man pages section 3: Extended Library Functions, Volume 4

Beta

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

Part No: 819–2248–30
December 2009

091223@23031

Contents

Preface

Both novice users and those familar with the SunOS operating system can use online man pages to obtain information about the system and its features. A man page is intended to answer concisely the question "What does it do?" The man pages in general comprise a reference manual. They are not intended to be a tutorial.

Overview

The following contains a brief description of each man page section and the information it references:

- Section 1 describes, in alphabetical order, commands available with the operating system.
- Section 1M describes, in alphabetical order, commands that are used chiefly for system maintenance and administration purposes.
- Section 2 describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value.
- Section 3 describes functions found in various libraries, other than those functions that directly invoke UNIX system primitives, which are described in Section 2.
- Section 4 outlines the formats of various files. The C structure declarations for the file formats are given where applicable.
- Section 5 contains miscellaneous documentation such as character-set tables.
- Section 7 describes various special files that refer to specific hardware peripherals and device drivers. STREAMS software drivers, modules and the STREAMS-generic set of system calls are also described.
- Section 9E describes the DDI (Device Driver Interface)/DKI (Driver/Kernel Interface), DDI-only, and DKI-only entry-point routines a developer can include in a device driver.
- Section 9F describes the kernel functions available for use by device drivers.
- Section 9S describes the data structures used by drivers to share information between the driver and the kernel.

Below is a generic format for man pages. The man pages of each manual section generally follow this order, but include only needed headings. For example, if there are no bugs to report,

there is no BUGS section. See the `intro` pages for more information and detail about each section, and `man(1)` for more information about man pages in general.

NAME — This section gives the names of the commands or functions documented, followed by a brief description of what they do.

SYNOPSIS — This section shows the syntax of commands or functions. When a command or file does not exist in the standard path, its full path name is shown. Options and arguments are alphabetized, with single letter arguments first, and options with arguments next, unless a different argument order is required.

The following special characters are used in this section:

- [ ] — Brackets. The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument must be specified.
- . . . — Ellipses. Several values can be provided for the previous argument, or the previous argument can be specified multiple times, for example, "`filename...`".
- | — Separator. Only one of the arguments separated by this character can be specified at a time.
- { } — Braces. The options and/or arguments enclosed within braces are interdependent, such that everything enclosed must be treated as a unit.

PROTOCOL — This section occurs only in subsection 3R to indicate the protocol description file.

DESCRIPTION — This section defines the functionality and behavior of the service. Thus it describes concisely what the command does. It does not discuss OPTIONS or cite EXAMPLES. Interactive commands, subcommands, requests, macros, and functions are described under USAGE.

IOCTL — This section appears on pages in Section 7 only. Only the device class that supplies appropriate parameters to the `ioctl(2)` system call is called `ioctl` and generates its own heading. `ioctl` calls for a specific device are listed alphabetically (on the man page for that specific device).

	`ioctl` calls are used for a particular class of devices all of which have an `io` ending, such as `mtio(7I)`.
OPTIONS	This secton lists the command options with a concise summary of what each option does. The options are listed literally and in the order they appear in the SYNOPSIS section. Possible arguments to options are discussed under the option, and where appropriate, default values are supplied.
OPERANDS	This section lists the command operands and describes how they affect the actions of the command.
OUTPUT	This section describes the output – standard output, standard error, or output files – generated by the command.
RETURN VALUES	If the man page documents functions that return values, this section lists these values and describes the conditions under which they are returned. If a function can return only constant values, such as 0 or –1, these values are listed in tagged paragraphs. Otherwise, a single paragraph describes the return values of each function. Functions declared void do not return values, so they are not discussed in RETURN VALUES.
ERRORS	On failure, most functions place an error code in the global variable `errno` indicating why they failed. This section lists alphabetically all error codes a function can generate and describes the conditions that cause each error. When more than one condition can cause the same error, each condition is described in a separate paragraph under the error code.
USAGE	This section lists special rules, features, and commands that require in-depth explanations. The subsections listed here are used to explain built-in functionality: Commands Modifiers Variables Expressions Input Grammar
EXAMPLES	This section provides examples of usage or of how to use a command or function. Wherever possible a complete

	example including command-line entry and machine response is shown. Whenever an example is given, the prompt is shown as `example%`, or if the user must be superuser, `example#`. Examples are followed by explanations, variable substitution rules, or returned values. Most examples illustrate concepts from the SYNOPSIS, DESCRIPTION, OPTIONS, and USAGE sections.
ENVIRONMENT VARIABLES	This section lists any environment variables that the command or function affects, followed by a brief description of the effect.
EXIT STATUS	This section lists the values the command returns to the calling program or shell and the conditions that cause these values to be returned. Usually, zero is returned for successful completion, and values other than zero for various error conditions.
FILES	This section lists all file names referred to by the man page, files of interest, and files created or required by commands. Each is followed by a descriptive summary or explanation.
ATTRIBUTES	This section lists characteristics of commands, utilities, and device drivers by defining the attribute type and its corresponding value. See `attributes(5)` for more information.
SEE ALSO	This section lists references to other man pages, in-house documentation, and outside publications.
DIAGNOSTICS	This section lists diagnostic messages with a brief explanation of the condition causing the error.
WARNINGS	This section lists warnings about special conditions which could seriously affect your working conditions. This is not a list of diagnostics.
NOTES	This section lists additional information that does not belong anywhere else on the page. It takes the form of an aside to the user, covering points of special interest. Critical information is never covered here.
BUGS	This section describes known bugs and, wherever possible, suggests workarounds.

