) exit(1);

    /*
     * At this point, when each insertion occurs, myInsertCallback
     * will be called once (with a pointer to foo). When each removal
     * occurs, myRemoveCallback will be called twice. One call will
     * be given a pointer to foo, and the other will be given a
     * pointer to bar.
     */

    status = SCF_Terminal_removeEventListener(myTerminal,
        myListener2);
    if (status != SCF_STATUS_SUCCESS) exit(1);

    /*
     * Now, when a removal occurs, myRemoveCallback will only be
     * called once, with a pointer to bar.
     */

    /* ... */
}
```

Example 3: Use initial state events to show user the terminal state in a GUI.

```
void myCallback(SCF_Event_t event, SCF_Terminal_t eventTerminal,
    void *unused) {
    if (event == SCF_EVENT_CARDPRESENT) {
        /* Set initial icon to a terminal with a card present. */
    }
    else if (event == SCF_EVENT_CARDABSENT) {
        /* Set initial icon to a terminal without a card present. */
    }
    else if (event == SCF_EVENT_CARDINSERTED) {
        /* Show animation for card being inserted into a terminal. */
    }
    else if (event == SCF_EVENT_CARDREMOVED) {
        /* Show animation for card being removed from a terminal. */
    }
}

main() {
    SCF_Terminal_t myTerminal;
    SCF_ListenerHandle_t myListener;

    /* (...call SCF_Session_getTerminal to open myTerminal...) */

    status = SCF_Terminal_addEventListener(myTerminal,
        SCF_EVENT_ALL, &myCallback, NULL, &myListener);
    if (status != SCF_STATUS_SUCCESS) exit(1);


    /* ... */
}
```

See **attributes**(5) for descriptions of the following attributes:

tab() allbox; cw(2.750000i)| cw(2.750000i) lw(2.750000i)| lw(2.750000i). ATTRIBUTE TYPEATTRIBUTE VALUE Interface StabilityEvolving MT-LevelMT-Safe

libsmartcard(3LIB), **SCF_Card_reset**(3SMARTCARD), **SCF_Session_close**(3SMARTCARD), **SCF_Session_getTerminal**(3SMARTCARD), **SCF_Session_updateEventListener**(3SMARTCARD), **SCF_Terminal_close**(3SMARTCARD), **SCF_Terminal_removeEventListener**(3SMARTCARD), **attributes**(5)

NAME

SCF_Terminal_addEventListener, SCF_Terminal_updateEventListener, SCF_Terminal_removeEventListener – receive asychronous event notification

SYNOPSIS

cc [*flag...*] *file...* **-lsmartcard** [*library...*]
#include <**smartcard/scf.h**>

SCF_Status_t SCF_Terminal_addEventListener(**SCF_Terminal_t** *terminal*, **SCF_Event_t** *events*, **void(****callback)(SCF_Event_t, SCF_Terminal_t, void *)*, **void** **userData*, **SCF_ListenerHandle_t** **listenerHandle*);

SCF_Status_t SCF_Terminal_updateEventListener(**SCF_Terminal_t** *terminal*, **SCF_ListenerHandle_t** *listenerHandle*, **SCF_Event_t** *events*);

SCF_Status_t SCF_Terminal_removeEventListener(**SCF_Terminal_t** *terminal*, **SCF_ListenerHandle_t** *listenerHandle*);

terminal
A terminal (from **SCF_Session_getTerminal**(3SMARTCARD)) to which the event listener should be added or removed.

events Events to deliver to the callback. An event will not be delivered if it is not listed. The caller can register for multiple events by performing a bitwise OR of the desired events. The valid events are:

SCF_EVENT_ALL
All of the events listed below will be delivered.

SCF_EVENT_CARDINSERTED
A smartcard was inserted into the terminal.

SCF_EVENT_CARDREMOVED
A smartcard was removed from the terminal.

SCF_EVENT_CARDPRESENT
Indicates that a card was present in the terminal when the event listener was first added. This event allows event listeners to determine the initial state of the terminal before an insert or remove event occurs. Either this event or the **SCF_EVENT_CARDABSENT** (see below) event will be delivered only once upon adding an event listener and immediately before any other events are delivered. Future card movements will generate **SCF_EVENT_CARDINSERTED** and **SCF_EVENT_CARDREMOVED** events, but not **SCF_EVENT_CARDPRESENT** or **SCF_EVENT_CARDABSENT** events. An event listener can assume that if a **SCF_EVENT_CARDPRESENT** event is delivered, the next card movement event will be a **SCF_EVENT_CARDREMOVED**.

SCF_EVENT_CARDABSENT
Indicates that a card was not present in the terminal when the event listener was first added. This event allows event listeners to determine the initial state of the terminal before an insert or remove event occurs. Either this event or the **SCF_EVENT_CARDPRESENT** event (see above) will be

delivered only once upon adding an event listener and immediately before any other events are delivered. Future card movements will generate **SCF_EVENT_CARDINSERTED** and **SCF_EVENT_CARDREMOVED** events, but not **SCF_EVENT_CARDPRESENT** or **SCF_EVENT_CARDABSENT** events. An event listener can assume that if a **SCF_EVENT_CARDABSENT** event is delivered, the next card movement event will be a **SCF_EVENT_CARDINSERTED**.

SCF_EVENT_CARDRESET
The smartcard currently present has been reset (see **SCF_Card_reset**(3SMARTCARD)).

SCF_EVENT_TERMINALCLOSED
The terminal is in the process of being closed (due to a call to **SCF_Session_close**(3SMARTCARD) or **SCF_Terminal_close**(3SMARTCARD)), so no further events will be delivered. The *terminal* argument provided to the callback will still be valid.

SCF_EVENT_COMMERROR
The connection to the server has been lost. No further events will be delivered.

callback
A function pointer that will be executed when the desired event occurs. The function must take three arguments. The first is a **SCF_Event_t** containing the event that occured. The second argument is an **SCF_Terminal_t** containing the terminal on which the event occured. The third is a **void** * that can be used to provide arbitrary data to the *callback* when it is executed.

userData
A pointer to arbitrary user data. The data is not accessed by the library. The pointer is simply provided to the callback when an event is issued. This argument can safely be set to *NULL* if not needed. The callback must be able to handle this case.

listenerHandle
A unique "key" that is provided by **SCF_Terminal_addEventListener( )** to refer to a specific event listener registration. This allows multiple event listeners to be selectivly updated or removed.

These functions allow an application to receive notification of events on a terminal as they occur. The concept is similar to a signal handler. When an event occurs, a thread in the SCF library will execute the provided *callback* function. Once added, the listener will receive events until it is removed or either the terminal or session is closed.

When the callback function is executed, the callback arguments specify the event that occured and the terminal on which it occurred. Additionally, each callback will receive the *userData* pointer that was provided when the listener was added. The library does not make a copy of the memory pointed to by *userData*, so applications must take care not to

deallocate that memory until it is known that the callback will no longer access it (for example, by removing the event listener). Each invocation of the callback will be for exactly one event. If the library needs to deliver multiple events, they will be dispatched one at a time. Because the callback is executed from a thread, any operations it performs must be thread safe. For each callback registration, the library creates a new thread to deliver events to that callback. The callback is expected to perform minimal work and return quickly.

An application can add multiple callbacks on a terminal. Any event that occurs will be delivered to all listeners that registered for that event type. The same callback can be registered multiple times. Each call to **SCF_Terminal_addEventListener( )** will result in a new **SCF_ListenerHandle_t**. The events a callback receives can be changed by calling **SCF_Session_updateEventListener( )** with the handle that was returned when the listener was initially added. If the listener is set to receive no events (that is, the events parameter has no bits set), the listener will remain registered but will not receive any events. To remove a listener and release allocated resources, use **SCF_Terminal_removeEventListener( )** or close the terminal.

If the event listener was successfully added or removed, **SCF_STATUS_SUCCESS** is returned. Otherwise, an error value is returned and the internal list of registered event listeners remains unaltered.

These functions will fail if:

SCF_STATUS_BADARGS
The callback function pointer and/or *listenerHandle* is null, or an unknown event was specified.

SCF_STATUS_BADHANDLE
The specified terminal has been closed or is invalid, or the event listener handle could not be found to update or remove.

SCF_STATUS_COMMERROR
The connection to the server was lost.

SCF_STATUS_FAILED
An internal error occurred.

Example 1: Register for card movements.

```
struct myState_t {
    int isStateKnown;
    int isCardPresent;
};

void myCallback(SCF_Event_t event, SCF_Terminal_t eventTerminal,
    void *data) {
    struct myState_t *state = data;
    if (event == SCF_EVENT_CARDINSERTED) {
        printf("--- Card inserted ---\n");
        state->isCardPresent = 1;
    }
    else if (event == SCF_EVENT_CARDREMOVED) {
        printf("--- Card removed ---\n");
        state->isCardPresent = 0;
    }
    state->isStateKnown = 1;
}

main() {
    SCF_Status_t status;
    SCF_Terminal_t myTerminal;
    SCF_ListenerHandle_t myListener;
    struct myState_t myState;

    /* (...call SCF_Session_getTerminal to open myTerminal...) */

    myState.isStateKnown = 0;
    status = SCF_Terminal_addEventListener(myTerminal,
        SCF_EVENT_CARDINSERTED|SCF_EVENT_CARDREMOVED, &myCallback,
        &myState, &myListener);
    if (status != SCF_STATUS_SUCCESS) exit(1);

    while(1) {
        if (!myState.isStateKnown)
            printf("Waiting for first event...\n");
        else {
            if (myState.isCardPresent)
                printf("Card is present.\n");
            else
                printf("Card is not present.\n");
        }
        sleep(1);
    }
}
```

Example 2: Use different callbacks for each event.

```
void myInsertCallback(SCF_Event_t event, SCF_Terminal_t eventTerminal,
    void *data) {

    /* ... */
}

void myRemoveCallback(SCF_Event_t event, SCF_Terminal_t eventTerminal,
    void *data) {
    /* ... */
}
```

```
main () {
    SCF_Status_t status;
    SCF_Terminal_t terminal;
    SCF_ListenerHandle_t myListener1, myListener2, myListener3;
    int foo, bar;

    /* (...call SCF_Session_getTerminal to open myTerminal...) */

    status = SCF_Terminal_addEventListener(myTerminal,
        SCF_EVENT_CARDINSERTED, &myInsertCallback, &foo,
        &myListener1);
    if (status != SCF_STATUS_SUCCESS) exit(1);

    status = SCF_Terminal_addEventListener(myTerminal,
        SCF_EVENT_CARDREMOVED, &myRemoveCallback, &foo,
        &myListener2);
    if (status != SCF_STATUS_SUCCESS) exit(1);

    status = SCF_Terminal_addEventListener(myTerminal,
        SCF_EVENT_CARDREMOVED, &myRemoveCallback, &bar,
        &myListener3);
    if (status != SCF_STATUS_SUCCESS) exit(1);

    /*
     * At this point, when each insertion occurs, myInsertCallback
     * will be called once (with a pointer to foo). When each removal
     * occurs, myRemoveCallback will be called twice. One call will
     * be given a pointer to foo, and the other will be given a
     * pointer to bar.
     */

    status = SCF_Terminal_removeEventListener(myTerminal,
        myListener2);
    if (status != SCF_STATUS_SUCCESS) exit(1);

    /*
     * Now, when a removal occurs, myRemoveCallback will only be
     * called once, with a pointer to bar.
     */

    /* ... */
}
```

Example 3: Use initial state events to show user the terminal state in a GUI.

```
void myCallback(SCF_Event_t event, SCF_Terminal_t eventTerminal,
    void *unused) {
    if (event == SCF_EVENT_CARDPRESENT) {
        /* Set initial icon to a terminal with a card present. */
    }
    else if (event == SCF_EVENT_CARDABSENT) {
        /* Set initial icon to a terminal without a card present. */
    }
    else if (event == SCF_EVENT_CARDINSERTED) {
        /* Show animation for card being inserted into a terminal. */
    }
    else if (event == SCF_EVENT_CARDREMOVED) {
        /* Show animation for card being removed from a terminal. */
    }
}

main() {
    SCF_Terminal_t myTerminal;
    SCF_ListenerHandle_t myListener;

    /* (...call SCF_Session_getTerminal to open myTerminal...) */

    status = SCF_Terminal_addEventListener(myTerminal,
        SCF_EVENT_ALL, &myCallback, NULL, &myListener);
    if (status != SCF_STATUS_SUCCESS) exit(1);


    /* ... */
}
```

See **attributes**(5) for descriptions of the following attributes:

tab() allbox; cw(2.750000i)| cw(2.750000i) lw(2.750000i)| lw(2.750000i). ATTRIBUTE TYPEATTRIBUTE VALUE Interface StabilityEvolving MT-LevelMT-Safe

libsmartcard(3LIB), **SCF_Card_reset**(3SMARTCARD), **SCF_Session_close**(3SMARTCARD), **SCF_Session_getTerminal**(3SMARTCARD), **SCF_Session_updateEventListener**(3SMARTCARD), **SCF_Terminal_close**(3SMARTCARD), **SCF_Terminal_removeEventListener**(3SMARTCARD), **attributes**(5)

NAME

SCF_Terminal_waitForCardPresent, SCF_Terminal_waitForCardAbsent, SCF_Card_waitForCardRemoved – wait for a card to be inserted or removed

SYNOPSIS

cc [*flag...*] *file...* **-lsmartcard** [*library...*]
#include <**smartcard/scf.h**>

SCF_Status_t SCF_Terminal_waitForCardPresent(**SCF_Terminal_t** *terminal*, **unsigned int** *timeout*);

SCF_Status_t SCF_Terminal_waitForCardAbsent(**SCF_Terminal_t** *terminal*, **unsigned int** *timeout*);

SCF_Status_t SCF_Card_waitForCardRemoved(**SCF_Card_t** *card*, **unsigned int** *timeout*);

card A card that was returned from **SCF_Terminal_getCard**(3SMARTCARD).

terminal
A terminal that was returned from **SCF_Session_getTerminal**(3SMARTCARD).

timeout
The maximum number or seconds to wait for the desired state to be reached. If the timeout is 0, the function will immediately return **SCF_STATUS_TIMEOUT** if the terminal or card is not in the desired state. A timeout of **SCF_TIMEOUT_MAX** can be specified to indicate that the function should never timeout.

These functions determine if a card is currently available in the specified terminal.

The **SCF_Card_waitForCardRemoved( )** function differs from **SCF_Terminal_waitForCardAbsent( )** in that it checks to see if a specific card has been removed. If another card (or even the same card) has since been reinserted, **SCF_Card_waitForCardRemoved( )** will report that the old card was removed, while the **SCF_Terminal_waitForCardAbsent( )** will instead report that there is a card present.

If the desired state is already true, the function will immediately return **SCF_STATUS_SUCCESS**. Otherwise it will wait for a change to the desired state, or for the timeout to expire, whichever occurs first.

Unlike an event listener (**SCF_Terminal_addEventListener**(3SMARTCARD)), these functions return the state of the terminal, not just events. To use an electronics analogy, event listeners are edge-triggered, while these functions are level-triggered.

If the desired state is reached before the timeout expires, **SCF_STATUS_SUCCESS** is returned. If the timeout expires, **SCF_STATUS_TIMEOUT** is returned. Otherwise, an error value is returned.

These functions will fail if:

SCF_STATUS_BADHANDLE
The specified *terminal* or *card* has been closed or is invalid.

SCF_STATUS_COMMERROR
The server closed the connection.

SCF_STATUS_FAILED
An internal error occured.

Example 1: Determine if a card is currently inserted.

```
int isCardCurrentlyPresent(SCF_Terminal_t myTerminal) {
    SCF_Status_t status;

    /*
     * The timeout of zero makes sure this call will always
     * return immediately.
     */
    status = SCF_Terminal_waitForCardPresent(myTerminal, 0);

    if (status == SCF_STATUS_SUCCESS) return (TRUE);
    else if (status == SCF_STATUS_TIMEOUT) return (FALSE);

    /*
     * For other errors, this example just assumes no card
     * is present. We don't really know.
     */
    return (FALSE);
}
```

Example 2: Remind the user every 5 seconds to remove their card.

```
SCF_Status_t status;
SCF_Terminal_t myTerminal;

/* (...call SCF_Session_getTerminal to open myTerminal...) */

status = SCF_Terminal_waitForCardAbsent(myTerminal, 0);
while (status == SCF_STATUS_TIMEOUT) {
    printf("Please remove the card from the terminal!\n");
    status = SCF_Terminal_waitForCardAbsent(myTerminal, 5);
}

if (status == SCF_STATUS_SUCCESS)
    printf("Thank you.\n");
else
    exit(1);

/* ... */
```

Example 3: Demonstrate the difference between the card-specific and terminal-specific calls.

```
SCF_Status_t status;
SCF_Terminal_t myTerminal;
SCF_Card_t myCard;

/* (...call SCF_Session_getTerminal to open myTerminal...) */

status = SCF_Terminal_getCard(myTerminal, &myCard);
if (status != SCF_STATUS_SUCCESS) exit(1);

/*
 * While we sleep, assume user removes the card
 * and inserts another card.
 */
sleep(10);

status = SCF_Terminal_waitForCardAbsent(myTerminal, 0);
/*
 * In this case, status is expected to be SCF_STATUS_TIMEOUT, as there
 * is a card present.
 */

status = SCF_Card_waitForCardRemoved(myCard, 0);
/*
 * In this case, status is expected to be SCF_STATUS_SUCCESS, as the
 * card returned from SCF_Terminal_getCard was indeed removed (even
 * though another card is currently in the terminal).
 */

/* ... */
```

See **attributes**(5) for descriptions of the following attributes:

tab() allbox; cw(2.750000i)| cw(2.750000i) lw(2.750000i)| lw(2.750000i). ATTRIBUTE TYPEATTRIBUTE VALUE Interface StabilityEvolving MT-LevelMT-Safe

libsmartcard(3LIB), **SCF_Session_getTerminal**(3SMARTCARD), **SCF_Terminal_addEventListener**(3SMARTCARD), **SCF_Terminal_getCard**(3SMARTCARD), **attributes**(5)

NAME

SCF_Terminal_waitForCardPresent, SCF_Terminal_waitForCardAbsent, SCF_Card_waitForCardRemoved – wait for a card to be inserted or removed

SYNOPSIS

cc [*flag...*] *file...* **-lsmartcard** [*library*...]
#include <**smartcard/scf.h**>

SCF_Status_t SCF_Terminal_waitForCardPresent(**SCF_Terminal_t** *terminal*, **unsigned int** *timeout*);

SCF_Status_t SCF_Terminal_waitForCardAbsent(**SCF_Terminal_t** *terminal*, **unsigned int** *timeout*);

SCF_Status_t SCF_Card_waitForCardRemoved(**SCF_Card_t** *card*, **unsigned int** *timeout*);

card A card that was returned from **SCF_Terminal_getCard**(3SMARTCARD).

terminal
A terminal that was returned from **SCF_Session_getTerminal**(3SMARTCARD).

timeout
The maximum number or seconds to wait for the desired state to be reached. If the timeout is 0, the function will immediately return **SCF_STATUS_TIMEOUT** if the terminal or card is not in the desired state. A timeout of **SCF_TIMEOUT_MAX** can be specified to indicate that the function should never timeout.

These functions determine if a card is currently available in the specified terminal.

The **SCF_Card_waitForCardRemoved( )** function differs from **SCF_Terminal_waitForCardAbsent( )** in that it checks to see if a specific card has been removed. If another card (or even the same card) has since been reinserted, **SCF_Card_waitForCardRemoved( )** will report that the old card was removed, while the **SCF_Terminal_waitForCardAbsent( )** will instead report that there is a card present.

If the desired state is already true, the function will immediately return **SCF_STATUS_SUCCESS**. Otherwise it will wait for a change to the desired state, or for the timeout to expire, whichever occurs first.

Unlike an event listener (**SCF_Terminal_addEventListener**(3SMARTCARD)), these functions return the state of the terminal, not just events. To use an electronics analogy, event listeners are edge-triggered, while these functions are level-triggered.

If the desired state is reached before the timeout expires, **SCF_STATUS_SUCCESS** is returned. If the timeout expires, **SCF_STATUS_TIMEOUT** is returned. Otherwise, an error value is returned.

These functions will fail if:

SCF_STATUS_BADHANDLE
The specified *terminal* or *card* has been closed or is invalid.

SCF_STATUS_COMMERROR
The server closed the connection.

SCF_STATUS_FAILED
An internal error occured.

Example 1: Determine if a card is currently inserted.

```
int isCardCurrentlyPresent(SCF_Terminal_t myTerminal) {
    SCF_Status_t status;

    /*
     * The timeout of zero makes sure this call will always
     * return immediately.
     */
    status = SCF_Terminal_waitForCardPresent(myTerminal, 0);

    if (status == SCF_STATUS_SUCCESS) return (TRUE);
    else if (status == SCF_STATUS_TIMEOUT) return (FALSE);

    /*
     * For other errors, this example just assumes no card
     * is present. We don’t really know.
     */
    return (FALSE);
}
```

Example 2: Remind the user every 5 seconds to remove their card.

```
SCF_Status_t status;
SCF_Terminal_t myTerminal;

/* (...call SCF_Session_getTerminal to open myTerminal...) */

status = SCF_Terminal_waitForCardAbsent(myTerminal, 0);
while (status == SCF_STATUS_TIMEOUT) {
    printf("Please remove the card from the terminal!\n");
    status = SCF_Terminal_waitForCardAbsent(myTerminal, 5);
}

if (status == SCF_STATUS_SUCCESS)
    printf("Thank you.\n");
else
    exit(1);

/* ... */
```

Example 3: Demonstrate the difference between the card-specific and terminal-specific calls.

```
SCF_Status_t status;
SCF_Terminal_t myTerminal;
SCF_Card_t myCard;

/* (...call SCF_Session_getTerminal to open myTerminal...) */

status = SCF_Terminal_getCard(myTerminal, &myCard);
if (status != SCF_STATUS_SUCCESS) exit(1);

/*
 * While we sleep, assume user removes the card
 * and inserts another card.
 */
sleep(10);

status = SCF_Terminal_waitForCardAbsent(myTerminal, 0);
/*
 * In this case, status is expected to be SCF_STATUS_TIMEOUT, as there
 * is a card present.
 */

status = SCF_Card_waitForCardRemoved(myCard, 0);
/*
 * In this case, status is expected to be SCF_STATUS_SUCCESS, as the
 * card returned from SCF_Terminal_getCard was indeed removed (even
 * though another card is currently in the terminal).
 */

/* ... */
```

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

libsmartcard(3LIB), **SCF_Session_getTerminal**(3SMARTCARD), **SCF_Terminal_addEventListener**(3SMARTCARD), **SCF_Terminal_getCard**(3SMARTCARD), **attributes**(5)

NAME

SCF_strerror – get a string describing a status code

SYNOPSIS

cc [*flag*...] *file*... **-lsmartcard** [*library*...]
#include <**smartcard/scf.h**>

const char ***SCF_strerror**(**SCF_Status_t** *error*);

error A value returned from a smartcard SCF function call. A list of all current codes is contained in <**smartcard/scf.h**>

The **SCF_strerror( )** function provides a mechanism for generating a brief message that describes each **SCF_Status_t** error code. An application might use the message when displaying or logging errors.

The string returned by the function does not contain any newline characters. Returned strings must not be modified or freed by the caller.

A pointer to a valid string is always returned. If the provided *error* is not a valid SCF error code, a string is returned stating that the error code is unknown. A null pointer is never returned.

Example 1: Report a fatal error.

```
SCF_Status_t status;
SCF_Session_t mySession;

status = SCF_Session_getSession(&mySession);
if (status != SCF_STATUS_SUCCESS) {
    printf("Smartcard startup error: %s\n", SCF_strerror(status));
    exit(1);
}
```

/* ... */

Messages returned from **SCF_strerror( )** are in the native language specified by the **LC_MESSAGES** locale category; see **setlocale**(3C). The C locale is used if the native strings could not be loaded.

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

libsmartcard(3LIB), **SCF_Session_getSession**(3SMARTCARD), **strerror**(3C), **attributes**(5)

NAME

clb.conf – Content Load Balancer Configuration File

SYNOPSIS

/etc/opt/SUNWclb/clb.conf

DESCRIPTION

The clb.conf file is a local file that identifies the interface set configured for use by the Content Load Balancer in a system. The file is used by the /etc/rc1.d/K32clbctl script and the /etc/rc2.d/S93clbctl script, which runs at boot time to configure Content Load Balanced interfaces. If changes are made to the clb.conf file, the system must be rebooted for the changes to take effect.

The interfaces that participate in the load balancing are entered one per line. '#' is used for comment lines.

EXAMPLES

The following example shows the clb.conf file for a system with two Content Load Balancer interfaces, <ce0> and the VLAN interface <ce6001>.

```
#
#
Configure
<ce0>
and
<ce6001>
ce0
ce6001
```

SEE ALSO

clbconfig(1M)

NOTES

Each entry should be entered on one line with no breaks or carriage returns.

NAME

envmond.conf - configuration file for environment monitor daemon

SYNOPSIS

/usr/platform/SUNW,UltraSPARC-IIi-Netract/envmond.conf

DESCRIPTION

The **envmond.conf** file is the configuration file for **envmond**(1M), the system environment monitor daemon. The daemon monitors environmental devices to check for conditions that may require some action. The **envmond (1M)** daemon logs appropriate messages to a system log file via **syslogd**(1M).

Each configuration file entry provides the daemon information about a shared object library, referred to as a policy, which has the knowledge to monitor a device. Each policy entry describes an interface between the envmond daemon and the policy. The policy entry in the **envmond.conf** file can contain configurable parameters in the *policy-args* field.

All policy entries have the same format:

poll-interval policy-name policy-args

The three fields shown above are separated by whitespace. Use the backslash (/) at the end of a line to continue *policy-args* to the line following.

The fields in the **envmond.conf** file are described as follows:

poll-interval
: Given in seconds as a decimal number, specifies how often to invoke the policy check function. If *poll-interval* is 0, the policy check function will never be called.

policy-name
: The file name, with optional path, of the file implementing the policy. The default location for the policy files is **/usr/platform/SUNW,UltraSPARC-IIi-Netract/lib/envmond/sparcv9**

policy-args
: An optional list of whitespace-separated arguments to be passed to the policy during initialization. The number and format of these arguments is policy-dependent.

The following sections describe policies shipped with the implementation of **envmond***(1M).*

fancpu Policy
: The fancpu policy polls I2C slave devices every *poll-interval* seconds to get the current CPU temperature and the fantray status. If the CPU temperature reaches a warning temperature threshold, a warning message is printed on the system console and to the system log file specified in **syslog.conf**(4). If the CPU temperature reaches the shutdown temperature, a critical error message is printed on the system console by **syslogd**(1M). The system is then halted by the **shutdown**(1M) command. The fan status will be reflected by the corresponding LEDs on the System Status Board, and with log messages sent to **syslogd.**

powersupply Policy

The powersupply policy sets and clears the power supply LEDs on the System Status Board to reflect power supply status. The policy also handles an interrupt event if a power supply fails.

scsb Policy

The System Controller and Status Board Policy is primarily to configure the scsb driver for cPCI Slot Status LED control. The default **scsb_led_ctrl** setting is false, meaning that the scsb driver controls the cPCI slot LEDs. If **scsb_led_ctrl** is set to true, then some application is responsible for slot LED updates.

EXAMPLES

Example 1: Sample Entries

The first entry, below, invokes the powersupply shared library every 60 seconds. The second entry specifies that the scsb policy controls the cPCI Slot Status LED.

```
60 powersupply.so
scsb.so scsb_led_crtl=false
```

FILES

/usr/platform/SUNW,UltraSPARC-IIi-Netract/
Installation directory.

The following relative pathnames are all beneath the directory named above.

lib/envmond/sparcv9/envmond
Executable for the environmental daemon.

lib/envmond/sparcv9/fancpu.so
Policy for CPU temperature and fan speed control.

lib/envmond/sparcv9/powersupply.so
Policy for power supply monitoring.

SEE ALSO

envmond(1M), **syslogd**(1M), **syslogd.conf**(4)

NAME

av1394 – 1394 audio/video driver

SYNOPSIS

unit@GUID

The **av1394** driver is an *IEEE 1394* compliant target driver that supports the *IEC 61883* Consumer Audio/Video Equipment - Digital Interface standard. The driver is used to receive and transmit isochronous data streams in the common isochronous packet (CIP) format, as well as asynchronous function control protocol (FCP) frames. The driver also supports connection management procedures (CMP).

DEVICE SPECIAL FILES

/dev/av/N/async
Device node for asynchronous data

/dev/av/N/isoch
Device node for isochronous data

FILES

kernel/drv/sparcv9/av1394
64-bit ELF kernel module

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	Limited to PCI-based SPARC systems
Availability	SUNWav1394x

hci1394(7D)

IEEE Std 1394-1995 Standard for a High Performance Serial Bus

IEC 61883 Consumer Audio/Video Equipment - Digital Interface

NAME

bge – SUNW,bge Gigabit Ethernet driver for Broadcom BCM5704

SYNOPSIS

/dev/bge*

DESCRIPTION

The **bge** Gigabit Ethernet driver is a multi-threaded, loadable, clonable, GLD-based STREAMS driver supporting the Data Link Provider Interface, **dlpi**(7P), on Broadcom BCM5703C or BCM5704 Gigabit Ethernet controllers fitted to the system motherboard. These devices incorporate both MAC and PHY functions and provide three-speed (copper) Ethernet operation on the RJ-45 connectors.

The **bge** driver functions include controller initialization, frame transmit and receive, promiscuous and multicast support, and error recovery and reporting.

The **bge** driver and hardware support 'auto-negotiation,' a protocol specified by the 1000Base-T standard. Auto-negotiation allows each device to advertise its capabilities and discover those of its peer (link partner). The highest common denominator supported by both link partners is automatically selected, yielding the greatest available throughput, while requiring no manual configuration. The **bge** driver also allows you to configure the advertised capabilities to less than the maximum (where the full speed of the interface is not required), or to force a specific mode of operation, irrespective of the link partner's advertised capabilities.

APPLICATION PROGRAMMING INTERFACE

The cloning character-special device, **/dev/bge**, is used to access all BCM570x devices fitted to the system motherboard.

The **bge** driver is dependent on **/kernel/misc/gld**, a loadable kernel module that provides the **bge** driver with the DLPI and STREAMS functionality required of a LAN driver. See **gld**(7D) for more details on the primitives supported by the driver.

You must send an explicit DL_ATTACH_REQ message to associate the opened stream with a particular device (PPA). The PPA ID is interpreted as an unsigned integer data type and indicates the corresponding device instance (unit) number. The driver returns an error (DL_ERROR_ACK) if the PPA field value does not correspond to a valid device instance number for the system. The device is initialized on first attach and de-initialized (stopped) at last detach.

The values returned by the driver in the DL_INFO_ACK primitive in response to a DL_INFO_REQ are:

- Maximum SDU is 1500 (ETHERMTU - defined in <**sys/ethernet.h**>).
- Minimum SDU is 0.
- DLSAP address length is 8.
- MAC type is **DL_ETHER**.
- **SAP** length value is *-2*, meaning the physical address component is followed immediately by a 2-byte **SAP** component within the **DLSAP** address.
- Broadcast address value is the Ethernet/IEEE broadcast address (FF:FF:FF:FF:FF:FF).

Once in the DL_ATTACHED state, you must send a DL_BIND_REQ to associate a particular Service Access Point (SAP) with the stream.

CONFIGURATION

By default, the **bge** driver performs auto-negotiation to select the link speed and mode. Link speed and mode can be any one of the following, (as described in the *IEEE803.2* standards):

- 1000 Mbps, full-duplex
- 1000 Mbps, half-duplex
- 100 Mbps, full-duplex
- 100 Mbps, half-duplex
- 10 Mbps, full-duplex
- 10 Mbps, half-duplex

The auto-negotiation protocol automatically selects:

- Speed (1000 Mbps, 100 Mbps, or 10 Mbps)
- Operation mode (full-duplex or half-duplex)

as the highest common denominator supported by both link partners. Because the **bge** device supports all modes, the effect is to select the highest throughput mode supported by the other device.

Alternatively, you can set the capabilities advertised by the **bge** device using **ndd**(1M). The driver supports a number of parameters whose names begin with *adv_* (see below). Each of these parameters contains a boolean value that determines whether the device advertises that mode of operation. In addition, the *adv_autoneg_cap* parameter controls whether autonegotiation is performed. If *adv_autoneg_cap* is set to 0, the driver forces the mode of operation selected by the first non-zero parameter in priority order as listed below:

```
                    (highest priority/greatest throughput)
       adv_1000fdx_cap         1000Mbps full duplex
       adv_1000hdx_cap         1000Mpbs half duplex
       adv_100fdx_cap          100Mpbs full duplex
       adv_100hdx_cap          100Mpbs half duplex
       adv_10fdx_cap           10Mpbs full duplex
       adv_10hdx_cap           10Mpbs half duplex
```
(lowest priority/least throughput)

For example, to prevent the device 'bge2' from advertising gigabit capabilities, enter (as super-user):

```
# ndd -set /dev/bge2 adv_1000hdx_cap 0
# ndd -set /dev/bge2 adv_1000fdx_cap 0
```

All capabilities default to enabled. Note that changing any capability parameter will cause the link to go down while the link partners renegotiate the link speed/duplex using the newly changed capabilities.

The current settings of the parameters may be found using **ndd -get**. In addition, the driver exports the current state, speed, and duplex setting of the link via **ndd** parameters (these are read only and may not be changed). For example, to check link state of device **bge0**:

```
# ndd -get /dev/bge0 link_status
1
# ndd -get /dev/bge0 link_speed
100
# ndd -get /dev/bge0 link_duplex
1
```

The output above indicates that the link is up and running at 100Mbps full-duplex.

FILES

/dev/bge*
Character special device

/kernel/drv/sparcv9/bge
bge driver binary

/*platform*/*platform-name*/kernel/drv/bge.conf
bge configuration file

ATTRIBUTES

See **attributes**(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	SPARC

SEE ALSO

attributes(5), **gld**(7D), **streamio**(7I), **dlpi**(7P)

Writing Device Drivers

STREAMS Programming Guide

Network Interfaces Programmer's Guide

NAME

ehci – Enhanced host controller driver

SYNOPSIS

usb@unit-address

The **ehci** driver is a USBA (Solaris USB Architecture) compliant nexus driver that supports the Enhanced Host Controller Interface Specification 1.0, an industry standard developed by Intel.

The **ehci** driver supports control, bulk and interrupt transfers. It enables support for USB 2.0 devices in the USBA 1.0 framework

FILES

/kernel/drv/usba10_ehci
32-bit ELF kernel module for the USBA 1.0 framework∗

/kernel/drv/sparcv9/usba10_ehci
64-bit ELF kernel module for the USBA 1.0 framework∗

∗ Please see *www.sun.com/desktop/whitepapers.html* for more information regarding USB dual framework implementation, *USBA 1.0*, and *USB 2.0*.

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based SPARC systems
Availability	SUNWusb, SUNWusbx

attributes(5), **hubd**(7D), **ohci**(7D), **usba**(7D)

Writing Device Drivers

Universal Serial Bus Specification 2.0

Enhanced Host Controller Interface Specification 1.0

System Administration Guide: Basic Administration

http://www.sun.com/desktop/whitepapers.html

http://www.sun.com/io

All host controller errors are passed to the client drivers. Root errors are documented in **hubd**(7D).

In addition to being logged, the following messages may appear on the system console. All messages are formatted in the following manner:

WARNING: <device path> (usba10_ehci><instance number>):
Error message...

Connecting a high speed isochronous device to a high speed
port is not supported." 6 High speed isochronous transfers are not supported.

Unrecoverable USB hardware error.

There was an unrecoverable USB hardware error reported by the **ehci** controller. Reboot the system. If this problem persists, contact your system vendor.

No SOF interrupts.

The USB hardware is not generating Start Of Frame interrupts. Reboot the system. If this problem persists, contact your system vendor.

NAME

ge – GEM Gigabit-Ethernet device driver

SYNOPSIS

/dev/ge

DESCRIPTION

The **ge Sun** Gigabit-Ethernet driver is a multi-threaded, loadable, clonable, STREAMS hardware driver supporting the connectionless Data Link Provider Interface, **dlpi**(7P), over **GEM SBus and PCI** Gigabit-Ethernet add-in Adapters. Multiple **GEM based** adapters installed within the system are supported by the driver. The **ge** driver provides basic support for the **GEM based Ethernet** hardware and it is used to handle the **SUNW,sbus-gem (SBus GEM) and pci108e,2bad** (PCI GEM) devices. Functions include chip initialization, frame transmit and receive, multicast and promiscuous support, and error recovery and reporting. The **GEM** device provides 1000BASE-SX networking interfaces using the **GEM ASIC,** external SERDES and Fiber optical Transceiver. The GEM ASIC provides the appropriate bus interface, MAC functions and the Physical code sub-layer (PCS) functions. The external SERDES connects to a fiber transceiver and provides the physical connection.

The 1000Base-SX standard specifies an "auto-negotiation" protocol to automatically select the mode of operation. In addition to to the duplex mode of the operation, the GEM ASIC can auto-negotiate for IEEE 802.3x Frame Based Flow Control capabilities. The GEM PCS is capable of doing "auto-negotiation" with the remote-end of the link (Link Partner) and receives the capabilities of the remote end. It selects the **Highest Common Denominator** mode of operation based on the priorities. It also supports **forced-mode** of operation where the driver can select the mode of operation.

APPLICATION PROGRAMMING INTERFACE

The cloning character-special device **/dev/ge** is used to access all **ge** controllers installed within the system.

ge and DLPI

The **ge** driver is a "style 2" Data Link Service provider. All **M_PROTO** and **M_PCPROTO** type messages are interpreted as **DLPI** primitives. Valid **DLPI** primitives are defined in <**sys/dlpi.h**>**.** Refer to **dlpi**(7P) for more information. An explicit **DL_ATTACH_REQ** message by the user is required to associate the opened stream with a particular device (**ppa**). The **ppa** ID is interpreted as an **unsigned long** data type and indicates the corresponding device instance (unit) number. An error (**DL_ERROR_ACK**) is returned by the driver if the **ppa** field value does not correspond to a valid device instance number for this system. The device is initialized on first attach and de-initialized (stopped) at last detach.

The values returned by the driver in the **DL_INFO_ACK** primitive in response to the **DL_INFO_REQ** from the user are as follows:

- The maximum SDU is **1500** (**ETHERMTU -** defined in <sys/ethernet.h>).
- The minimum SDU is **0**.
- The **dlsap** address length is **8.**
- The MAC type is **DL_ETHER**.
- The **sap** length values is –**2** meaning the physical address component is followed immediately by a 2 byte **sap** component within the DLSAP address.

- The service mode is **DL_CLDLS**.
- No optional quality of service (QOS) support is included at present so the QOS fields are **0**.
- The provider style is **DL_STYLE2**.
- The version is **DL_VERSION_2**.
- The broadcast address value is Ethernet/IEEE broadcast address (**0xFFFFFF**).

Once in the **DL_ATTACHED** state, the user must send a **DL_BIND_REQ** to associate a particular SAP (Service Access Pointer) with the stream. The **ge** driver interprets the **sap** field within the **DL_BIND_REQ** as an Ethernet "type" therefore valid values for the **sap** field are in the [**0-0xFFFF**] range. Only one Ethernet type can be bound to the stream at any time.

If the user selects a **sap** with a value of **0**, the receiver will be in "802.3 mode". All frames received from the media having a "type" field in the range [**0-1500**] are assumed to be 802.3 frames and are routed up all open Streams which are bound to **sap** value **0**. If more than one Stream is in "802.3 mode" then the frame will be duplicated and routed up multiple Streams as **DL_UNITDATA_IND** messages.

In transmission, the driver checks the **sap** field of the **DL_BIND_REQ** if the **sap** value is **0**, and if the destination type field is in the range [**0-1500**]. If either is true, the driver computes the length of the message, not including initial **M_PROTO** mblk (message block), of all subsequent **DL_UNITDATA_REQ** messages and transmits 802.3 frames that have this value in the MAC frame header length field.

The **ge** driver **DLSAP** address format consists of the 6 byte physical (Ethernet) address component followed immediately by the 2 byte **sap** (type) component producing an 8 byte **DLSAP** address. Applications should *not* hard code to this particular implementation-specific **DLSAP** address format but use information returned in the **DL_INFO_ACK** primitive to compose and decompose **DLSAP** addresses. The **sap** length, full **DLSAP** length, and **sap**/physical ordering are included within the **DL_INFO_ACK**. The physical address length can be computed by subtracting the **sap** length from the full **DLSAP** address length or by issuing the **DL_PHYS_ADDR_REQ** to obtain the current physical address associated with the stream.

Once in the **DL_BOUND** state, the user may transmit frames on the Ethernet by sending **DL_UNITDATA_REQ** messages to the **ge** driver. The **ge** driver will route received Ethernet frames up all those open and bound streams having a **sap** which matches the Ethernet type as **DL_UNITDATA_IND** messages. Received Ethernet frames are duplicated and routed up multiple open streams if necessary. The **DLSAP** address contained within the **DL_UNITDATA_REQ** and **DL_UNITDATA_IND** messages consists of both the **sap** (type) and physical (Ethernet) components.

In addition to the mandatory connectionless **DLPI** message set the driver additionally supports the following primitives.

ge Primitives

The **DL_ENABMULTI_REQ** and **DL_DISABMULTI_REQ** primitives enable/disable reception of individual multicast group addresses. A set of multicast addresses may be iteratively created and modified on a per-stream basis using these primitives. These primitives are accepted by the driver in any state following **DL_ATTACHED**.

The **DL_PROMISCON_REQ** and **DL_PROMISCOFF_REQ** primitives with the **DL_PROMISC_PHYS** flag set in the **dl_level** field enables/disables reception of all ("promiscuous mode") frames on the media including frames generated by the local host.
When used with the **DL_PROMISC_SAP** flag set this enables/disables reception of all **sap** (Ethernet type) values. When used with the **DL_PROMISC_MULTI** flag set this enables/disables reception of all multicast group addresses. The effect of each is always on a per-stream basis and independent of the other **sap** and physical level configurations on this stream or other streams.

The **DL_PHYS_ADDR_REQ** primitive returns the 6 octet Ethernet address currently associated (attached) to the stream in the **DL_PHYS_ADDR_ACK** primitive. This primitive is valid only in states following a successful **DL_ATTACH_REQ**.

The **DL_SET_PHYS_ADDR_REQ** primitive changes the 6 octet Ethernet address currently associated (attached) to this stream. The credentials of the process which originally opened this stream must be superuser. Otherwise **EPERM** is returned in the **DL_ERROR_ACK**. This primitive is destructive in that it affects all other current and future streams attached to this device. An **M_ERROR** is sent up all other streams attached to this device when this primitive is successful on this stream. Once changed, all streams subsequently opened and attached to this device will obtain this new physical address. Once changed, the physical address will remain until this primitive is used to change the physical address again or the system is rebooted, whichever comes first.

ge DRIVER

By default, the ge driver performs "auto-negotiation" to select the **mode** and **flow control capabilities** of the link.

The link can be in one of the *4* following modes:
- 1000 Mbps, full-duplex
- 1000 Mbps, half-duplex
- Symmetric Pause
- Asymmetric Pause

These speeds and modes are described in the 1000Base-TX standard.

The *auto–negotiation* protocol automatically selects:
- Operation mode (half-duplex or full-duplex)
- Flow Control Capability (Symmetric and/or Asymmetric)

The auto–negotiation protocol does the following:
- Gets all the modes of operation supported by the Link Partner
- Advertises its capabilities to the Link Partner
- Selects the highest common denominator mode of operation based on the priorities

The *GEM Hardware* is capable of all of the operating modes listed above, when by *default*, auto-negotiation is used to bring up the link and select the common mode of operation with the Link Partner. The PCS also supports *forced-mode* of operation in which the driver can select the mode of operation and the flow control capabilities, using the *ndd* utility.

The **GEM** device also supports programmable **“IPG”** (Inter-Packet Gap) parameters **ipg1** and **ipg2**. By default, the driver sets **ipg1** to 8 **byte-times** and **ipg2** to 4 **byte-times** (which are the standard values). Sometimes, the user may want to alter these values from the standard 1000 Mpbs **IPG** set to 0.096 microseconds.

ge Parameter List

The ge driver provides for setting and getting various parameters for the **GEM** device. The parameter list includes **current transceiver status**, **current link status**, **inter-packet gap**, **PCS capabilities** and **link partner capabilities**.

The PCS has two set of capabilities: one set reflects the capabilities of the **hardware**, which are **read-only (RO)** parameters and the second set reflects the values chosen by the user and is used in **speed selection**. There are **read/write (RW)** capabilities. At boot time, these two sets of capabilities will be the same. The Link Partner capabilities are also read only parameters because the current default value of these parameters can only be read and cannot be modified.

FILES

/dev/ge	**ge** special character device.
/kernel/drv/ge.conf	System wide default device driver properties

SEE ALSO

ndd(1M), **netstat**(1M), **driver.conf**(4), **dlpi**(7P), **ie**(7D), **le**(7D) **hme**(7D) **qfe**(7D)

NAME

grbeep – Platform-dependent beep driver for SMBus-based hardware

SYNOPSIS

beep@unit-address

DESCRIPTION

The **grbeep** driver generates beeps on platforms (including Sun Blade 100, 150, 1500, 2500) that use SMBbus-based registers and USB keyboards. When the **KIOCCMD** ioctl is issued to the USB keyboard module (see **usbkbm**(7M)) with command **KBD_CMD_BELL/KBD_CMD_NOBELL**, **usbkbm**(7M) passes the request to the **grbeep** driver to turn the beep on and off, respectively.

FILES

/platform/sun4u/kernel/drv/sparcv9/grbeep
64-bit ELF kernel driver

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	SMBus-based SPARC
Availability	SUNWcarx.u

SEE ALSO

kbd(1), **attributes**(5), **bbc_beep**(7D), **kb**(7M), **usbkbm**(7M)

Writing Device Drivers

DIAGNOSTICS

None

NAME

hid – Human interface device (HID) class driver

SYNOPSIS

keyboard@unit-address
mouse@unit-address
input@unit-address:consumer_control

The **hid** driver is a USBA (Solaris USB Architecture) compliant client driver that supports the *Human Interface Device Class (HID) 1.0* specification. The Human Interface Device (HID) class encompasses devices controlled by humans to operate computer systems. Typical examples of HID devices include keyboards, mice, trackballs, and joysticks. HID also covers front-panel controls such as knobs, switches, and buttons. A USB device with multiple interfaces may have one interface for audio and a HID interface to define the buttons that control the audio.

The **hid** driver is general and primarily handles the USB functionality of the device and generic HID functionality. For example, HID interfaces are required to have an interrupt pipe for the device to send data packets, and the **hid** driver opens the pipe to the interrupt endpoint and starts polling. The **hid** driver is also responsible for managing the device through the default control pipe. In addition to being a USB client driver, the **hid** driver is also a STREAMS driver so that modules may be pushed on top of it.

The HID specification is flexible, and HID devices dynamically describe their packets and other parameters through a HID report descriptor. The HID parser is a misc module that parses the HID report descriptor and creates a database of information about the device. The **hid** driver queries the HID parser to find out the type and characteristics of the HID device. The HID specification predefines packet formats for the boot protocol keyboard and mouse.

/kernel/drv/hid
32 bit ELF kernel hid module for original USBA framework∗

/kernel/drv/sparcv9/hid
64 bit ELF kernel hid module for original USBA framework∗

/kernel/drv/usba10_hid
32 bit ELF kernel hid module for USBA 1.0 framework∗

/kernel/drv/sparcv9/usba10_hid
64 bit ELF kernel hid module for USBA 1.0 framework∗

/kernel/drv/usba10_hid.conf
usba10_hid configuration file

/kernel/misc/hidparser
32 bit ELF kernel hidparser module for the original USBA framework∗

/kernel/misc/sparcv9/hidparser
64 bit ELF kernel hidparser module for the original USBA framework∗

/kernel/misc/usba10_hidparser
32 bit ELF kernel hidparser module for the USBA 1.0 framework∗

/kernel/misc/sparcv9/usba10_hidparser
64 bit ELF kernel hidparser module for the USBA 1.0 framework*

* Please see *www.sun.com/desktop/whitepapers.html* for more information regarding USB dual framework implementation, *USBA 1.0, and USB 2.0.*

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	Original USBA drivers and files: PCI-based systems USBA 1.0 drivers and files: PCI-based SPARC systems
Availability	SUNWusb
	SUNWusbx

cfgadm_usb(1M), **attributes**(5), **usba**(7D)

Writing Device Drivers

STREAMS Programming Guide

Universal Serial Bus Specification 2.0

Device Class Definition for Human Interface Devices (HID) 1.1

System Administration Guide: Basic Administration

http://www.sun.com/desktop/whitepapers.html

http://www.sun.com/io

None.

NOTES The hid driver currently supports only keyboard, mouse and audio HID control device.

NAME

hsi – S-Bus based high speed serial line interface.

SYNOPSIS

#include <fcntl.h>
open(/dev/hih*n*, mode);
open(/dev/hih, mode);

DESCRIPTION

The **hsi** module is a loadable and unloadable STREAMS driver that implements the sending and receiving of data packets such as HDLC frames over synchronous serial lines. The **hsi** driver is a standalone driver that supports HSI/S S-Bus based serial interface hardware and provides physical level data transfer services for upper data link layer protocols (e.g. HDLC or SDLC).

The **hih***n* devices provide what is known as a **data path** which supports the transfer of data via **read**(2) and **write**(2) system calls, as well as **ioctl**(2) calls. Data path opens are exclusive in order to protect against injection or diversion of data by another process.

The **hih** device provides a separate **control path** for use by programs that need to configure or monitor a connection independent of any exclusive access restrictions imposed by data path opens. Up to three control paths may be active on a particular serial channel at any one time. Control path accesses are restricted to **ioctl**(2) calls only; no data transfer is possible.

When used in synchronous modes, the Z16C35 ISCC supports several options for **clock sourcing** and data encoding. Both the transmit and receive clock sources can be set to be the external receive clock (RTxC) and the internal baud rate generator (BRG). Additionally, the transmit clock source can be set to the external transmit clock (TRxC).

The **baud rate generator** is a programmable divisor that derives a clock frequency from the PCLK input signal to the ISCC. A programmed baud rate is translated into a 16-bit **time constant** that is stored in the ISCC. When using the BRG as a clock source the driver may answer a query of its current speed with a value different from the one specified. This is because baud rates translate into time constants in discrete steps, and reverse translation shows the change. If an exact baud rate is required that cannot be obtained with the BRG, an external clock source must be selected.

A **local loopback mode** is available, primarily for use by the **hsi_loop(1M)** utility for testing purposes, and should not be confused with **SDLC loop mode,** which is not supported on this interface. This option should be selected casually, or left in use when not needed.

The **hsi** driver keeps running totals of various hardware generated events for each channel. These include numbers of packets and characters sent and received, abort conditions detected by the receiver, receive CRC errors, transmit underruns, receive overruns, input errors and output errors, and message block allocation failures. Input errors are logged whenever an incoming message must be discarded, such as when an abort or CRC error is detected, a receive overrun occurs, or when no message block is available to store incoming data. Output errors are logged when the data must be discarded due to underruns, CTS drops during transmission, CTS timeouts, or excessive watchdog timeouts caused by a cable break.

IOCTLS

The **hsi** driver supports several **ioctl()** commands, including:

S_IOCGETMODE	Return a **struct scc_mode** containing parameters currently in use. These include the transmit and receive clock sources, boolean loop-back and NRZI mode flags and the integer baudrate.
S_IOCSETMODE	The argument is a **struct scc_mode** from which the ISCC channel will be programmed.
S_IOCGETSTATS	Return a **struct hs_stats** containing the current totals of hardware-generated events. These include numbers of packets and characters sent and received by the driver, aborts and CRC errors detected, transmit underruns, and receive overruns.
S_IOCCLRSTATS	Clear the hardware statistics for this channel.
S_IOCGETSPEED	Returns the currently set baudrate as an integer. This may not reflect the actual data transfer rate if external clocks are used.
S_IOCGETMCTL	Returns the current state of the CTS and DCD incoming modem interface signals as an integer.

The following structures are used with **hsi** ioctl() commands:

```
struct scc_mode {
      charsm_txclock; /* transmit clock sources */
      charsm_rxclock; /* receive clock sources */
      charsm_iflags;  /* data and clock inversion flags (non-zsh) */
      u_char        sm_config;  /* boolean configuration options */
      int  sm_baudrate; /* real baud rate */
      int  sm_retval; /* reason codes for ioctl failures */
};

struct hs_stats {
      unsigned int ipack; /* input packets */
      unsigned int opack; /* output packets */
      unsigned int ichar; /* input bytes */
      unsigned int ochar; /* output bytes */
      int  abort;    /* abort received */
      int  crc;      /* CRC error */
      int  cts;      /* CTS timeouts */
      int  dcd;      /* Carrier drops */
      int  overrun; /* receive overrun */
      int  underrun; /* transmit underrun */
      int  ierror;   /* input error */
      int  oerror;   /* output error */
      int  nobuffers; /* rcv side memory allocation failure */
    int ishort;       /* input packet too short ( < CRC-bytes+1) */
    int ilong;        /* input packet too long (> mru) */
    int inactive;     /* input packet rcvd when rcv is inactive */
    int idma;         /* receive dma error */
    int olong;      /* output packet too long (> mtu) */
    int ohung;      /* transmit hung (usually missing clock) */
    int odma;       /* transmit dma error */
};
```

ERRORS

An **open()** will fail if a STREAMS message block cannot be allocated, or:

ENXIO	The unit being opened does not exist.
EBUSY	The device is in use by another serial protocol.

An **ioctl()** will fail if:

EINVAL	An attempt was made to select an invalid clocking source.
EINVAL	The baud rate specified for use with the baud rate generator would translate to a null time constant in the ISCC's registers.

FILES

/dev/hih[0-n], /dev/hih
Character-special devices.

/usr/include/sys/ser_sync.h
Header file specifying synchronous serial communication definitions.

SEE ALSO

hsi_init(1M), hsi_loop(1M), hsi_stat(1M), hsi_trace(1M)

Refer to the *Zilog Z16C35 ISCC Serial Communications Controller Technical Manual* for details of the ISCC's operation and capabilities.

DIAGNOSTICS

hih data open failed, no memory, rq=*nnn*
hih clone open failed, no memory, rq=*nnn* A kernel memory allocation failed for one of the private data structures. The value of *nnn* is the address of the read queue passed to open(2).

hih_open: can't alloc message block
The open could not proceed because an initial STREAMS message block could not be made available for incoming data.

hih: clone device *d* **must be attached before use!**
An operation was attempted through a control path before that path had been attached to a particular serial channel.

hih*n***: invalid operation for clone dev.**
An inappropriate STREAMS message type was passed through a control path. Only M_IOCTL and M_PROTO message types are permitted.

hih*n***: not initialized, can't send message**
An M_DATA message was passed to the driver for a channel that had not been programmed at least once since the driver was loaded. The ISCC's registers were in an unknown state. The S_IOCSETMODE ioctl command performs the programming operation.

hih*n***: transmit hung**
The transmitter was not successfully restarted after the watchdog timer

expired.

hihN: Bad PPA = N.
SunHSI/S driver received a DL_ATTACH_REQ, which has an out-of-range PPA number N, from upper layers.

hihN: port N not installed.
The SunHSI/S port N, which is referenced by the PPA number in a received DL_ATTACH_REQ message, is not installed in the system.

hihN: out of STREAMS mblocks.
Running out of streams mblocks for SunHSI/S port N.

hihN: xmit hung.
Transmission hung on SunHSI/S port N. This usually happens because of cabling problems or due to missing clocks from the CSU/DSU or modem.

hihN: <hih_rxsoft> no buffers - rxbad.
Running out of streams mblocks for SunHSI/S port N in hih_rxsoft() routine.

WARNING: hih_init: changed baudrate from 100000 to 99512.
The baud rate specified was rounded to a value the SunHSI/S hardware can support.

NAME

hsip – PCI-Bus based high speed serial line interface.

SYNOPSIS

#include <**fcntl.h**>
#include <**/usr/include/sys/ser_sync.h**>
open(/dev/hihp*n*, mode);
open(/dev/hihp, mode);

DESCRIPTION

The **hsip** module is a loadable and unloadable STREAMS driver that implements the sending and receiving of data packets such as HDLC frames over synchronous serial lines. The **hsip** driver is a standalone driver that supports HSI / P PCI-Bus based serial interface hardware and provides phsipcal level data transfer services for upper data link layer protocols (e.g. HDLC or SDLC).

The **hihp***n* devices provide what is known as a **data path** which supports the transfer of data via **read**(2) and **write**(2) system calls, as well as **ioctl**(2) calls. Data path opens are exclusive in order to protect against injection or diversion of data by another process.

The **hihp** device provides a separate **control path** for use by programs that need to configure or monitor a connection independent of any exclusive access restrictions imposed by data path opens. Up to three control paths may be active on a particular serial channel at any one time. Control path accesses are restricted to **ioctl**(2) calls only; no data transfer is possible.

The HSIP ports support several options for **clock sourcing** and data encoding. Both the transmit and receive clock sources can be set to be the external transmit clock (TxC), external receive clock (RxC), the internal baud rate generator (BRG), or the output of the SCC's Digital Phase-Lock Loop (DPLL).

The **baud rate generator** is a programmable divisor that derives a clock frequency from the PCLK input signal to the SCC. A programmed baud rate is translated into a 16-bit **time constant** that is stored in the SCC. When using the BRG as a clock source the driver may answer a query of its current speed with a value different from the one specified. This is because baud rates translate into time constants in discrete steps, and reverse translation shows the change. If an exact baud rate is required that cannot be obtained with the BRG, an external clock source must be selected.

Use of the DPLL option requires the selection of NRZI data encoding and the setting of a non-zero value for the baud rate, because the DPLL uses the BRG as its reference clock source.

A **local loopback mode** is available, primarily for use by the **hsip_loop(1m)** utility for testing purposes, and should not be confused with **SDLC loop mode,** which is not supported on this interface. Also, an **auto-echo** feature may be selected that causes all incoming data to be routed to the transmit data line, allowing the port to act as the remote end of a digital loop. Neither of these options should be selected casually, or left in use when not needed.

The **hsip** driver keeps running totals of various hardware generated events for each channel. These include numbers of packets and characters sent and received, abort conditions detected by the receiver, receive CRC errors, transmit underruns, receive overruns, input errors and output errors. Input errors are logged whenever an incoming message must be discarded, such as when an abort or CRC error is detected, a receive overrun occurs, or when no message block is available to store incoming data. Output errors are logged when the data must be discarded due to underruns, CTS drops during transmission, CTS timeouts, or excessive watchdog timeouts caused by a cable break.

IOCTLS

The **hsip** driver supports several **ioctl()** commands, including:

S_IOCGETMODE	Return a **struct scc_mode** containing parameters currently in use. These include the transmit and receive clock sources, boolean loopback and NRZI mode flags and the integer baudrate.
S_IOCSETMODE	The argument is a **struct scc_mode** from which the SCC channel will be programmed.
S_IOCGETSTATS	Return a **struct sl_stats** containing the current totals of hardware-generated events. These include numbers of packets and characters sent and received by the driver, aborts and CRC errors detected, transmit underruns, and receive overruns.
S_IOCCLRSTATS	Clear the hardware statistics for this channel.
S_IOCGETSPEED	Returns the currently set baudrate as an integer. This may not reflect the actual data transfer rate if external clocks are used.
S_IOCGETMCTL	Returns the current state of the CTS and DCD incoming modem interface signals as an integer.

The following structures are used with **hsip** ioctl() commands:

```
struct scc_mode {
        char    sm_txclock;     /* transmit clock sources */
        char    sm_rxclock;     /* receive clock sources */
        char    sm_iflags;      /* data and clock inversion flags (non-zsh) */
        u_char  sm_config;      /* boolean configuration options */
        int     sm_baudrate;    /* real baud rate */
        int     sm_retval;      /* reason codes for ioctl failures */
};

struct sl_stats {
        int     ipack;          /* input packets */
        int     opack;          /* output packets */
        int     ichar;          /* input bytes */
        int     ochar;          /* output bytes */
        int     abort;          /* abort received */
        int     crc;            /* CRC error */
        int     cts;            /* CTS timeouts */
        int     dcd;            /* Carrier drops */
        int     overrun;        /* receive overrun */
        int     underrun;       /* transmit underrun */
        int     ierror;         /* input error */
        int     oerror;         /* output error */
        int     nobuffers;      /* receive side memory allocation failure */
};
```

ERRORS

An **open()** will fail if a STREAMS message block cannot be allocated, or:

ENXIO	The unit being opened does not exist.
EBUSY	The device is in use by another serial protocol.

An **ioctl()** will fail if:

EINVAL	An attempt was made to select an invalid clocking source.
EINVAL	The baud rate specified for use with the baud rate generator would translate to a null time constant in the SCC's registers.

FILES

/dev/hihp[0-n], /dev/hihp
Character-special devices.

/usr/include/sys/ser_sync.h
Header file specifying synchronous serial communication definitions.

SEE ALSO

hsip_init(1M), hsip_loop(1M), hsip_stat(1M),

Refer to the *Motorola MC68360 Quad Integrated Communications Controller Technical Manual* for details of the SCC's operation and capabilities.

DIAGNOSTICS

hihp data open failed, no memory, rq=*nnn*
hihp clone open failed, no memory, rq=*nnn* A kernel memory allocation failed for one of the private data structures. The value of *nnn* is the address of the read queue passed to open(2).

hihp_open: can't alloc message block
The open could not proceed because an initial STREAMS message block could not be made available for incoming data.

hihp: clone device *d* must be attached before use!
An operation was attempted through a control path before that path had been attached to a particular serial channel.

hihp*n*: invalid operation for clone dev.
An inappropriate STREAMS message type was passed through a control path. Only M_IOCTL and M_PROTO message types are permitted.

hihp*n*: not initialized, can't send message
An M_DATA message was passed to the driver for a channel that had not been programmed at least once since the driver was loaded. The S_IOCSETMODE ioctl command performs the programming operation.

hihp*n***: transmit hung**
The transmitter was not successfully restarted after the watchdog timer expired.

NAME

hubd – USB hub driver

SYNOPSIS

hub@unit-address

The **hubd** is a USBA (Solaris USB Architecture) compliant client driver that supports USB hubs conforming to the *Universal Serial Bus Specification 2.0*.∗ The **hubd** driver supports bus-powered and self-powered hubs. The driver supports hubs with individual port power, ganged power and no power switching.

When a device is attached to a hub port, the **hubd** driver enumerates the device by determining its type and assigning an address to it. For multi-configuration devices, **hubd** sets the preferred configuration (refer to **cfgadm_usb**(1M) to select a configuration). The **hubd** driver attaches a driver to the device if one is available for the default or selected configuration. When the device is disconnected from the hub port, the **hubd** driver offlines any driver instance attached to the device.

∗Hubd for the original USBA framework supports *USB 1.0* and *1.1* hubs only. Hubd for the *USBA 1.0* framework supports *USB 2.0* hubs as well. Please see *www.sun.com/desktop/whitepapers.html* for more information regarding *USBA 1.0*, USB dual framework, and *USB 2.0*.

/kernel/drv/hubd
32 bit ELF kernel module for original USBA framework∗

/kernel/drv/sparcv9/hubd
64 bit ELF kernel module for original USBA framework∗

/kernel/drv/usba10_hubd
32 bit ELF kernel module for USBA 1.0 framework∗

/kernel/drv/sparcv9/usba10_hubd
64 bit ELF kernel module for USBA 1.0 framework∗

/kernel/drv/usba10_hubd.conf
usba10_hubd configuration file

∗ Please see *www.sun.com/desktop/whitepapers.html* for more information regarding USB dual framework implementation, *USBA 1.0*, and *USB 2.0*.

See **attributes**(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE

Architecture	Original USBA drivers and files: PCI-based systems USBA 1.0 drivers and files: PCI-based SPARC systems
Availability	SUNWusb, SUNWusbx

cfgadm_usb(1M), **attributes**(5), **usba**(7D)

Writing Device Drivers

Universal Serial Bus Specification 2.0

System Administration Guide: Basic Administration

http://www.sun.com/desktop/whitepapers.html

http://*www.sun.com/io*

In addition to being logged, the following messages may also appear on the system console. All messages are formatted in the following manner:

WARNING: <device path> (usb<instance number>): Error message...

where <**instance number**> is the instance number of **hubd** and <**device path**> is the physical **path to the device in /devices** directory. Messages from the root hub are displayed with a **usb**<**instance number**> prefix instead of **hub**<**instance number**> as the root hub is an integrated part of the host controller.

Connecting a low/full speed device to a high speed external
hub is not supported." 6 (USBA 1.0 only) The USB software does not currently support low or full speed (USB 1.x) devices connected to an external high speed hub which is, in turn, connected to a high speed (USB 2.0) port. Do one of the following to fix:

(1) connect the high speed external hub to a full speed port

(2) connect the low/full speed devices to a full speed external hub

(3) connect the low/full speed devices directly to any USB controller port.

Connecting device on port <*number*>
failed." 6 The driver failed to enumerate the device connected on port <*number*> of hub. If enumeration fails, disconnect and re-connect.

Global over current condition. Please disconnect hub.
The driver detected an over current condition. This means that the aggregate current being drawn by the devices on the downstream port exceeds a preset value. Refer to section 7.2.1.2 and 11.13 of the *Universal Serial Bus Specification 2.0*. You must remove and insert this hub to render it and its down stream devices

functional again. If this message continues to display for a particular hub, you may need to remove downstream devices to eliminate the problem.

Cannot access device. Please reconnect <device name>.
: This hub has been disconnected because a device other than the original one has been inserted. The driver informs you of this fact by displaying the name of the original device.

Devices not identical to the previous one on this port.
: Please disconnect and reconnect." 6 Same condition as described above; however in this case, the driver is unable to identify the original device with a name string.

Local power has been lost, please disconnect hub.
: A USB self-powered hub has lost external power. All USB devices connected down-stream from this hub will cease to function. Disconnect the hub, plug in the external power-supply and then plug in the hub again.

Hub driver supports max of <*n*>
: ports on hub. Hence, using the first <*number of physical ports*> of <*n*> ports available." 6 The current hub driver supports hubs that have <*n*> ports or less. A hub with *more than* <*n*> ports has been plugged in. Only the first <*n*> out of the total <*number of* physical ports> ports are usable.

NAME

nf – FDDI device driver

SYNOPSIS

#**include** <**sys/nf.h**>
#**include** <**sys/dlpi.h**>

DESCRIPTION

nf is a multi-threaded, loadable, clonable, STREAMS hardware device driver supporting the connectionless Data Link Provider Interface, **dlpi**(7), over DP83265A (BSI-2) FDDI controller in the SBus card. There is no fixed limitation on the number of FDDI cards supported by the driver. The **nf** driver provides basic support for the BSI-2, BMAC and PLAYER+ hardware. Functions include chip initialization, frame transmit and receive, multicast and promiscuous support, and error recovery and reporting.

The cloning character-special device **/dev/nf** is used to access BSI-2 controller installed within the system.

nf and DLPI

The **nf** driver is a "style 2" Data Link Service provider. All M_PROTO and M_PCPROTO type msgs are interpreted as DLPI primitives. An explicit DL_ATTACH_REQ message by the user is required to associate the opened stream with a particular device (**ppa**). The **ppa** ID is interpreted as an **unsigned long** and indicates the corresponding device instance (unit) number. An error (DL_ERROR_ACK) is returned by the driver if the **ppa** field value does not correspond to a valid device instance number for this system. The device is initialized on first attach and de-initialized (stopped) on last detach.

The values returned by the driver in the DL_INFO_ACK primitive in response to the DL_INFO_REQ from the user are as follows:

- The max SDU is 4352 (FDDIMTU).
- The min SDU is 0.
- The **dlsap** address length is 8.
- The MAC type is DL_FDDI.
- The **sap** length value is –2 meaning the physical address component is followed immediately by a 2 byte **sap** component within the DLSAP address.
- The service mode is DL_CLDLS.
- No optional quality of service (QOS) support is included at present so the QOS fields are 0.
- The provider style is DL_STYLE2.
- The version is DL_VERSION_2.
- The broadcast address value is Ethernet/IEEE broadcast address (0xFFFFFF).

Once in the DL_ATTACHED state, the user must send a DL_BIND_REQ to associate a particular SAP (Service Access Pointer) with the stream. The **nf** driver interprets the **sap** field within the DL_BIND_REQ as an Ethernet “type” therefore valid values for the **sap** field are in the [0-0xFFFF] range. Only one Ethernet type can be bound to the stream at any time.

In addition to Ethernet V2 service, an “802.3 mode” is provided by the driver and works as follows. **sap** value 0 is treated as equivalent and represent a desire by the user for “802.3 mode”. If the value of the **sap** field of the DL_BIND_REQ is 0, then the driver computes the length of the message, not including initial M_PROTO mblk, of all subsequent DL_UNITDATA_REQ messages and transmits 802.3 frames having this value in the MAC frame header length field and a value of 0xaaaa030000 in the snap header. All frames received from the media having a “type” field in the range [0-1500] are assumed to be 802.3 frames and are routed up all open streams which are bound to **sap** value 0. If more than one stream is in “802.3 mode” then the frame will be duplicated and routed up multiple streams as DL_UNITDATA_IND messages.

The **nf** driver DLSAP address format consists of the 6 byte physical (FDDI) address component followed immediately by the 2 byte **sap** (type) component producing an 8 byte DLSAP address. Applications should *not* hardcode to this particular implementation-specific DLSAP address format but use information returned in the DL_INFO_ACK primitive to compose and decompose DLSAP addresses. The **sap** length, full DLSAP length, and **sap**/physical ordering are included within the DL_INFO_ACK. The physical address length can be computed by subtracting the **sap** length from the full DLSAP address length or by issuing the DL_PHYS_ADDR_REQ to obtain the current physical address associated with the stream.

Once in the DL_BOUND state, the user may transmit frames on the FDDI ring by sending DL_UNITDATA_REQ messages to the **nf** driver. The **nf** driver will route received FDDI frames up all those open and bound streams having a **sap** which matches the type as DL_UNITDATA_IND messages. Received FDDI frames are duplicated and routed up multiple open streams if necessary. The DLSAP address contained within the DL_UNITDATA_REQ and DL_UNITDATA_IND messages consists of both the **sap** (type) and physical (FDDI) components.

nf Primitives

In addition to the mandatory connectionless DLPI message set the driver additionally supports the following primitives.

The DL_ENABMULTI_REQ and DL_DISABMULTI_REQ primitives enable/disable reception of individual multicast group addresses. A set of multicast addresses may be iteratively created and modified on a per-stream basis using these primitives. These primitives are accepted by the driver in any state following DL_ATTACHED.

The DL_PROMISCON_REQ and DL_PROMISCOFF_REQ primitives with the DL_PROMISC_PHYS flag set in the dl_level field enables/disables reception of all (“promiscuous mode”) frames on the media including frames generated by the local host. When used with the DL_PROMISC_SAP flag set this enables/disables reception of all **sap** (Ethernet type) values. When used with the DL_PROMISC_MULTI flag set this enables/disables reception of all multicast group addresses. The effect of each is always on a per-stream basis and independent of the other **sap** and physical level configurations on this stream or other streams.

The DL_PHYS_ADDR_REQ primitive return the 6 octet MAC address currently associated (attached) to the stream in the DL_PHYS_ADDR_ACK primitive. This primitive is valid only in states following a successful DL_ATTACH_REQ.

The DL_SET_PHYS_ADDR_REQ primitive changes the 6 octet MAC address currently associated (attached) to this stream. The credentials of the process which originally opened this stream must be superuser or EPERM is returned in the DL_ERROR_ACK. This primitive is destructive in that it affects all other current and future streams attached to this device. An M_ERROR is sent up all other streams attached to this device when this primitive on this stream is successful. Once changed, all streams subsequently opened and attached to this device will obtain this new physical address. Once changed, the physical address will remain so until this primitive is used to change the physical address again or the system is rebooted, whichever comes first.

By default the first interface will use the systems MAC address but subsequent interfaces will use the FDDI local address.

FILES **/dev/nf**

SEE ALSO **smt**(7), **dlpi**(7),

NAME ohci – OpenHCI host controller driver

SYNOPSIS **usb@unit-address**

The **ohci** driver is a USBA (Solaris USB Architecture) compliant nexus driver that supports the *Open Host Controller Interface Specification 1.0a*, an industry standard developed by Compaq, Microsoft, and National Semiconductor.

The **ohci** driver supports bulk, interrupt, control and isochronous transfers.

/kernel/drv/ohci
32 bit ELF kernel module for original USBA framework∗

/kernel/drv/sparcv9/ohci
64 bit ELF kernel module for original USBA framework∗

/kernel/drv/usba10_ohci
32 bit ELF kernel module for USBA 1.0 framework∗

/kernel/drv/sparcv9/usba10_ohci
64 bit ELF kernel module for USBA 1.0 framework∗

∗ Please see *www.sun.com/desktop/whitepapers.html* for more information regarding USB dual framework implementation, *USBA 1.0*, and *USB 2.0*.

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	Original USBA drivers and files: PCI-based systems USBA 1.0 drivers and files: PCI-based SPARC systems
Availability	SUNWusb, SUNWusbx

attributes(5), **ehci**(7D), **hubd**(7D), **usba**(7D)

Writing Device Drivers

Universal Serial Bus Specification 2.0

Open Host Controller Interface Specification for USB 1.0a

System Administration Guide: Basic Administration

http://www.sun.com/desktop/whitepapers.html *http://www.sun.com/io*

All host controller errors are passed to the client drivers. Root hub errors are documented in **hubd**(7D).

In addition to being logged, the following messages may appear on the system console. All messages are formatted in the following manner:

WARNING: <device path> ohci<instance number>>: Error message...

or

WARNING: <device path> usba10_ohci<instance number>>:
Error message...

Unrecoverable USB Hardware Error.
: There was an unrecoverable USB hardware error reported by the OHCI Controller. Please reboot the system. If this problem persists, contact your system vendor.

No SOF interrupts.
: The USB hardware is not generating **Start Of Frame** interrupts. Please reboot the system. If this problem persists, contact your system vendor.

NAME

pf – FDDI device driver

SYNOPSIS

#include <**sys/pf.h**> **#include** <**sys/dlpi.h**>

DESCRIPTION

pf is a multi-threaded, loadable, clonable, STREAMS hardware device driver supporting the connectionless Data Link Provider Interface, **dlpi**(7), over PBS FDDI controller in the PCI card. The driver also provides support for Applications to get statistics and status of Station Management. There is no fixed limitation on the number of FDDI cards supported by the driver. The **pf** driver provides basic support for the PBS, BMAC and PLAYER+ hardware. Functions include chip initialization, LLC/SMT frame transmit and receive, multicast and promiscuous support, and error recovery and reporting.

The cloning character-special device **/dev/pf** is used to access PBS controller installed within the system.

pf and DLPI

The **pf** driver is a "style 2" Data Link Service provider. All M_PROTO and M_PCPROTO type msgs are interpreted as DLPI primitives. An explicit DL_ATTACH_REQ message by the user is required to associate the opened stream with a particular device (**ppa**). The **ppa** ID is interpreted as an **unsigned long** and indicates the corresponding device instance (unit) number. An error (DL_ERROR_ACK) is returned by the driver if the **ppa** field value does not correspond to a valid device instance number for this system. The device is initialized on first attach and de-initialized (stopped) on last detach.

The values returned by the driver in the DL_INFO_ACK primitive in response to the DL_INFO_REQ from the user are as follows:

- The max SDU is 4352 (FDDIMTU).
- The min SDU is 0.
- The **dlsap** address length is 8.
- The MAC type is DL_FDDI.
- The **sap** length value is –2 meaning the physical address component is followed immediately by a 2 byte **sap** component within the DLSAP address.
- The service mode is DL_CLDLS.
- No optional quality of service (QOS) support is included at present so the QOS fields are 0.
- The provider style is DL_STYLE2.
- The version is DL_VERSION_2.
- The broadcast address value is Ethernet/IEEE broadcast address (0xFFFFFF).

Once in the DL_ATTACHED state, the user must send a DL_BIND_REQ to associate a particular SAP (Service Access Pointer) with the stream. The **pf** driver interprets the **sap** field within the DL_BIND_REQ as an Ethernet "type" therefore valid values for the **sap** field are in the [0-0xFFFF] range. Only one Ethernet type can be bound to the stream at any time.

In addition to Ethernet V2 service, an "802.3 mode" is provided by the driver and works as follows. **sap** value 0 is treated as equivalent and represent a desire by the user for "802.3 mode". If the value of the **sap** field of the DL_BIND_REQ is 0, then the driver computes the length of the message, not including initial M_PROTO mblk, of all subsequent DL_UNITDATA_REQ messages and transmits 802.3 frames having this value in the MAC frame header length field and a value of 0xaaaa030000 in the snap header. All frames received from the media having a "type" field in the range [0-1500] are assumed to be 802.3 frames and are routed up all open streams which are bound to **sap** value 0. If more than one stream is in "802.3 mode" then the frame will be duplicated and routed up multiple streams as DL_UNITDATA_IND messages.

The **pf** driver DLSAP address format consists of the 6 byte physical (FDDI) address component followed immediately by the 2 byte **sap** (type) component producing an 8 byte DLSAP address. Applications should *not* hardcode to this particular implementation-specific DLSAP address format but use information returned in the DL_INFO_ACK primitive to compose and decompose DLSAP addresses. The **sap** length, full DLSAP length, and **sap**/physical ordering are included within the DL_INFO_ACK. The physical address length can be computed by subtracting the **sap** length from the full DLSAP address length or by issuing the DL_PHYS_ADDR_REQ to obtain the current physical address associated with the stream.

Once in the DL_BOUND state, the user may transmit frames on the FDDI ring by sending DL_UNITDATA_REQ messages to the **pf** driver. The **pf** driver will route received FDDI frames up all those open and bound streams having a **sap** which matches the type as DL_UNITDATA_IND messages. Received FDDI frames are duplicated and routed up multiple open streams if necessary. The DLSAP address contained within the DL_UNITDATA_REQ and DL_UNITDATA_IND messages consists of both the **sap** (type) and physical (FDDI) components.

pf Primitives

In addition to the mandatory connectionless DLPI message set the driver additionally supports the following primitives.

The DL_ENABMULTI_REQ and DL_DISABMULTI_REQ primitives enable/disable reception of individual multicast group addresses. A set of multicast addresses may be iteratively created and modified on a per-stream basis using these primitives. These primitives are accepted by the driver in any state following DL_ATTACHED.

The DL_PROMISCON_REQ and DL_PROMISCOFF_REQ primitives with the DL_PROMISC_PHYS flag set in the dl_level field enables/disables reception of all ("promiscuous mode") frames on the media including frames generated by the local host. When used with the DL_PROMISC_SAP flag set this enables/disables reception of all **sap** (Ethernet type) values. When used with the DL_PROMISC_MULTI flag set this enables/disables reception of all multicast group addresses. The effect of each is always on a per-stream basis and independent of the other **sap** and physical level configurations on this stream or other streams.

The DL_PHYS_ADDR_REQ primitive return the 6 octet MAC address currently associated (attached) to the stream in the DL_PHYS_ADDR_ACK primitive. This primitive is valid only in states following a successful DL_ATTACH_REQ.

The DL_SET_PHYS_ADDR_REQ primitive changes the 6 octet MAC address currently associated (attached) to this stream. The credentials of the process which originally opened this stream must be superuser or EPERM is returned in the DL_ERROR_ACK. This primitive is destructive in that it affects all other current and future streams attached to this device. An M_ERROR is sent up all other streams attached to this device when this primitive on this stream is successful. Once changed, all streams subsequently opened and attached to this device will obtain this new physical address. Once changed, the physical address will remain so until this primitive is used to change the physical address again or the system is rebooted, whichever comes first.

By default the first interface will use the systems MAC address but subsequent interfaces will use the FDDI local address.

pf and SMT

The driver provides information on its PHYs and some FORMAC error counters.

The user has to include these two lines in the program before the line '#include <pfsmt.h>'

```
#define     SMT7_2      0
#define     CFG_YES     1
```

The cloning character special device **/dev/pf** is used to access the driver. An explicit DL_ATTACH_REQ message by the user is required to associate the opened stream with a particular device(ppa) where ppa corresponds to the interface instance number.

Once in the DL_ATTACHED state, the user need not send a DL_BIND_REQ. The user can interact with the driver with **ioctl(2)** calls. The arguments for the ioctl are

```
        ioctl (int fd, int request, SMTCB *smtp)
```

The request is **smt** driver specific and can be SMT_GET or SMT_ACT. SMTCB is defined as follows in the header file pfsmt.h

```
typedef struct {
        int         command;
        int         sub_command;
        int         param1;
        int         param2;
        int         param3;
        char        *where;
        int         length;
} SMTCB;
```

SMT_GET:

SMT_GET provides a variety of functions such as to read the HPC registers and to get the smt status. command field of smtp should be initialized to one of the following values

HPC_BMAC1_REGS	: To read the BMAC registers
HPC_READ	: To read the HPC registers
HPC_PORT1_REGS	: To read RMT port1
and HPC_PORT2_REGS	and port2 registers

Some of the commands provide sub commands. The field sub_command should be initialzed to these sub commands.

1. HPC_BMAC1_REGS

HPC_BMAC1_REGS enables the user to read the BMAC registers. HPC_BMAC1_REGS provides two sub commands GET_COUNTER_GROUP and GET_NEIGHBOR_ADDR. GET_COUNTER_GROUP is used to get various SMT counter values.

GET_COUNTER_GROUP needs the SMTCB *smtp to be initialized as follows

```
COUNTER_GROUP  ct;

smtp->command = HPC_BMAC1_REGS;
smtp->sub_command = GET_COUNTER_GROUP;
smtp->where = (char *) &ct;
smtp->length = sizeof (ct);
```

GET_NEIGHBOR_ADDR enables the user to get the MAC address of the Neighbour station. GET_NEIGHBOR_ADDR needs the SMTCB *smtp to be initialized as follows

```
char          addr_buf[12];

smtp->command = HPC_BMAC1_REGS;
smtp->sub_command = GET_NEIGHBOR_ADDR;
smtp->where = addr_buf;
smtp->length = 12;
```

2. HPC_READ

HPC_READ enables the user to read the HPC registers. HPC_READ does not provide any sub commands. HPC_READ needs the SMTCB *smtp to be initialized as follows

```
smtp->command = HPC_READ;
smtp->param1 = HPC_READ | HPC_SIZE_BYTE
                      | <HPC_reg_offset>;
smtp->where = (char *) smtp;
```

where HPC_register_offset offset is set of register space provided by the HPC. For the set of reister offsets refer to the file pfsmt.h

3. HPC_PORT1_REGS and HPC_PORT2_REGS

HPC_PORT1_REGS enables the user to get the status of the Connection Management. HPC_PORT2_REGS is for the second port if the interface is a DAS. The sub command for HPC_PORT1_REGS is GET_PORT_GROUP. HPC_PORT1_REGS needs the SMTCB *smtp to be initialized as follows

```
FDDI_PORT_GROUP port; smtp->command = HPC_PORT1_REGS;
smtp->sub_command = GET_PORT_GROUP;
smtp->where = (char *) &port;
smtp->length = sizeof (port);
```

The two important status returned in the structure port are port.ecm_state and port.pcm_state. port.ecm_state corresponds to the current state of the ECM state machine. The valid values are OUT, IN, TRACE, PATHTEST, INSERT, CHECK and DEINSERT. The value returned in port.ecm_state is the index into the list of the ECM States. port.pcm_state corresponds to the current state of the PCM state machine. The Valid values are OFF, BREAK, TRACE, CONNECT, NEXT, SIGNAL, JOIN, VERIFY, ACTIVE, MAINT. The value returned in port.pcm_state in an index into the list of PCM States.

SMT_ACT:

SMT_ACT is supported to set the state of the smt driver. The command field should always be set to SMT_CTL. SMT_ACT provides two sub commands SMT_ACCEPT_FRAME and SMT_CLOSE. SMT_ACCEPT_FRAME needs to be used when any SMT API client is active.

```
smtp->command = SMT_CTL;
smtp->sub_command = SMT_ACCEPT_FRAME;
```

SMT_CLOSE needs to be used when the API client exits.

```
smtp->command = SMT_CTL;
smtp->sub_command = SMT_CLOSE;
```

To transmit SMT NSA frames the user should bind to FDDI_NSA sap. To transmit other SMT frames the user may bind to FDDI_SMTINFO sap.

FILES

/dev/pf

SEE ALSO

dlpi(7)

NAME

scsa2usb – SCSI to USB bridge driver

SYNOPSIS

storage@unit-address

The **scsa2usb** driver is a USBA (Solaris USB architecture) compliant nexus driver that sup*ports the USB Mass Storage Bulk Only Transport Specification 1.0* and *USB* Control/Bulk/Interrupt (CBI) Transport Specification 1.0. The **scsa2usb** driver also supports USB storage devices that implement CBI Transport without the interrupt completion for status (that is, Control/Bulk (CB) devices.) It supports bus-powered and self-powered USB mass storage devices. This nexus driver is both a USB client driver and a SCSA HBA driver. As such, the **scsa2usb** driver only supports disk devices that utilize the above two transports.

The **scsa2usb** nexus driver maps **SCSA** target driver requests to **USBA** client driver requests.

The **scsa2usb** driver creates a child device info node for each logical unit (LUN) on the mass storage device. The standard Solaris **SCSI** disk driver is attached to those nodes. Refer to **sd**(7D).

This driver supports multiple LUN devices and creates a separate child device info node for each LUN. All child LUN nodes attach to **sd**(7D).

All USB mass storage devices are treated as removable media devices. Thus, a USB mass storage device can be formatted by **rmformat**(1) and managed by Volume Manager. With or without Volume Manager, you can mount, eject, hot remove and hot insert a USB mass storage device, as the following sections explain.

Some devices may be supported by the USB mass storage driver even though they do not identify themselves as compliant with the USB mass storage class.

The **scsa2usb.conf** file contains an **attribute-override-list** that lists the vendor ID, product ID, and revision for matching mass storage devices, as well as fields for overriding the default device attributes. The entries in this list are commented out by default and may be uncommented to enable support of particular devices.

Follow the information given in the **scsa2usb.conf** file to see if a particular device can be supported using the override information. Also see http:*//www.sun.com/io*.

Using Volume Management

Mass storage devices are managed by Volume Manager. **vold**(1M) creates a device nickname which can be listed with **eject**(1). The device is mounted using **volrmmount**(1) under /rmdisk/*label*.

See **volrmmount**(1M) to unmount the device and **eject**(1) to eject the media. If the device is ejected while it is mounted, **vold**(1M) unmounts the device before ejecting it. It also kills any active applications that are accessing the device.

Hot removing a mass storage device with **vold**(1M) active will fail with a console warning. **To hot remove or insert a USB storage device, first stop vold**(1M) by issuing the command **/etc/init.d/volmgt stop**. After the device has been removed or inserted, restart **vold**(1M) by issuing the command **/etc/init.d/volmgt start**.

You can also permanently disable **vold** for removable devices by commenting out the **rmscsi** *line in* **vold.conf**. See the *System Administration Guide*, *Volume I* and *Solaris Common Desktop* Environment: User's Guide for details on how to manage a removable device with CDE and Removable Media Manager. See **dtfile.1X** under CDE for information on how to use Removable Media Manager.

Using mount(1M) and **umount**(1M)

Use **mount**(1M) to mount the device and **umount**(1M) to unmount the device. Use **eject**(1) to eject the media. No **vold** nicknames can be used. (**vold.1m** is disabled.)

Removing the storage device while it is being accessed or mounted will fail with a console warning. To hot remove the storage device from the system, unmount the file system, then kill all applications accessing the device. Next, hot remove the device. A storage device can be hot inserted at any time.

For a comprehensive listing of (non-bootable) USB mass-storage devices that are compatible with this driver, see *http://www.sun.com/io*.

DEVICE SPECIAL

Block special file names are located in **/dev/dsk**; raw file names are located in **/dev/rdsk**. Input/output requests to the devices must follow the same restrictions as those for **SCSI** disks. Refer to **sd**(7D).

IOCTLS

Refer to **dkio**(7I) and **cdio**(7I).

ERRORS

Refer to **sd**(7D).

The device special files for the USB mass storage device are created like those for a **SCSI** disk. Refer to **sd**(7D).

/dev/dsk/c*n***t***n***d***n***s***n*
Block files

/dev/rdsk/c*n***t***n***d***n***s***n*
Raw files

/vol/dev/aliases/zip0
Symbolic link to the character device for the media in Zip drive 0

/vol/dev/aliases/jaz0
Symbolic link to the character device for the media in Jaz drive 0.

/vol/dev/aliases/rmdisk0
Symbolic link to the character device for the media in removable drive 0. This is a generic removable media device.

/kernel/drv/scsa2usb
32-bit ELF kernel module for original USBA framework*

/kernel/drv/sparcv9/scsa2usb
64-bit ELF kernel module for original USBA framework*

/kernel/drv/scsa2usb.conf
Configuration file; can be used to override specific characteristics for **scsa2usb** module

/kernel/drv/usba10_scsa2usb
32-bit ELF kernel module for USBA 1.0 framework∗

/kernel/drv/sparcv9/usba10_scsa2usb
64-bit ELF kernel module for USBA 1.0 framework∗

/kernel/drv/usba10_scsa2usb.conf
Configuration file; can be used to override specific characteristics for **usba10_scsa2usb** module

∗ Please see *www.sun.com/desktop/whitepapers.html* for more information regarding USB dual framework implementation, *USBA 1.0*, and *USB 2.0*.

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	Original USBA drivers and files: PCI-based systems USBA 1.0 drivers and files: PCI-based SPARC systems
Availability	SUNWusb, SUNWusbx

cdrw(1), **eject**(1), **rmformat**(1), **volrmmount**(1), **cfgadm_scsi**(1M), **cfgadm_usb**(1M), **fdisk**(1M), **mount**(1M), **umount**(1M), **vold**(1M), **scsi**(4), **attributes**(5), **usba**(7D), **usb_sd**(7D), **pcfs**(7FS), **cdio**(7I), **dkio**(7I)

Writing Device Drivers

System Administration Guide, Volume I

Solaris Common Desktop Environment: User's Guide

Universal Serial Bus Specification 2.0

Universal Serial Bus Mass Storage Class Specification Overview 1.0

Universal Serial Bus Mass Storage Class Bulk-Only Transport Specification 1.0

Universal Serial Bus Mass Storage Class Control/Bulk/Interrupt (CBI) Transport Specification 1.0

System Administration Guide: Basic Administration

http://www.sun.com/desktop/whitepapers.html

http://www.sun.com/io

Refer to **sd**(7D).

In addition to being logged, the following messages may appear on the system console. All messages are formatted in the following manner:

Warning: <device path> scsa2usb<instance number>:
Error Message...

or

Warning: <device path> usba10_scsa2usb<instance number>:
Error Message...

Cannot access device. Please reconnect <*name*>.
: There was an error in accessing the mass-storage device during reconnect. Please reconnect the device.

Device reported incorrect luns (adjusting to 1) *or* device reported <value> luns (adjusting to 1).
: The mass-storage device reported that it supports an invalid number of LUNs. The driver has adjusted the number of LUNs supported to 1.

Device is busy and cannot be suspended. Please close
: files, unmount and eject." 6 The system wide suspend failed because the mass-storage device is busy. Close the device, unmount the file system and eject the media before retrying the suspend.

Device is not identical to the previous one on this port.
: Please disconnect and reconnect." 6 Another USB device has been inserted on a port that was connected to a mass-storage device. Please disconnect the USB device and reconnect the mass-storage device back into that port.

Disconnected device was busy, please reconnect.
: Disconnection of the mass-storage device failed because the device is busy. Please reconnect the device.

Reinserted device is accessible again.
: The mass-storage device that was hot-removed from its USB slot has been re-inserted to the same slot and is available for access.

Syncing not supported.
: While a system is panicking, a file system is mounted on the mass-storage media. Syncing is not supported by the **scsa2usb** driver.

NOTES

The Zip 100 drive does not comply with *Universal Serial Bus Specification 1.0* and cannot be power managed. Power Management support for Zip 100 has been disabled.

If the system panics while a UFS file system is mounted on the mass storage media, no syncing will take place for the mass-storage device. (Syncing is not supported by the **scsa2usb** driver.) As a result, the file system on the media will not be consistent on reboot.

If a PCFS file system is mounted, no syncing is needed and the filesystem will be consistent on reboot.

If a mass-storage device is busy, system suspend cannot proceed and the system will immediately resume again.

Attempts to remove a mass-storage device from the system will fail. The failure will be logged to the console. An attempt to replace the removed device with some other USB device will also fail. To successfully remove a USB mass-storage device you must "close" all references to it.

An Iomega Zip 100Mb disk cannot be formatted on an Iomega Zip250 drive. See the Iomega web site at *http://www.iomega.com* for details.

Concurrent I/O to devices with multiple LUNs on the same device is not supported.

Some USB CD-RW devices may perform inadequately at their advertised speeds. To compensate, use USB CD-RW devices at lower speeds (2X versus 4X). See **cdrw(1)** for details.

This driver also supports CBI devices that do not use USB interrupt pipe for status completion.

NAME

smt – FDDI SMT Apps Interface device driver

SYNOPSIS

#include <sys/nfsmt.h>

DESCRIPTION

smt is a multi-threaded, loadable, clonable, STREAMS device driver supporting Data Link Provider Interface, **dlpi**(7), for Application programs to get the statistics and status of the Station Management. smt driver provides packet throughput statistics, reconfiguration events and interface exceptions. It also provides the information on its PHYs and some FORMAC error counters.

The user has to include these two lines in the program before the line '#include <nfsmt.h>'

```
#define     SMT7_2      0
#define     CFG_YES     1
```

The cloning character special device **/dev/smt** is used to access the driver. An explicit DL_ATTACH_REQ message by the user is required to associate the opened stream with a particular device(ppa) where ppa corresponds to the interface instance number.

Once in the DL_ATTACHED state, the user need not send a DL_BIND_REQ. The user can interact with the driver with **ioctl(2)** calls. The arguments for the ioctl are

```
        ioctl (int fd, int request, SMTCB *smtp)
```

The request is **smt** driver specific and can be SMT_GET or SMT_ACT. SMTCB is defined as follows in the header file nfsmt.h

```
typedef struct {
        int         command;
        int         sub_command;
        int         param1;
        int         param2;
        int         param3;
        char        *where;
        int         length;
} SMTCB;
```

SMT_GET:

SMT_GET provides a variety of functions such as to read the HPC registers and to get the smt status. command field of smtp should be initialized to one of the following values

```
        HPC_BMAC1_REGS          : To read the BMAC registers
        HPC_READ                : To read the HPC registers
        HPC_PORT1_REGS          : To read RMT port1
        and HPC_PORT2_REGS        and port2 registers
```

Some of the commands provide sub commands. The field sub_command should be initialzed to these sub commands.

1. HPC_BMAC1_REGS

HPC_BMAC1_REGS enables the user to read the BMAC registers. HPC_BMAC1_REGS provides two sub commands GET_COUNTER_GROUP and GET_NEIGHBOR_ADDR. GET_COUNTER_GROUP is used to get various SMT counter values.

GET_COUNTER_GROUP needs the SMTCB *smtp to be initialized as follows

```
COUNTER_GROUP  ct;

smtp->command = HPC_BMAC1_REGS;
smtp->sub_command = GET_COUNTER_GROUP;
smtp->where = (char *) &ct;
smtp->length = sizeof (ct);
```

GET_NEIGHBOR_ADDR enables the user to get the MAC address of the Neighbour station. GET_NEIGHBOR_ADDR needs the SMTCB *smtp to be initialized as follows

```
char          addr_buf[12];

smtp->command = HPC_BMAC1_REGS;
smtp->sub_command = GET_NEIGHBOR_ADDR;
smtp->where = addr_buf;
smtp->length = 12;
```

2. HPC_READ

HPC_READ enables the user to read the HPC registers. HPC_READ does not provide any sub commands. HPC_READ needs the SMTCB *smtp to be initialized as follows

```
smtp->command = HPC_READ;
smtp->param1 = HPC_READ | HPC_SIZE_BYTE
                        | <HPC_reg_offset>;
smtp->where = (char *) smtp;
```

where HPC_register_offset offset is set of register space provided by the HPC. For the set of reister offsets refer to the file nfsmt.h

3. HPC_PORT1_REGS and HPC_PORT2_REGS

HPC_PORT1_REGS enables the user to get the status of the Connection Management. HPC_PORT2_REGS is for the second port if the interface is a DAS. The sub command for HPC_PORT1_REGS is GET_PORT_GROUP. HPC_PORT1_REGS needs the SMTCB *smtp to be initialized as follows

```
FDDI_PORT_GROUP port;

smtp->command = HPC_PORT1_REGS;
smtp->sub_command = GET_PORT_GROUP;
smtp->where = (char *) &port;
smtp->length = sizeof (port);
```

The two important status returned in the structure port are port.ecm_state and port.pcm_state. port.ecm_state corresponds to the current state of the ECM state machine. The valid values are OUT, IN, TRACE, PATHTEST, INSERT, CHECK and DEINSERT. The value returned in port.ecm_state is the index into the list of the ECM

States. port.pcm_state corresponds to the current state of the PCM state machine. The Valid values are OFF, BREAK, TRACE, CONNECT, NEXT, SIGNAL, JOIN, VERIFY, ACTIVE, MAINT. The value returned in port.pcm_state in an index into the list of PCM States.

SMT_ACT:

SMT_ACT is supported to set the state of the smt driver. The command field should always be set to SMT_CTL. SMT_ACT provides two sub commands SMT_ACCEPT_FRAME and SMT_CLOSE. SMT_ACCEPT_FRAME needs to be used when any SMT API client is active.

```
smtp->command = SMT_CTL;
smtp->sub_command = SMT_ACCEPT_FRAME;
```

SMT_CLOSE needs to be used when the API client exits.

```
smtp->command = SMT_CTL;
smtp->sub_command = SMT_CLOSE;
```

FILES **/dev/smt**

SEE ALSO **nf**(7), **dlpi**(7),

NAME

ugen – USB generic driver

SYNOPSIS

Node Name@unit-address

#include <sys/usb/clients/ugen/usb_ugen.h>

ugen is a generic USBA (Solaris USB Architecture) compliant client character driver that presents USB devices to applications through a standard **open**(2), **close**(2), **read**(2), **write**(2), **aioread**(3AIO), **aiowrite**(3AIO) Unix interface. Uninterpreted raw data are transferred to and from the device via file descriptors created for each USB endpoint. Status is obtained by reading file descriptors created for endpoint and full device status.

ugen supports control, bulk, and interrupt-IN transfers. Isochronous and interrupt-OUT transfers are not supported.

BINDING

ugen can bind to a device with one or more interfaces in its entirety, or to a single interface of that device. The binding type depends on information that is passed to **add_drv**(1M) or **update_drv**(1M).

An **add_drv**(1M) command binds **ugen** to a list of device types it is to control. **update_drv**(1M) adds an additional device type to the list of device types being managed by the driver.

Names used to bind drivers to the entire device, as well as those used for binding to just one interface, are shown in the output of the **prtconf -v** command. A list of names for each device is shown in the ouput as that device's "compatible" property. Each name in the list is called a "compatible" name. Be sure the device is powered on and connected before you issue **prtconf -v**.

prtconf entries for most devices with multiple interfaces have "usb,device" as their last compatible name. Their "compatible" property is similar to:

```
name='compatible' type=string items=5
  value='usb472,b0b0.100.config1' + 'usb472,b0b0.100' +
 'usb472,b0b0.1' + 'usb472,b0b0' + 'usb,device'
```

Names are listed from most specific to most general, and the system searches them in that order. Specify any name listed before "usb,device" to bind the entire device to **ugen**. "usb,device" must not be used. If the system finds no other matching name first, "usb,device" binds the full device to **usb_mid**(7D). **usb_mid** then creates a child for each interface, enabling different drivers to bind to each child. Each child will then have its own "compatible" property with a list of names. All compatible names of interface children begin with "usbif." For example:

```
name='compatible' type=string items=2
 value='usbif472,b0b0.100.config1.0' + 'usbif472,b0b0.config1.0'
```

If the device has just one configuration and one interface, and is of device class 0, no default "usb,device" compatible name is added; instead a list of "usbif" compatible names is appended. For example:

```
name='compatible' type=string items=8
  value='usb430,100.105' + 'usb430,100' + 'usbif430,class3.1.2' +
  'usbif430,class3.1' + 'usbif430,class3' + 'usbif,class3.1.2' +
 'usbif,class3.1' + 'usbif,class3'
```

To bind the new device type while keeping the original device types, issue an **update_drv**(1M) command of the following form (on a single line):

```
update_drv -a -m '* <device perms> <owner> <group>'
   -i '"<new device type>"' ugen
```

or for the dual framework configuration:

```
update drv -a -m '* <device perms>  <owner>  <group>'
   -i '"<new device type>"' usba10_ugen
```

An example showing how to bind an entire composite device follows:

```
update_drv -a -m '* 0666 root sys' -i '"usb472,b0b0"' ugen
```

or for the dual framework configuration:

```
update_drv -a -m '* 0666 root sys' -i '"usb472,b0b0"' usba10_ugen
```

Compatible names representing single interfaces of composite devices are of the following form:

```
"usbif<vid>,<pid>.config<cfg value>.<interface number>"
```

An example showing how to bind a child device representing interface 0 of configuration 1 of a composite device follows:

```
update_drv -a -m '* 0666 root sys'
   -i '"usbif472,b0b0.config1.0"' ugen
```

or for the dual framework configuration:

```
update_drv -a -m '* 0666 root sys'
 -i '"usbif472,b0b0.config1.0"' usba10_ugen
```

LOGICAL DEVICE NAME FORMAT

For each device or child device it manages, **ugen** creates one logical device name for device-wide status and one logical device name for endpoint 0. (If **ugen** controls multiple child devices that correspond to different interfaces of the same device, the multiple device-wide status and endpoint logical device names created will share control and access the same device pipes.) **ugen** also creates logical device names for all other endpoints within the device node's binding scope (interface or device), plus logical device names for their status.

When **ugen** is bound to an entire device, the following logical device names are created (each on a single line). *N* represents the instance number of the device type.

```
Endpoint 0 (default endpoint):

        /dev/usb/<vid>.<pid>/<N>/cntrl0
        /dev/usb/<vid>.<pid>/<N>/cntrl0stat

    For example:

        /dev/usb/472.b0b0/0/cntrl0
        /dev/usb/472.b0b0/0/cntrl0stat

Configuration 1, Endpoints > 0, alternate 0:

        /dev/usb/<vid>.<pid>/<N>/if<interface#>
                                <in|out|cntrl><endpoint#>
        /dev/usb/<vid>.<pid>/<N>/if<interface#>
                                <in|out|cntrl><endpoint#>stat

    For example:

        /dev/usb/472.b0b0/0/if0in1
        /dev/usb/472.b0b0/0/if0in1stat

Configuration 1, Endpoints > 0, alternate > 0:

        /dev/usb/<vid>.<pid>/<N>/if<interface#>.
                                <alternate><in|out|cntrl><endpoint#>
        /dev/usb/<vid>.<pid>/<N>/if<interface#>.
                                <alternate<in|out|cntrl><endpoint#>stat

    For example:

        /dev/usb/472.b0b0/0/if0.1in3
        /dev/usb/472.b0b0/0/if0.1in3stat

Configuration > 1, Endpoints > 0, alternate 0:

        /dev/usb/<vid>.<pid>/<N>/cfg<value>if<interface#>
                                <in|out|cntrl><endpoint#>
        /dev/usb/<vid>.<pid>/<N>/cfg<value>if<interface#>
                                <in|out|cntrl><endpoint#>stat

    For example:

       /dev/usb/472.b0b0/0/cfg2if0in1
       /dev/usb/472.b0b0/0/cfg2if0in1stat

Configuration > 1, Endpoints > 0, alternate > 0:
        /dev/usb/<vid>.<pid>/<N>/cfg<value>if<interface#>.
                                <alternate<in|out|cntrl><endpoint#>
        /dev/usb/<vid>.<pid>/<N>/cfg<value>if<interface#>.
                                <alternate<in|out|cntrl><endpoint#>stat

    For example:

        /dev/usb/472.b0b0/0/cfg2if0.1in1
        /dev/usb/472.b0b0/0/cfg2if0.1in1stat

  Device status:

        /dev/usb/<vid>.<pid>/<N>/devstat

    For example:

        /dev/usb/472.b0b0/0/devstat
```

When **ugen** is bound to a single device interface, the following logical device nodes are created:

```
Endpoint 0 (default endpoint):

         /dev/usb/<vid>.<pid>/<N>/if<interface#>cntrl0
         /dev/usb/<vid>.<pid>/<N>/if<interface#>cntrl0stat

    For example:

         /dev/usb/472.b0b0/0/if0cntrl0
```

/dev/usb/472.b0b0/0/if0cntrl0stat

The format for all other logical device names is identical to the format used when **ugen** is bound to the entire device.

Opening the endpoint of a different configuration or different alternate interface will cause an implicit change of configuration or a switch to an alternate interface. A configuration change is prohibited when any non-zero endpoint device nodes are open. An alternate interface switch is prohibited if any endpoint in the same interface is open.

All **ugen** logical device name files must be opened exclusively using the O_EXCL flag. Opens attempted without O_EXCL fail with EACCES. All logical device name files created for returning status must also be opened with O_RDONLY.

HOT-PLUGGING

A device may be hot-removed at any time. Following hot-removal, the device status changes to USB_DEV_STAT_DISCONNECTED, the status of open endpoints change to USB_LC_STAT_DISCONNECTED upon their access, and all subsequent transfer requests fail. Endpoints are reactivated by first reinserting the device and then closing and reopening all endpoints that were open when the device was disconnected.

CPR (CHECKPOINT/RESUME)

CPR (Checkpoint/Resume) may be initiated at any time and is treated similarly to a hot-removal. Upon successful suspend and resume, all subsequent transfer requests fail as an indication to the application to reinitialize. Applications should close and reopen all endpoints to reinstate them. All endpoint and device status on Resume (before close and reopen) is USB_LC_STAT_SUSPENDED. A system suspend will fail while **ugen** is performing a transfer.

DEVICE STATUS MANAGEMENT

Applications can monitor device status changes by reading the device status from the device status logical name. When opened without O_NONBLOCK and O_NDELAY, all reads from that file descriptor (with the exception of the the intial read that follows the open) block until a device status change occurs.

Device statuses are:

USB_DEV_STAT_ONLINE
: Device is available.

USB_DEV_STAT_DISCONNECTED
: Device has been disconnected.

USB_DEV_STAT_RESUMED
: Device has been resumed, however, endpoints which were open on suspend have not yet been closed and reopened.

USB_DEV_STAT_UNAVAILABLE
: Device has been reconnected, however, endpoints which were open on disconnect have not yet been closed and reopened, or the device is powered down.

The following code reads the device status device logical name:

```
int fd;
int status;

if ((fd = open("/dev/usb/472.b0b0/0/devstat",
   O_RDONLY|O_EXCL)) < 0) {
        /* handle error */
}

if (read(fd, &status, sizeof(status))  != sizeof(status)) {
        /* handle error */
}
```

```
switch (status) {
case USB_DEV_STAT_DISCONNECTED:
        printf ("Terminating as device has been disconnected./n");
        exit (0);

case USB_DEV_STAT_RESUMED:
case USB_DEV_STAT_UNAVAILABLE:
        /*
         * Close and reopen endpoints to reestablish device access,
         * then reset device.
         */
        break;

case USB_DEV_STAT_ONLINE:
default:
        break;
}
```

Endpoint status is returned via the endpoint status device logical names. See the ERRORS section for more information on endpoint status values.

CONTROL TRANSFERS

Applications requiring I/O on a control endpoint should open the corresponding logical device name and use regular UNIX I/O system calls. For example: **read**(2), **write**(2), **aioread**(3AIO) and **aiowrite**(3AIO). **poll**(2) is not supported on control endpoints.

A control endpoint must be opened with **O_EXCL | O_RDWR** and cannot be opened with **O_NONBLOCK** or **O_NDELAY**.

For example:

```
fd = open("/dev/usb/472.b0b0/0/cntrl0", O_EXCL | O_RDWR);

fdstat = open("/dev/usb/472.b0b0/0/cntrl0stat", O_EXCL | O_RDONLY);
```

Control endpoints can be read and written. A **read** operation receives data *from* the device and a **write** operation sends data *to* the device.

To perform a control-IN transfer, perform a **write**(2) of USB setup data (see section 9.3 of the *USB 2.0* specification) followed by a **read**(2) on the same control endpoint to fetch the desired data. For example:

```
void init_cntrl_req(
    uchar_t *req, uchar_t bmRequestType, uchar_t bRequest,
    ushort_t wValue, ushort_t wIndex, ushort_t wLength)
        {
                req[0] = bmRequestType;
                req[1] = bRequest;
                req[2] = 0xFF & wValue;
                req[3] = 0xFF & (wValue >> 8);
                req[4] = 0xFF & wIndex;
                req[5] = 0xFF & (wIndex >> 8);
                req[6] = 0xFF & wLength;
                req[7] = 0xFF & (wLength >> 8);
        }

        ....


        uchar_t dev_descr_req[8];
        usb_dev_descr_t descr;

        init_cntrl_req(dev_descr_req,
            USB_DEV_REQ_DEV_TO_HOST, USB_REQ_GET_DESCR,
            USB_DESCR_TYPE_SETUP_DEV, 0, sizeof (descr));

        count = write(fd, dev_descr_req, sizeof (dev_descr_req));
        if (count != sizeof (dev_descr_req)) {
                /* do some error recovery */
                ...
        }

        count = read(fd, &descr, sizeof (descr));
        if (count != sizeof (descr)) {
                /* do some error recovery */
    }
```

The application can issue any number of reads to read data received on a control end-point. **ugen** successfully completes all reads, returning the number of bytes transferred. Zero is returned when there is no data to transfer.

If the **read/write** fails and returns **-1**, you can access the endpoint's status device logical name for precise error information:

```
int status;

        count = read(fdstat, &status, sizeof (status));
        if (count == sizeof (status)) {
                switch (status) {
                case USB_LC_STAT_SUSPENDED:
                case USB_LC_STAT_DISCONNECTED:
                        /* close all endpoints */
                        ...
                        break;
                default:
                        ...
                        break;
                }
        }
```

Refer to the ERRORS section for all possible error values.

To perform a control-OUT transfer, write the USB setup data followed by any accompanying data bytes.

```
    init_cntrl_req(wbuf, .......);
    bcopy(data, &wuf[8], sizeof (data);

    count = write(fd, wbuf, sizeof (wbuf));
```

A **write**(2) returns the number of bytes actually transferred, (whether or not the **write** is completely successful), provided that some data is actually data transferred. When no data is transferred, **write**(2) returns **-1**. Applications can read the corresponding endpoint status to retrieve detailed error information.

INTERRUPT TRANSFERS

Applications requiring data from an interrupt-IN endpoint should open the corresponding logical device name and use **read**(2), **aioread**(3AIO) and **poll**(2) system calls. Interrupt-OUT endpoints are not currently supported.

An interrupt endpoint must be opened with **O_EXCL | O_RDONLY**. It can also be opened using **O_NONBLOCK** or **O_NDELAY** if desired.

```
fd = open("/dev/usb/472.b0b0/0/if0in1", O_EXCL | O_RDONLY);
fdstat = open("/dev/usb/472.b0b0/0/if0in1stat", O_EXCL | O_RDONLY);
```

ugen starts polling interrupt endpoints immediately upon opening them and stops polling them upon closure. (Polling refers to interrogation of the device by the driver and should not be confused with **poll**(2), which is an interrogation of the driver by the application.)

A **read**(2) of an endpoint opened with the **O_NONBLOCK** or **O_NDELAY** flags set will not block when there is insufficient data available to satisfy the request. The **read** simply returns what it can without signifying any error.

ugen enables buffering of up to one second of incoming data. In case of buffer overflow, **ugen** stops polling the interrupt endpoint until the application consumes all the data. A **read**(2) of an empty buffer returns **-1**, sets the endpoint status to **USB_LC_STAT_INTR_BUF_FULL** and causes **ugen** to start polling the endpoint again. To retrieve the status, the application can open and read the corresponding endpoint's status device logical name.

```
for (;;) {
        count = read(fd, buf, sizeof(buf));
        if (count == -1) {
                int cnt, status;

                cnt = read(fdstat, &status, sizeof (status));
                if (cnt == -1) {
                         /* more error recovery here */
                } else {
                        switch (status) {
                        case USB_LC_STAT_INTR_BUF_FULL:
                               ...
                               break;
                        default:
                               ...
                               break;
                        }
                }
           }
           /* process the data */
           ....
   }
```

ugen will never drop data. However, the device may drop data if the application cannot read it at the rate that it is produced.

An application can open multiple interrupt-IN endpoints and can call **poll**(2) to monitor the availability of new data.

```
        struct pollfd pfd[2];

        bzero(pfd, sizeof (pfd));
        pfd[0].fd = fd1; /* fd1 is one interrupt endpoint. */
        pfd[0].events = POLLIN;
        pfd[1].fd = fd2; /* fd2 is another interrupt endpoint. */
        pfd[1].events = POLLIN;

        for (;;) {
                poll(pfd, 2, -1);

                if (pfd[0].revents & POLLIN) {
                        count = read(fd1, buf, sizeof (buf));
                        ....
                }
                if (pfd[1].revents & POLLIN) {
                        count = read(fd2, buf, sizeof (buf));
                        ....
                }
   }
```

poll(2) can also be used for concurrent monitoring of multiple interrupt endpoints and device status. Substitute the file descriptor of the device status endpoint (opened without O_NONBLOCK or O_NDELAY) for one of the interrupt endpoints in the code example above.

BULK TRANSFERS

Applications requiring I/O on a bulk endpoint can open the corresponding logical device name and perform regular UNIX I/O system calls. For example: **read**(2), **write**(2), **aioread**(3AIO) and **aiowrite**(3AIO). **poll**(2) is not supported on bulk endpoints.

A bulk endpoint must be opened with **O_EXCL | O_RDWR** and cannot be opened with **O_NONBLOCK** or **O_NDELAY:**

fd = open("/dev/usb/472.b0b0/0/if0in2", O_EXCL | O_RDWR);

fdstat = open("/dev/usb/472.b0b0/0/if0in2stat", O_EXCL | O_RDONLY);

Data can be read from a bulk-IN endpoint as follows:

```
count = read(fd, buf, sizeof (buf)):
        if (count == -1) {
                /* error recovery */
        }

        Data can be written to a bulk-OUT endpoint as follows:

        count = write(fd, buf, sizeof (buf)):
        if (count == -1) {
                /* error recovery */
    }
```

ERRORS

The following statuses are returned by endpoint status device logical names:

USB_LC_STAT_NOERROR
No error.

USB_LC_STAT_CRC
CRC error detected.

USB_LC_STAT_BITSTUFFING
Bit stuffing error.

USB_LC_STAT_DATA_TOGGLE_MM
Data toggle did not match.

USB_LC_STAT_STALL
Endpoint returned stall.

USB_LC_STAT_DEV_NOT_RESP
Device not responding.

USB_LC_STAT_UNEXP_PID
Unexpected Packet Identifier (PID).

USB_LC_STAT_PID_CHECKFAILURE
Check bits on PID failed.

USB_LC_STAT_DATA_OVERRUN
Data overrun.

USB_LC_STAT_DATA_UNDERRUN
Data underrun.

USB_LC_STAT_BUFFER_OVERRUN
: Buffer overrun.

USB_LC_STAT_BUFFER_UNDERRUN
: Buffer underrun.

USB_LC_STAT_TIMEOUT
: Command timed out.

USB_LC_STAT_NOT_ACCESSED
: Not accessed by the hardware.

USB_LC_STAT_UNSPECIFIED_ERR
: Unspecified USBA or HCD error.

USB_LC_STAT_NO_BANDWIDTH
: No bandwidth available.

USB_LC_STAT_HW_ERR
: Host Controller h/w error.

USB_LC_STAT_SUSPENDED
: Device was suspended.

USB_LC_STAT_DISCONNECTED
: Device was disconnected.

USB_LC_STAT_INTR_BUF_FULL
: Interrupt data buffer full.

USB_LC_STAT_INTERRUPTED
: Request was interrupted.

USB_LC_STAT_NO_RESOURCES
: No resources available for request.

USB_LC_STAT_INTR_POLLING_FAILED
: Failed to restart polling.

The following system call **errno** values are returned:

EBUSY
: The endpoint has been opened and another open is attempted.

EACCES
: An endpoint open was attempted with incorrect flags.

ENOTSUP
: Operation not supported.

ENXIO
: Device associated with the file descriptor does not exist.

ENODEV
: Device has been hot-removed or a suspend/resume happened before this command.

EIO
: An I/O error occurred. Send a read on the endpoint status minor node to get the exact error information.

EINTR
: Interrupted system call.

ENOMEM
: No memory for the allocation of internal structures.

```
   /dev/usb/<vid>.<pid>/<N>/cntrl0
   /dev/usb/<vid>.<pid>/<N>/cntrl0stat

   /dev/usb/<vid>.<pid>/<N>/if<interface#>
                   <in|out|cntrl><endpoint#>
   /dev/usb/<vid>.<pid>/<N>/if<interface#>
                   <in|out|cntrl><endpoint#>stat

   /dev/usb/<vid>.<pid>/<N>/if<interface#>.
                   <alternate><in|out|cntrl<endpoint#>
   /dev/usb/<vid>.<pid>/<N>/if<interface#>.
                   <alternate><in|out|cntrl><endpoint#>stat

   /dev/usb/<vid>.<pid>/<N>/cfg<value>if<interface#>
                   <in|out|cntrl><endpoint#>
   /dev/usb/<vid>.<pid>/<N>/cfg<value>if<interface#>
                   <in|out|cntrl<endpoint#stat>

   /dev/usb/<vid>.<pid>/<N>/cfg<value>if<interface#>.
                   <alternate><in|out|cntrl><endpoint#>
   /dev/usb/<vid>.<pid>/<N>/cfg<value>if<interface#>.
                   <alternate><in|out|cntrl><endpoint#>stat


   /dev/usb/<vid>.<pid>/<N>/devstat

   /dev/usb/<vid>.<pid>/<N>/if<interface#>cntrl0
   /dev/usb<vid>.<pid>/<N>/if<interface#>cntrl0stat
```

where *N* is an integer representing the instance number of this type of device. (All logical device names for a single device share the same *N*.)

/kernel/drv/usba10_ugen
: 32 bit ELF kernel module for USBA 1.0 framework*

/kernel/drv/sparcv9/usba10_ugen
: 64 bit ELF kernel module for USBA 1.0 framework*

/kernel/drv/usba10_ugen.conf
: Configuration file needed for **usba10_ugen**.

∗ Please see *www.sun.com/desktop/whitepapers.html* for more information regarding USB dual framework implementation, *USBA 1.0*, and *USB 2.0*.

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based SPARC systems
Availability	SUNWusb, SUNWusbx

usba(7D)

http://www.sun.com/desktop/whitepapers.html

DIAGNOSTICS

Instance number too high (<value>)
  Too many devices are using this driver.

Too many minor nodes
  Device has too many minor nodes. Not all are available.

NOTES

Isochronous and interrupt-OUT transfers are not supported.

ugen returns **-1** for all commands and sets **errno** to **ENODEV** when device has been hot-removed or resumed from a suspend. The application must close and reopen all open minor nodes to reinstate successful communication.

ugen is available only through *USB 2.0* ports operated by the *USBA 1.0* framework. Please see *www.sun.com/desktop/whitepapers.html* for more information regarding USB dual framework implementation, USBA 1.0, and USB 2.0.

NAME

usb_ac – USB audio control driver

SYNOPSIS

sound-control@unit-address

The **usb_ac** driver is a USBA (Solaris USB Architecture) compliant client driver that supports the *USB Audio Class 1.0* specification.

The audio control driver is a USB class driver and offers functionality similar to the **audiocs** (sun4u) and **audiots** (Sun Blade 100) drivers which use the Solaris audio mixer framework (**mixer**(7I)). Unlike the **audiocs** and **audiots** drivers, the USB audio device may have play-only or record-only capability.

Drivers corresponding to other USB audio interfaces on the device, including the **usb_as**(7D) audio streaming driver or the **hid**(7D) driver, are plumbed under the USB audio control driver and do not directly interface with user applications.

The **usb_ac** driver supports USB audio class compliant devices with a feature unit. For a list of recommended devices, visit: *www.sun.com/io.*

APPLICATION PROGRAM INTERFACE

This interface is described in the **mixer**(7I) and **audio**(7I) man pages.

Driver Versions

Applications that open **/dev/audio** may use the **AUDIO_GETDEV ioctl( )** to determine which audio device is being used. The USB audio driver returns the string "USB Audio" in the name field of the audio_device structure. The version field displays the version number and the config field displays the string "external."

The USB audio device provides support for an external speaker and microphone.

Audio Mixer Mode

Use the **/usr/kernel/drv/usb_ac.conf** (original USBA framework) and **/usr/kernel/drv/usba10_usb_ac.conf** (USBA 1.0 framework) configururation files to configure the USB audio driver. These files determine whether the audio mixer is enabled or disabled. See the **mixer**(7I) manual page for details. You can change the audio mixer mode at any time by using the **mixerctl(1)** or **sdtaudiocontrol(1)** applications.

Audio Data Formats

The USB audio device supports the audio data formats shown below. Please note that at a minimum, the device must support a sampling frequency of 44100 Hz or 48000 Hz. In the table below, mode "M" indicates that mixer mode is enabled, while "C" indicates that mixer mode is disabled or in compatibility mode.

Sample Rate	Encoding	Precision	Channels	Mode
8000 Hz	u-Law or A-Law	8	1 or 2	M and C
9600 Hz	u-Law or A-Law	8	1 or 2	M and C
11025 Hz	u-law or A-law	8	1 or 2	M and C
16000 Hz	u-law or A-law	8	1 or 2	M and C
18900 Hz	u-law or A-law	8	1 or 2	M and C
22050 Hz	u-law or A-law	8	1 or 2	M and C
32000 Hz	u-law or A-law	8	1 or 2	M and C
33075 Hz	u-law or A-law	8	1 or 2	M and C

37800 Hz	u-law or A-law	8	1 or 2	M and C
44100 Hz	u-law or A-law	8	1 or 2	M and C
48000 Hz	u-law or A-law	8	1 or 2	M and C
8000 Hz	linear	8 or 16	1 or 2	M and C
9600 Hz	linear	8 or 16	1 or 2	M and C
11025 Hz	linear	8 or 16	1 or 2	M and C
16000 Hz	linear	8 or 16	1 or 2	M and C
18900 Hz	linear	8 or 16	1 or 2	M and C
22050 Hz	linear	8 or 16	1 or 2	M and C
32000 Hz	linear	8 or 16	1 or 2	M and C
33075 Hz	linear	8 or 16	1 or 2	M and C
37800 Hz	linear	8 or 16	1 or 2	M and C
44100 Hz	linear	8 or 16	1 or 2	M and C
48000 Hz	linear	8 or 16	1 or 2	M and C

Audio Status Change Notification

As described in the **audio**(7I) and **mixer**(7I) man pages, it is possible to request asynchronous notification of changes in the state of an audio device.

ERRORS

If a device is hot-removed while it is active, all subsequent opens will return EIO. All other errors are defined in the **audio**(7I) man page.

/usr/kernel/drv/usb_ac
32 bit ELF kernel module for original USBA framework.∗

/usr/kernel/drv/sparcv9/usb_ac
64 bit ELF kernel module for original USBA framework.∗

/usr/kernel/drv/usb_ac.conf
usb_ac audio driver configuration file.

/usr/kernel/drv/usba10_usb_ac
32 bit ELF kernel module for USBA 1.0 framework.∗

/usr/kernel/drv/sparcv9/usba10_usb_ac
64 bit ELF kernel module for USBA 1.0 framework.∗

/usr/kernel/drv/usba10_usb_ac.conf
usba10_usb_ac audio driver configuration file.

/dev/audio
Symlink to the system's primary audio device, not necessarily a USB audio device.

/dev/audioctl
/dev/audio control device.

/dev/sound/[0-N]
Represents the audio devices on the system and is not necessarily a USB audio device.

/dev/sound/[0-N]ctl

/dev/sound audio control device.

∗Please see *www.sun.com/desktop/whitepapers.html* for more information regarding USB dual framework implementation, USBA 1.0, and USB 2.0

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	Original USBA drivers and files: PCI-based systems USBA 1.0 drivers and files: PCI-based SPARC systems
Availability	SUNWaud (32-bit)
	SUNWaudx (64-bit)
Stability level	Evolving

mixerctl(1), **cfgadm_usb**(1M), **ioctl**(2), **attributes**(5), **hid**(7D), **usba**(7D), **usb_as**(7D), **audio**(7I), **mixer**(7I), **streamio**(7I), **usb_ah**(7M)

Writing Device Drivers

Universal Serial Bus Specification 2.0

Universal Serial Bus Device Class Definition for Audio Devices, Release 1.0

System Administration: Basic Administration

http://www.sun.com/desktop/whitepapers.html

http://www.sun.com/io

DIAGNOSTICS

In addition to being logged, the following messages may appear on the system console. All messages are formatted in the following manner:

Warning: <device path> (usb_ac<instance num>): Error Message...

or

Warning: <device path> (usba10_usb_ac<instance num>):
Error Message...

Failure to plumb audio streams drivers.
: The usb audio streaming driver or the **hid** driver could not be plumbed under the audio control driver and the device is not usable.

Device was disconnected while open. Data may have been
: lost." 6 The device was hot-removed or powered off while it was open and a possible data transfer was in progress. The job was aborted.

Cannot access device. Please reconnect <name>.
: There was an error in accessing the device during reconnect. Please reconnect the device.

Device is not identical to the previous one on this port.

Please disconnect and reconnect." 6 A USB audio device was hot-removed while open. A new device was hot-inserted which is not identical to the original USB audio device. Please disconnect the new USB device and reconnect the original device to the same port.

Busy device has been reconnected.
A device that was hot-removed from a USB port has been re-inserted again.

NOTES

The USB audio device will be power managed if the device is idle.

USB audio devices do not have line out or port control.

If a USB audio device is hot-removed while active, it prints a console warning message requesting you to put the device back in the same port and informing you that there may be data loss. Hot-removal of an active audio device is strongly discouraged.

Close all applications before hot-removing or hot-inserting a device. If an application is open when a device is hot-removed, inserting the device in a different port will create new **/dev/sound** links but **/dev/audio** will not be affected. Hotplugging an active device is not recommended.

On slower IA machines and with higher frequency sample rates, you may encounter some audio quality problems.

To make a USB audio device the primary audio device (for example: **/dev/audio**), close all audio applications, disconnect all USB audio devices, modunload all other audio drivers and then simply reconnect the USB audio device. This will cause **/dev/audio** to point to the USB audio **/dev/sound** entry.

Most Solaris audio applications and 3rd party audio applications available on Solaris work well with USB audio devices. For details of the application behavior with USB audio devices, visit *www.sun.com/io*.

NAME

usb_ah – USB audio HID STREAMS module

The **usb_ah** STREAMS module enables the USB input control device which is a member of the Human Interface Device (HID) class and provides support for volume change and mute button. The **usb_ah** module is pushed on top of a HID class driver instance (see **hid**(7D)) and below an Audio Control class driver instance (see **usb_ac**(7D)). It translates the HID specific events to the events that are supported by the Solaris audio mixer framework.

/kernel/strmod/usb_ah
32-bit ELF kernel STREAMS module for USBA 1.0 framework∗

/kernel/strmod/sparcv9/usb_ah
64-bit ELF kernel STREAMS module for USBA 1.0 framework∗

∗ Please see *www.sun.com/desktop/whitepapers.html* for more information regarding USB dual framework implementation, *USBA 1.0*, and *USB 2.0*.

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based SPARC systems
Availability	SUNWusb, SUNWusbx
Interface Stability	Evolving

mixerctl(1), **hid**(7D), **usba**(7D), **usb_ac**(7D), **usb_as**(7D), **usb_mid**(7D), **audio**(7I), **mixer**(7I)

STREAMS Programming Guide

System Administration Guide: Basic Administration

Universal Serial Bus Specification 2.0

Device Class Definition for Human Interface Devices (HID) 1.1

http://www.sun.com/desktop/whitepapers.html

DIAGNOSTICS

None

NOTES

If USB audio drivers are not loaded, buttons will not be active.

NAME usb_as – USB audio streaming driver

SYNOPSIS sound@unit-address

The **usb_as** driver is a USBA (Solaris USB Architecture) compliant client driver that supports the *USB Audio Class 1.0* specification.

The **usb_as** driver processes audio data messages during play and record and sets sample frequency, precision, encoding and other functions on request from the USB audio control driver. See **usb_ac**(7D).

This driver is plumbed under the USB audio control driver and does not directly interface with the user application.

/usr/kernel/drv/usb_as
  32 bit ELF kernel module for original USBA framework*

/usr/kernel/drv/sparcv9/usb_as
  64 bit ELF kernel module for original USBA framework*

/usr/kernel/drv/usba10_usb_as
  32 bit ELF kernel module for USBA 1.0 framework*

/usr/kernel/drv/sparcv9/usba10_usb_as
  64 bit ELF kernel module for USBA 1.0 framework*

/usr/kernel/drv/usba10_usb_as.conf
  usba10_usb_as configuration file

* Please see *www.sun.com/desktop/whitepapers.html* for more information regarding USB dual framework implementation, *USBA 1.0*, and *USB 2.0*.

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	Original USBA drivers and files: PCI-based systems USBA 1.0 drivers and files: PCI-based SPARC systems
Availability	SUNWaud, SUNWaudx
Stability level	Evolving

mixerctl(1), **attributes**(5), **usba**(7D), **usb_ac**(7D), **audio**(7I), **mixer**(7I), **streamio**(7I)

Writing Device Drivers

Universal Serial Bus Specification 2.0

System Administration Guide: Basic Administration

http://www.sun.com/desktop/whitepapers.html

http://*www.sun.com/io*

DIAGNOSTICS

In addition to being logged, the following messages may appear on the system console. All messages are formatted in the following manner:

Warning: <device path> usb_as<instance num>: Error Message...

or

Warning: <device path> usba10_usb_as<instance num>:
Error Message...

where <device path> is the physical path to the device in **/devices** directory.

No bandwidth available.
: There is no bandwidth available for the isochronous pipe. As a result, no data will be transferred during play and record.

Cannot access device. Please reconnect <name>.
: There was an error in accessing the device during reconnect. Please reconnect the device.

Device is not identical to the previous one on this port.
: Please disconnect and reconnect." 6 A USB audio streaming interface was hot-removed while open. A new device was hot-inserted which is not identical to the original USB audio device. Please disconnect the new USB device and reconnect the original device to the same port.

NOTES

The USB audio streaming interface will be power managed if device is idle.

NAME usb_mid – USB Multi Interface Driver

SYNOPSIS **device**@*unit-address*

The **usb_mid** driver is a **USBA** (Solaris Universal Serial Bus Architecture) compliant nexus driver that binds to device level nodes of a composite (multi interface) device if no vendor or class specific driver is available. The **usb_mid** driver attempts to bind drivers to each of the composite device's interfaces.

/kernel/drv/usb_mid
32-bit ELF kernel module for original USBA framework∗

/kernel/drv/sparcv9/usb_mid
64-bit ELF kernel module for original USBA framework∗

/kernel/drv/usba10_usb_mid
32 bit ELF kernel module for USBA 1.0 framework∗

/kernel/drv/sparcv9/usba10_usb_mid
64 bit ELF kernel module for USBA 1.0 framework∗

kernel/drv/usba10_usb_mid.conf
usba10_usb_mid configuration file

∗ Please see *www.sun.com/desktop/whitepapers.html* for more information regarding USB dual framework implementation, *USBA 1.0*, and *USB 2.0*.

See **attributes**(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	Original USBA drivers and files: PCI-based systems USBA 1.0 drivers and files: PCI-based SPARC systems
Availability	SUNWusbx (32-bit)
	SUNWusb (64-bit)

cfgadm_usb(1M), **attributes**(5), **usba**(7D)

Universal Serial Bus Specification 2.0

System Administration Guide: Basic Administration

http://www.sun.com/desktop/whitepapers.html

http://www.sun.com/io

In addition to being logged, the following messages may appear on the system console. All messages are formatted in the following manner:

Warning: <device path> usb_mid<instance number>: Error Message...

or

Warning: <device path> usba10_usb_mid<instance number:
Error Message...

Cannot access device. Please reconnect <*device name*>.
: This device has been disconnected because a device other than the original one has been inserted. The driver informs you of this fact by displaying the name of the original device.

Device not identical to the previous one on this port.
: Please disconnect and reconnect." 6 Same condition as described above; however in this case, the driver is unable to identify the original device with a name string.

NAME

usb_sd – USB disk and storage device driver

SYNOPSIS

disk@target,lun:partition

The **usb_sd** driver supports devices which comply with the USB mass storage specification. It works in conjunction with the **scsa2usb**(7D) driver. It treats all USB devices as removable media by default, unless a device is exempted by a **scsa2usb** configuration file entry. (refer to **scsa2usb**(7D)).

To determine the disk drive type, use the **SCSI/ATAPI** inquiry command and read the volume label stored on block 0 of the drive. (The volume label describes the disk geometry and partitioning and must be present for the disk to be mounted by the system.) A volume label is not required for removable, rewritable or read-only media.

DEVICE SPECIAL FILES

Block-files access the disk using normal buffering mechanism and are read-from and written-to without regard to physical disk records. A "raw" interface enables direct transmission between the disk and the user’s read or write buffer. A single **read** or **write** call usually results in a single I/O operation; raw I/O is therefore more efficient when many bytes are transmitted. Block files names are found in **/dev/dsk**; raw file names are found in **/dev/rdsk**.

I/O requests to the raw device must be aligned on a 512-byte (**DEV_BSIZE**) boundary and all I/O request lengths must be in multiples of 512 bytes. Requests that do not meet these requirements will trigger an **EINVAL** error. There are no alignment or length restrictions on I/O requests to the block device.

CD-ROM DRIVE SUPPORT

A CD-ROM disk is single-sided and contains approximately 640 megabytes of data or 74 minutes of audio. When the CD-ROM is opened, the eject button is disabled to prevent manual removal of the disk until the last **close()** is called. No volume label is required for a CD-ROM. The disk geometry and partitioning information are constant and never change. If the CD-ROM contains data recorded in a Solaris-aware file system format, it can be mounted using the appropriate Solaris file system support.

DVD-ROM DRIVE SUPPORT

DVD-ROM media can be single or double-sided and can be recorded upon using a single or double layer structure. Double-layer media provides parallel or opposite track paths. A DVD-ROM can hold from between 4.5 Gbytes and 17 Gbytes of data, depending on the layer structure used for recording and if the DVD-ROM is single or double-sided.

When the DVD-ROM is opened, the eject button is disabled to prevent the manual removal of a disk until the last **close()** is called. No volume label is required for a DVD-ROM. If the DVD-ROM contains data recorded in a Solaris-aware file system format, it can be mounted using the appropriate Solaris file system support.

ZIP/JAZ DRIVE SUPPORT

ZIP/JAZ media provide varied data capacity points; a single **JAZ** drive can store up to 2 GBytes of data, while a ZIP-250 can store up to 250MBytes of data. **ZIP/JAZ** drives can be read-from or written-to using the appropriate drive.

When a **ZIP/JAZ** drive is opened, the eject button is disabled to prevent the manual removal of a disk until the last **close( )** is called. No volume label is required for a **ZIP/JAZ** drive. If the **ZIP/JAZ** drive contains data recorded in a Solaris-aware file system format, it can be mounted using the appropriate Solaris file system support.

DEVICE STATISTICS SUPPORT

Each device maintains I/O statistics for the device and for partitions allocated for that device. For each device/partition, the driver accumulates reads, writes, bytes read, and bytes written. The driver also initiates hi-resolution time stamps at queue entry and exit points to enable monitoring of residence time and cumulative residence-length product for each queue.

Not all device drivers make per-partition IO statistics available for reporting.

IOCTLS

Refer to **dkio**(7I), and **cdio**(7I)

ERRORS

EACCES
Permission denied

EBUSY
The partition was opened exclusively by another thread

EFAULT
The argument features a bad address

EINVAL
Invalid argument

ENOTTY
The device does not support the requested ioctl function

ENXIO
During opening, the device did not exist. During close, the drive unlock failed

EROFS
The device is read-only

EAGAIN
Resource temporarily unavailable

EINTR
A signal was caught during the execution of the **ioctl()** function

ENOMEM
Insufficient memory

EPERM
Insufficent access permission

EIO An I/O error occurred. Refer to notes for details on copy-protected DVD-ROM media.

FILES

/kernel/drv/usb_sd
: 32-bit ELF kernel module for both USB frameworks∗

/kernel/drv/sparcv9/usb_sd
: 64-bit ELF kernel module for both USB frameworks∗

/dev/dsk/cntndnsn
: block files

/dev/rdsk/cntndnsn
: raw files

Where:

cn	controller n
tn	SCSI target id n (0-6)
dn	SCSI LUN n (0-7 normally; some HBAs support LUNs to 15 or 32. See the specific manpage for details)
sn	partition n (0-7)

∗ Please see *http://www.sun.com/desktop/whitepapers.html* for more information regarding USB dual framework implementation, *USBA 1.0*, and *USB 2.0*.

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based SPARC systems
Availability	SUNWusb, SUNWusbx

sar(1), **cfgadm_scsi**(1M), **fdisk**(1M), **format**(1M), **iostat**(1M), **close**(2), **ioctl**(2), **lseek**(2), **read**(2), **write**(2), **scsi**(4), **filesystem**(5), **scsa2usb**(7D), **hsfs**(7FS), **pcfs**(7FS), **udfs**(7FS), **cdio**(7I), **dkio**(7I), **scsi_ifsetcap**(9F), **scsi_reset**(9F)

ANSI Small Computer System Interface-2 (SCSI-2)

ATA Packet Interface for CD-ROMs, SFF-8020i

Mt.Fuji Commands for CD and DVD, SFF8090v3

http://www.sun.com/desktop/whitepapers.html

http://*www.sun.com/io*

DIAGNOSTICS

Error for Command:'*<command name>*'
Error Level: Fatal
Requested Block: *<n>*
Error Block: *<m>*
Vendor:'*<vendorname>*'
Serial Number:'*<serial number>*'
Sense Key:*<sense key name>*

ASC: 0x<a> (<ASC name>), ASCQ: 0x<b>, FRU: 0x<c>
The command indicated by <command name> failed. The Requested Block is the block where the transfer started and the Error Block is the block that caused the error. Sense Key, **ASC**, and **ASCQ** information is returned by the target in response to a request sense command.

Caddy not inserted in drive
The drive is not ready because no caddy has been inserted.

Check Condition on REQUEST SENSE
A REQUEST SENSE command completed with a check condition. The original command will be retried a number of times.

Label says <m> blocks Drive says <n> blocks
There is a discrepancy between the label and what the drive returned on the **READ CAPACITY** command.

Not enough sense information
The request sense data was less than expected.

Request Sense couldn't get sense data
The **REQUEST SENSE** command did not transfer any data.

Reservation Conflict
The drive was reserved by another initiator.

SCSI transport failed: reason 'xxxx': {retrying|giving
up}" 6 The host adapter has failed to transport a command to the target for the reason stated. The driver will either retry the command or, ultimately, give up.

Unhandled Sense Key<*n*>
The REQUEST SENSE data included an invalid sense.

Unit not ready. Additional sense code 0x<n>
The drive is not ready.

Can't do switch back to mode 1
A failure to switch back to read mode 1.

Corrupt label - bad geometry
The disk label is corrupted.

Corrupt label - label checksum failed
The disk label is corrupted.

Corrupt label - wrong magic number
The disk label is corrupted.

Device busy too long
The drive returned busy during a number of retries.

Disk not responding to selection
The drive is powered down or died

Failed to handle UA
A retry on a Unit Attention condition failed.

I/O to invalid geometry
: The geometry of the drive could not be established.

Incomplete read/write - retrying/giving up
: There was a residue after the command completed normally.

No bp for direct access device format geometry
: A bp with consistent memory could not be allocated.

No bp for disk label
: A bp with consistent memory could not be allocated.

No bp for fdisk
: A bp with consistent memory could not be allocated.

No bp for rigid disk geometry
: A bp with consistent memory could not be allocated.

No mem for property
: Free memory pool exhausted.

No memory for direct access device format geometry
: Free memory pool exhausted.

No memory for disk label
: Free memory pool exhausted.

No memory for rigid disk geometry
: The disk label is corrupted.

No resources for dumping
: A packet could not be allocated during dumping.

Offline
: Drive went offline; probably powered down.

Requeue of command fails
: Driver attempted to retry a command and experienced a transport error.

sdrestart transport failed()
: Driver attempted to retry a command and experienced a transport error.

Transfer length not modulo
: Illegal request size.

Transport of request sense fails()
: Driver attempted to submit a request sense command and failed.

Transport rejected()
: Host adapter driver was unable to accept a command.

Unable to read label
: Failure to read disk label.

Unit does not respond to selection
: Drive went offline; probably powered down.

DVD-ROM media containing DVD-Video data may follow/adhere to the requirements of content scrambling system or copy protection scheme. Reading of copy-protected sector will cause I/O error. Users are advised to use the appropriate playback software to view video contents on DVD-ROM media containing DVD-Video data.

NAME

usba – Solaris USB Architecture (USBA)

USB provides a low-cost means for attaching peripheral devices, including mass-storage devices, keyboards, mice, and printers, to a system. For complete information on USB, go to the USB website at *http://www.usb.org*.

USB supports 126 hot-pluggable USB devices per USB bus. The maximum data transfer rate is 12 Mbits per second (Mbps).

The USBA consists of the original USBA framework and a more evolved framework called *USBA 1.0*. The original USBA framework provides compatibility with all drivers which *worked before the current release. The USBA 1.0* framework supports more devices (including *USB 2.0* devices), and offers better performance than the original USBA framework.

In this release, the original USBA framework operates all *USB 1.0* and *USB 1.1* ports *(including on-board ports), and the USB 1.0* framework operates all *USB 2.0* ports (such as PCI *USB 2.0*, or *USB 2.0/1394* combo cards). **prtconf**(1M) with the **-D** option associates devices with drivers whose names begin with "usba10" when the *USBA 1.0* framework services those devices. Please see *www.sun.com/desktop/whitepapers.html* for more information regarding the USB dual framework.

The *USBA 1.0* framework adheres to the *Universal Serial Bus 2.0* specification. The original USBA framework adheres to the *USB 1.1* specification. Both provide a transport layer abstraction to USB client drivers.

FILES

Drivers and modules of the original USB framework are:

FRAMEWORK MODULE
/kernel/misc/[sparcv9]/usba

CLIENT DRIVER	FUNCTION/DEVICE
/kernel/drv/[sparcv9]/hid	HID class
/kernel/drv/[sparcv9]/hubd	hub class
/kernel/drv/[sparcv9]/scsa2usb	mass storage class
/kernel/drv/[sparcv9]/usbprn	printer class
/usr/kernel/drv/[sparcv9]/usb_as	audio streaming
/usr/kernel/drv/[sparcv9]/usb_ac	audio control
/kernel/drv/[sparcv9]/usb_mid	multi-interface device

CLIENT STREAMS MODULES	FUNCTION/DEVICE
/kernel/strmod/[sparcv9]/usbkbm	keyboard
/kernel/strmod/[sparcv9]/usbms	mouse

HOST CONTROLLER INTERFACE DRIVERS	DEVICE
/kernel/drv/[sparcv9]/ohci	Open HCI

Drivers and modules of the USB 1.0 framework are:

FRAMEWORK MODULE
/kernel/misc/[sparcv9]/usba10

CLIENT DRIVER	FUNCTION/DEVICE
/kernel/drv/[sparcv9]/usba10_hid	HID class
/kernel/drv/[sparcv9]/usba10_hubd	hub class
/kernel/drv/[sparcv9]/usba10_scsa2usb	mass storage class
/kernel/drv/[sparcv9]/usba10_usbprn	printer class
/kernel/drv/[sparcv9]/usba10_usb_as	audio streaming
/kernel/drv/[sparcv9]/usba10_usb_ac	audio control
/kernel/drv/[sparcv9]/usba10_/usb_mid	multi-interface device

CLIENT STREAMS MODULES	FUNCTION/DEVICE
/kernel/strmod/[sparcv9]/usbkbm1	keyboard
/kernel/strmod/[sparcv9]/usbms1	mouse
/kernel/strmod/[sparcv9]/usb_ah	audio HID

HOST CONTROLLER INTERFACE DRIVERS	DEVICE
/kernel/drv/[sparcv9]/usba10_ehci	Enhanced HCI
/kernel/drv/[sparcv9]/usba10_ohci	Open HCI

See **attributes**(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	Original USBA drivers and files: PCI-based systems USBA 1.0 drivers and files: PCI-based SPARC systems
Availability	SUNWusb

	SUNWusbx

cfgadm_usb(1M), **attributes**(5), **ehci**(7D), **hid**(7D), **hubd**(7D), **ohci**(7D), **scsa2usb**(7D), **usb_ac**(7D), **usb_ah**(7D), **usb_as**(7D), **usbkbm**(7D), **usb_mid**(7D), **usbms**(7D), **usbprn**(7D)

Universal Serial Bus Specifications 1.1 and 2.0.

System Administration Guide: Basic Administration

http://www.sun.com/desktop/whitepapers.html

http://www.sun.com/io

The messages described below may appear on the system console as well as being logged. All messages are formatted in the following manner:

WARNING: Error message...

<name><number>: obsolete driver:
  usb_pipe_policy is <actual_version> expecting <version>" 6 The driver is using an older revision of USBA. The pipe policy revision used is older and this driver is not supported on the current platform. <*name*><*number*> refer to the driver name and its instance number, respectively.

No driver found for device <device_name>
  (interface <number> node name=<node_name>)" 6 The installed Solaris software does not contain a supported driver for this hardware. <*number*> is the interface number.

No driver found for device <name>.
  The installed Solaris software does not contain a supported driver for this hardware. <*name*> could be the device path name or the device name.

Onlining <path name> failed (<number>).
  The USB device driver could not be brought online due to internal kernel errors. <*number*> is the value returned due to the failure.

The driver for <name> is not for *USBA1.0*
  A device plugged into a port managed by the *USBA1.0* framework has only an original-USBA-framework-compatible driver installed. Please plug into a port managed by the original USBA framework. (Please see the DESCRIPTION section above regarding the dual framework.)

Attempt to corrupt USB list at <address>
  An internal USB data structure is inconsistent. Please reboot the system.

Draining callbacks timed out.
  A USB device or its driver is malfunctioning. Please hot-remove and reconnect the device, or reboot.

NAME

usbkbm – keyboard STREAMS module for Sun USB Keyboard

SYNOPSIS

open("/dev/kbd", O_RDWR)

The **usbkbm STREAMS** module processes byte streams generated by a keyboard attached to a **USB** port. **USB** keyboard is a member of Human Interface Device (HID) Class, and **usbkbm** only supports the keyboard protocol defined in the specification. Definitions for altering keyboard translation and reading events from the keyboard are in <**sys/kbio.h**> and <**sys/kbd.h**>.

The **usbkbm STREAMS** module adheres to the interfaces exported by **kb**(7M). Refer to the **DESCRIPTION** section of **kb**(7M) for a discussion of the keyboard translation modes and the **IOCTL** section for the supported **ioctl**(2) requests.

USB Keyboard **usbkbm** returns different values for the following ioctls than **kb**(7M):

KIOCTYPE

This **ioctl()** returns a new keyboard type defined for the **USB** keyboard. All types are listed below:

```
KB_SUN3   Sun Type 3 keyboard
KB_SUN4   Sun Type 4 keyboard
KB_ASCII  ASCII terminal masquerading as keyboard
KB_PC     Type 101 PC keyboard
KB_USB    USB keyboard
```

The **USB** keyboard type is **KB_USB**; **usbkbm** will return **KB_USB** in response to the **KIOCTYPE** ioctl.

KIOCLAYOUT

The argument is a pointer to an **int**. The layout code specified by the **bCountryCode** value returned in the **HID** descriptor is returned in the int pointed to by the argument. The **countrycodes** are defined in 6.2.1 of the **HID** 1.0 specification.

KIOCCMD

KBD_CMD_CLICK/KBD_CMD_NOCLICK

The **kb**(7M) indicates that inappropriate commands for particular keyboards are ignored. Because clicking is not supported on the **USB** keyboard, **usbkbm** ignores this command

KBD_CMD_SETLED

Set keyboard LEDs. Same as **kb**(7M).

KBD_CMD_GETLAYOUT

The country codes defined in 6.2.1 of the **HID** 1.0 specification are returned.

KBD_CMD_BELL/KBD_CMD_NOBELL

This command is supported although the **USB** keyboard does not have a buzzer. The request for the bell is rerouted.

KBD_CMD_RESET
There is no notion of resetting the keyboard as there is for the type4 keyboard. **usbkbm** ignores this command and does not return an error.

/kernel/strmod/usbkbm
32 bit ELF kernel module for original USBA framework∗

/kernel/strmod/sparcv9/usbkbm
64 bit ELF kernel module for original USBA framework∗

/kernel/strmod/usbkb1
32 bit ELF kernel module for USBA 1.0 framework∗

/kernel/strmod/sparcv9/usbkb1
64 bit ELF kernel module for USBA 1.0 framework∗

∗ Please see *www.sun.com/desktop/whitepapers.html* for more information regarding USB dual framework implementation, *USBA 1.0,* and *USB 2.0.*

See **attributes**(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	Original USBA drivers and files: PCI-based systems USBA 1.0 drivers and files: PCI-based SPARC systems
Availability	SUNWusb, SUNWusbx

dumpkeys(1), **kbd**(1), **loadkeys**(1), **ioctl**(2), **keytables**(4), **attributes**(5), **hid**(7D), **usba**(7D), **termio(7I)**, **kb**(7M)

System Administration Guide: Basic Administration

http://www.sun.com/desktop/whitepapers.html

http://www.sun.com/io

DIAGN0STICS

None

NAME

usbms – USB mouse STREAMS module

SYNOPSIS

#include <sys/vuid_event.h>

#include <sys/msio.h>

#include <sys/msreg.h>

The **usbms STREAMS** module processes byte streams generated by a **USB** mouse. A **USB** mouse is a member of the Human Interface Device (HID) class and the **usbms** module supports only the mouse boot protocol defined in the **HID** specification. The **usbms** module must be pushed on top of the **HID** class driver (see **hid**(7D)). In the **VUID_FIRM_EVENT** mode, the **usbms** module translates packets from the **USB** mouse into Firm events. The Firm event structure is defined in <**sys/vuid_event.h**>. The **STREAMS** module state is initially set to raw or **VUID_NATIVE** mode which performs no message processing. See the *HID 1.0* specification for the raw format of the mouse packets. To initiate mouse protocol conversion to Firm events, change the state to **VUID_FIRM_EVENT**.

VUIDGFORMAT
This option returns the current state of the **STREAMS** module. The state of the **usbms STREAMS** module may be either **VUID_NATIVE** (no message processing) or **VUID_FIRM_EVENT** (convert to Firm events).

VUIDSFORMAT
The argument is a pointer to an **int**. Set the state of the **STREAMS** module to the **int** pointed to by the argument.

```
typedef struct  vuid_addr_probe {
     short base; /* default vuid device addr directed too */
     union {
        short next;   /* next addr for default when VUIDSADDR */
        short current; /* current addr of default when VUIDGADDR */
     } data;
} Vuid_addr_probe;
```

VUIDSADDR
The argument is a pointer to a **Vuid_addr_probe** structure. **VUIDSADDR** sets the virtual input device segment address indicated by base to next.

If base does not equal **VKEY_FIRST**, **ENODEV** is returned.

VUIDGADDR
The argument is a pointer to a **Vuid_addr_probe** structure. Return the address of the virtual input device segment indicated by base to current.

If base does not equal **VKEY_FIRST**, **ENODEV** is returned.

ioctl() requests for changing and retrieving mouse parameters use the **Ms_parms** structure:

```
typedef struct {
      int     jitter_thresh;
      int     speed_law;
      int     speed_limit;
} Ms_parms;
```

jitter_thresh is the "jitter threshold" of the mouse. Motions fewer than **jitter_thresh** units along both axes are accumulated and then sent up the stream after 1/12 second.

speed_law indicates whether extremely large motions are to be ignored. If it is **1,** a "speed limit" is applied to mouse motions. Motions along either axis of more than **speed_limit** units are discarded.

MSIOGETPARMS
The argument is a pointer to a **Ms_params** structure. The **usbms** module parameters are returned in the structure.

MSIOSETPARMS
The argument is a pointer to a **Ms_params** structure. The **usbms** module parameters are set according to the values in the structure.

/kernel/strmod/usbms
32 bit ELF kernel module for original USBA framework*

/kernel/strmod/sparcv9/usbms
64 bit ELF kernel module for original USBA framework*

/kernel/strmod/usbms1
32 bit ELF kernel module for USBA 1.0 framework*

/kernel/strmod/sparcv9/usbms1
64 bit ELF kernel module for USBA 1.0 framework*

* Please see *www.sun.com/desktop/whitepapers.html* for more information regarding USB dual framework implementation, *USBA 1.0*, and *USB 2.0*.

See **attributes**(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	Original USBA drivers and files: PCI-based systems USBA 1.0 drivers and files: PCI-based SPARC systems
Availability	SUNWusb, SUNWusbx

ioctl(2), **attributes**(5), **hid**(7D), **usba**(7D)

System Administration Guide: Basic Administration

http://www.sun.com/desktop/whitepapers.html

http://www/sun.com/io

DIAGNOSTICS

None

NAME

usbprn – USB printer class driver

SYNOPSIS

#include <sys/usb/clients/printer/usb_printer.h>

#include <sys/ecppio.h>

usbprn@unit-address

The **usbprn** driver is a USBA (Solaris USB Architecture) compliant client driver that supports the *USB Printer Class 1.0* specification. The **usbprn** driver supports a subset of the **ecpp**(7D) parallel port driver functionality. However, unlike the STREAMS-based **ecpp** driver, **usbprn** is a character driver.

The **usbprn** driver supports all USB printer-class compliant printers. For a list of recommended printers and USB parallel printer adapters, visit http://*www.sun.com/io*.

The **usbprn** driver supports non-PostScript printers that utilize third-party PostScript conversion packages such as GhostScript. Conversion packages can be obtained from the Solaris Software companion CD, available at *http://www.sun.com/software/solaris/binaries/package.html*.

DEFAULT OPERATION

With certain minor exceptions (outlined in the Notes sections below), the **usbprn** driver supports a subset of the **ecpp**(7D) ioctl interfaces:

Configuration variables are set to their default values each time the USB printer device is attached. The **write_timeout** period (defined in the ECPPIOC_SETPARMS ioctl description below) is set to 90 seconds. The mode is set to centronics mode (ECPP_CENTRONICS). Parameters can be changed through the ECPPIOC_SETPARMS ioctl and read through the ECPPIOC_GETPARMS ioctl. Each time the USB printer device is opened, the device is marked as busy and all further opens will return EBUSY. Once the device is open, applications can write to the device and the driver can send data and obtain device id and status.

> **Note:**
>
> Unlike the ecpp(7D) driver, **usbprn** resets configuration variables to their default values with each **attach**(9E). (The **ecpp**(7D) driver resets configuration variables with each **open**(2).)

WRITE OPERATION

A **write**(2) operation returns the number of bytes successfully written to the device. If a failure occurs while a driver is transferring data to printer, the contents of the status bits are captured at the time of the error and can be retrieved by the application program using the ECPPIOC_GETERR **ioctl**(2) call. The captured status information is overwritten each time an ECPPIOC_TESTIO **ioctl**(2) occurs.

The **usbprn** driver supports **prnio(7I)** interfaces. Note that the **PRNIOC_RESET** command has no effect on USB printers.

The following **ioctl**(2) calls are supported for backward compatibility and are not recommended for new applications.

ECPPIOC_GETPARMS

Gets current transfer parameters. The argument is a pointer to **struct** ecpp_transfer_parms. If parameters are not configured after the device is opened, the structure will be set to its default configuration.

Note:

Unlike the ecpp(7D) driver, only the ECPP_CENTRONICS mode is currently supported in **usbprn**.

ECPPIOC_SETPARMS

Sets transfer parameters. The argument is a pointer to a **struct** ecpp_transfer_parms. If a parameter is out of range, **EINVAL** is returned. If the peripheral or host device cannot support the requested mode, **EPROTONOSUPPORT** is returned.

The transfer parameters structure is defined in <**sys/ecppio.h**>:

```
struct ecpp_transfer_parms {
    int  write_timeout;
    int  mode;
};
```

The **write_timeout** field, which specifies how long the driver will take to transfer 8192 bytes of data to the device, is set to a default value of 90 seconds. The **write_timeout** field must be greater than one second and less than 300 seconds (five minutes.)

Note:

Unlike the ecpp(7D) driver, only the ECPP_CENTRONICS mode is currently supported in **usbprn**. Also, the semantics of **write_timeout** in **usbprn** differ from **ecpp**(7D). Refer to **ecpp**(7D) for information.

BPPIOC_TESTIO

Tests the transfer readiness of a print device and checks status bits to determine if a **write**(2) will succeed. If status bits are set, a transfer will fail. If a transfer will succeed, zero is returned. If a transfer fails, the driver returns **EIO** and the state of the status bits are captured. The captured status can be retrieved using the BPPIOC_GETERR **ioctl**(2) call. BPPIOC_TESTIO and BPPIOC_GETERR are compatible to the ioctls specified in **bpp**(7D).

Note:

Unlike the ecpp(7D) driver, only the ECPP_CENTRONICS mode is currently supported in **usbprn**. Additionally, **bus_error** and **timeout_occurred** fields are not used in the **usbprn** interface. (In **ecpp**(7D), **timeout_occurred** is used.)

BPPIOC_GETERR

Get last error status. The argument is a pointer to a **struct bpp_error_status**. This structure indicates the status of all the appropriate status bits at the time of the most recent error condition during a **write**(2) call, or the status of the bits at the most recent BPPIOC_TESTIO **ioctl**(2) call.

```
struct bpp_error_status {
   char     timeout_occurred; /* not used */
   char     bus_error;          /* not used */
   uchar_t pin_status;         /* status of pins which
                            /* could cause error */
};
```

The pin_status field indicates possible error conditions. The error status structure **bpp_error_status** is defined in the include file <**sys/bpp_io.h**>. The valid bits for **pin_status** can be **BPP_ERR_ERR**, **BPP_SLCT_ERR**, and **BPP_PE_ERR**. A set bit indicates that the associated pin is asserted.

Note:

Unlike the ecpp(7D) driver, only the ECPP_CENTRONICS mode is currently supported in **usbprn**. Additionally, the **bus_error** and **timeout_occurred** fields are not used in the **usbprn** interface. (In **ecpp**(7D), **timeout_occurred** is used.) Unlike **ecpp**(7D), the BPP_BUSY_ERR status bit is not supported by USB printers.

ECPPIOC_GETDEVID

Gets the IEEE 1284 device ID from the peripheral. The argument is a pointer to a **struct ecpp_device_id**. Applications should set mode to ECPP_CENTRONICS. If another mode is used, the driver will return **EPROTONOSUPPORT**. **len** is the length of the buffer pointed to by **addr**. **rlen** is the actual length of the device ID string returned from the peripheral. If the returned **rlen** is greater than **len**, the application should call ECPPIOC_GETDEVID a second time with a buffer length equal to **rlen**.

The 1284 device ID structure:

```
struct ecpp_device_id {
  int  mode; /* mode to use for reading device id */
  int  len;  /* length of buffer */
  int  rlen;  /* actual length of device id string */
  char *addr; /* buffer address */
```

Note:

Unlike ecpp(7D), only the ECPP_CENTRONICS mode is currently supported in **usbprn**.

READ OPERATION

The **read** operation is not supported and returns **EIO**.

ERRORS

EBUSY

The device has been opened and another open is attempted. An attempt has been made to unload the driver while one of the units is open.

EINVAL

An unsupported IOCTL has been received. A ECPPIOC_SETPARMS **ioctl**(2) is attempted with an out of range value in the **ecpp_transfer_parms** structure.

EIO The driver has received an unrecoverable device error, or the device is not responding, or the device has stalled when attempting an access. A **write**(2) or **ioctl**(2) did not complete due to a peripheral access. A **read**(2) system call has been issued.

ENXIO
The driver has received an **open**(2) request for a unit for which the attach failed.

ENODEV
The driver has received an **open**(2) request for a device that has been disconnected.

EPROTONOSUPPORT
The driver has received a ECPPIOC_SETPARMS **ioctl**(2) for a mode argument other than ECPP_CENTRONICS in the **ecpp_transfer_parms** structure.

/kernel/drv/usbprn
32 bit ELF kernel module for original USBA framework∗

/kernel/drv/sparcv9/usbprn
64 bit ELF kernel module for original USBA framework∗

/kernel/drv/usba10_usbprn
32 bit ELF kernel module for USBA 1.0 framework∗

/kernel/drv/sparcv9/usba10_usbprn
64 bit ELF kernel module for USBA 1.0 framework∗

/kernel/drv/usba10_usbprn.conf
usba10_usbprn configuration file

/dev/printers/n
Character special files

∗ Please see *www.sun.com/desktop/whitepapers.html* for more information regarding USB dual framework implementation, *USBA 1.0*, and *USB 2.0*.

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	Original USBA drivers and files: PCI-based systems USBA 1.0 drivers and files: PCI-based SPARC systems
Availability	SUNWusb, SUNWusbx

cfgadm_usb(1M), **printmgr**(1M), **ioctl**(2), **open**(2), **read**(2), **write**(2), **attributes**(5), **bpp**(7D), **ecpp**(7D), **usba**(7D), **prnio**(7I), **attach**(9E)

Writing Device Drivers

Universal Serial Bus Specification 2.0

USB Device Class Definition for Printing Devices 1.0

System Administration Guide: Basic Administration

http://www.sun.com/desktop/whitepapers.html

http://www.sun.com/io

DIAGNOSTICS

In addition to being logged, the following messages may appear on the system console. All messages are formatted in the following manner:

Warning: <device path> usbprn<instance num>: Error Message...

or

Warning: <device path> usba10_usbprn<instance num>:
Error Message...

Device was disconnected while open. Data may have been
lost." 6 The device has been hot-removed or powered off while it was open and a possible data transfer was in progress. The job may be aborted.

Cannot access device. Please reconnect <device name>.
There was an error in accessing the printer during reconnect. Please reconnect the device.

Device is not identical to the previous one on this port.
Please disconnect and reconnect." 6 A USB printer was hot-removed while open. A new device was hot-inserted which is not identical to the original USB printer. Please disconnect the new USB device and reconnect the original printer to the same port.

Device has been reconnected, but data may have been lost.
The printer that was hot-removed from its USB port has been re-inserted again to the same port. It is available for access but the job that was running prior to the hot-removal may be lost.

NOTES

The USB printer will be power managed if the device is closed.

If a printer is hot-removed before a job completes, the job is terminated and the driver will return EIO. All subsequent opens will return **ENODEV**. If a printer is hot-removed, an LP reconfiguration may not be needed if a printer is re-inserted on the same port. If re-inserted on a different port, an LP reconfiguration may be required.

The USB Parallel Printer Adapter is not hotpluggable. The printer should be connected to USB Parallel Printer Adapter before plugging the USB cable into host or hub port and should be removed only after disconnecting the USB cable of USB Parallel Printer Adapter from the host or hub port.

NAME

usbser_edge – Digi Edgeport USB to serial converter driver

SYNOPSIS

#include <fcntl.h>

#include <sys/termios.h>

usbser_edge@unit

The **usbser_edge** driver is a loadable STREAMS and USBA (Solaris USB Architecture) compliant client driver which provides basic asynchronous communication support for Digi Edgeport USB-to-serial converters. Supported devices include Edgeport/2, Edgeport/21, Edgeport/4, Edgeport/421, Edgeport/8 and Edgeport/416. Serial device streams are built with appropriate modules that are pushed atop the **usbser_edge** driver by the **autopush**(1M) facility.

The **usbser_edge** module supports the **termio**(7I) device control functions specified by flags in the **c_cflag** word of the **termios** structure, and by the **IGNBRK**, **IGNPAR**, **PARMRK** and **INPCK** flags in the **c_iflag** word of the **termios** structure. All other **termio**(7I) functions must be performed by STREAMS modules pushed atop the driver. When a device is opened, the **ldterm**(7M) and **ttcompat**(7M) STREAMS modules are automatically pushed on top of the stream, providing the standard **termio**(7I) interface.

Use device logical names **/dev/term/[0-9]***∗* to access the serial ports. These names are typically used to provide a logical access point for a dial-in line that is used with a modem.

To allow a single tty line to be connected to a modem and used for incoming and outgoing calls, a special feature is available that is controlled by the minor device number. By accessing through device logical name**/dev/cua/[0-9]***∗*, it is possible to open a port without the Carrier Detect signal being asserted, either through hardware or an equivalent software mechanism. These devices are commonly known as dial-out lines.

Unlike onboard serial ports, the **usbser_edge** ports cannot serve as a local serial console.

APPLICATION PROGRAMMING INTERFACE

A dial-in line can be opened only if the corresponding dial-out line is closed. A blocking **/dev/term** open waits until the **/dev/cua** line is closed (which drops Data Terminal Ready, after which Carrier Detect usually drops as well) and carrier is detected again. A non-blocking **/dev/term** open returns an error if the **/dev/cua** is open.

If the **/dev/term** line is opened successfully (usually only when carrier is recognized on the modem), the corresponding **/dev/cua** line cannot be opened. This allows a modem and port to be used for dial-in (by enabling the line for login in **/etc/inittab**) or dial-out (by **tip**(1), or **uucp**(1C)) when no one is logged in on the line.

Device hot-removal is functionally equivalent to modem disconnect event, as defined in **termio**(7I).

IOCTLS

The **usbser_edge** driver supports the standard set of **termio**(7I) ioctl commands.

Input and output line speeds can be set to the following baud rates: 0, 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200, or 230400. Input and output line speeds cannot be set independently; for example, when the output speed is set, the input speed is automatically set to the same speed.

ERRORS

An **open()** fails under the following conditions:

ENXIO
The unit being opened does not exist.

EBUSY
The **/dev/cua** (dial-out) device is being opened while the/dev/term (dial-in device) is open, or the dial-in device is being opened with a no-delay open while the dial-out device is open.

EBUSY
The unit has been marked as exclusive-use by another process with a **TIOCEXCL ioctl()** call.

EIO USB device I/O error.

/kernel/drv/usbser_edge
32 bit ELF kernel module for original USBA framework∗

/kernel/drv/sparcv9/usbser_edge
64 bit ELF kernel module for original USBA framework∗

/kernel/drv/usba10_usbser_edge
32 bit ELF kernel module for USBA 1.0 framework∗

/kernel/drv/sparcv9/usba10_usbser_edge
64 bit ELF kernel module for USBA 1.0 framework∗

/kernel/drv/usba10_usbser_edge.conf
usba10_usbser_edge configuration file

/dev/cua/[0-9]∗
dial-out tty lines

/dev/term/[0-9]∗
dial-in tty lines

∗ Please see *www.sun.com/desktop/whitepapers.html* for more information regarding USB dual framework implementation, *USBA 1.0*, and *USB 2.0*.

See **attributes**(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	Original USBA drivers and files: PCI-based systems USBA 1.0 drivers and files: PCI-based SPARC systems
Availability	SUNWusb, SUNWusbx

strconf(1), **tip**(1), **uucp**(1C), **autopush**(1M), **ioctl**(2), **open**(2), **termios**(3C), **attributes**(5), **usba**(7D), **termio**(7I), **ldterm**(7M), **ttcompat**(7M)

http://www.sun.com/desktop/whitepapers.html

DIAGNOSTICS

In addition to being logged, the following messages may appear on the system console. All messages are formatted in the following manner:

Warning: <device path> usbser_edge<instance num>: Error Message...

or

Warning: <device path> usba10_usbser_edge<instance num>:
Error Message...

Device was disconnected while open. Data may have been
: lost. " 6 The device was hot-removed or powered off while it was open and a possible data transfer was in progress.

Device is not identical to the previous one on this port.
: Please disconnect and reconnect." 6 The USB device was hot-removed while open. A new device was hot-inserted which is not identical to the original device. Please disconnect the new device and reconnect the original device to the same port.

Device has been reconnected, but data may have been lost.
: The device that was hot-removed from its USB port has been re-inserted again to the same port. It is available for access but data from a previous transfer may be lost.

Cannot access device. Please reconnect <name>.
: The device was hot-removed and has not been reconnected. Please reconnect the device.