REFERENCE

Extended Library Functions, Volume 4

Name acl_check – check the validity of an ACL

Synopsis

```
cc [ flag... ] file... -lsec [ library... ]
#include <sys/acl.h>

int acl_check(acl_t *aclp, int isdir);
```

Description The `acl_check()` function checks the validity of an ACL pointed to by *aclp*. The *isdir* argument checks the validity of an ACL that will be applied to a directory. The ACL can be either a POSIX draft ACL as supported by UFS or NFSv4 ACL as supported by ZFS or NFSV4.

When the function verifies a POSIX draft ACL, the rules followed are described in `aclcheck(3SEC)`. For NFSv4 ACL, the ACL is verified against the following rules:

- The inheritance flags are valid.
- The ACL must have at least one ACL entry and no more than {`MAX_ACL_ENTRIES`}.
- The permission field contains only supported permissions.
- The entry type is valid.
- The flag fields contain only valid flags as supported by NFSv4/ZFS.

If any of the above rules are violated, the function fails with `errno` set to `EINVAL`.

Return Values If the ACL is valid, `acl_check()` returns 0. Otherwise `errno` is set to `EINVAL` and the return value is set to one of the following:

`EACL_INHERIT_ERROR`	There are invalid inheritance flags specified.
`EACL_FLAGS_ERROR`	There are invalid flags specified on the ACL that don't map to supported flags in NFSV4/ZFS ACL model.
`EACL_ENTRY_ERROR`	The ACL contains an unknown value in the type field.
`EACL_MEM_ERROR`	The system cannot allocate any memory.
`EACL_INHERIT_NOTDIR`	Inheritance flags are only allowed for ACLs on directories.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also `acl(2)`, `aclcheck(3SEC)`, `aclsort(3SEC)`, `acl(5)`, `attributes(5)`

Name aclcheck – check the validity of an ACL

Synopsis

```
cc [ flag... ] file... -lsec [ library... ]
#include <sys/acl.h>

int aclcheck(aclent_t *aclbufp, int nentries, int *which);
```

Description The `aclcheck()` function checks the validity of an ACL pointed to by *aclbufp*. The *nentries* argument is the number of entries contained in the buffer. The *which* parameter returns the index of the first entry that is invalid.

The function verifies that an ACL pointed to by *aclbufp* is valid according to the following rules:

- There must be exactly one `GROUP_OBJ` ACL entry.
- There must be exactly one `USER_OBJ` ACL entry.
- There must be exactly one `OTHER_OBJ` ACL entry.
- If there are any `GROUP` ACL entries, then the group ID in each group ACL entry must be unique.
- If there are any `USER` ACL entries, then the user ID in each user ACL entry must be unique.
- If there are any `GROUP` or `USER` ACL entries, then there must be exactly one `CLASS_OBJ` (ACL mask) entry.
- If there are any default ACL entries, then the following apply:
  - There must be exactly one default `GROUP_OBJ` ACL entry.
  - There must be exactly one default `OTHER_OBJ` ACL entry.
  - There must be exactly one default `USER_OBJ` ACL entry.
  - If there are any `DEF_GROUP` entries, then the group ID in each `DEF_GROUP` ACL entry must be unique.
  - If there are any `DEF_USER` entries, then the user ID in each `DEF_USER` ACL entry must be unique.
  - If there are any `DEF_GROUP` or `DEF_USER` entries, then there must be exactly one `DEF_CLASS_OBJ` (default ACL mask) entry.
- If any of the above rules are violated, then the function fails with `errno` set to `EINVAL`.

Return Values If the ACL is valid, `alcheck()` will return `0`. Otherwise `errno` is set to `EINVAL` and return code is set to one of the following:

`GRP_ERROR`	There is more than one `GROUP_OBJ` or `DEF_GROUP_OBJ` ACL entry.
`USER_ERROR`	There is more than one `USER_OBJ` or `DEF_USER_OBJ` ACL entry.
`CLASS_ERROR`	There is more than one `CLASS_OBJ` (ACL mask) or `DEF_CLASS_OBJ` (default ACL mask) entry.

OTHER_ERROR	There is more than one OTHER_OBJ or DEF_OTHER_OBJ ACL entry.
DUPLICATE_ERROR	Duplicate entries of USER, GROUP, DEF_USER, or DEF_GROUP.
ENTRY_ERROR	The entry type is invalid.
MISS_ERROR	Missing an entry. The *which* parameter returns –1 in this case.
MEM_ERROR	The system cannot allocate any memory. The which parameter returns –1 in this case.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Unsafe

See Also acl(2), aclsort(3SEC), attributes(5)

Name acl_free – free memory associated with an acl_t structure

Synopsis

```
cc [ flag... ] file... -lsec [ library... ]
#include <sys/acl.h>

void acl_free(acl_t *aclp);
```

Description The `acl_free()` function frees memory allocated for the acl_t structure pointed to by the *aclp* argument.

Return Values The `acl_free()` function does not return a value.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also `acl_get(3SEC)`, `acl(5)`, `attributes(5)`

Name acl_get, facl_get, acl_set, facl_set – get or set a file's Access Control List (ACL)

Synopsis

```
cc [ flag... ] file... -lsec [ library... ]
#include <sys/acl.h>

int *acl_get(const char *path, int flag, acl_t **aclp);

int *facl_get(int fd, int flag, acl_t **aclp);

int acl_set(const char *path, acl_t *aclp);

int facl_set(int fd, acl_t *aclp);
```

Description The `acl_get()` and `facl_get()` functions retrieve an Access Control List (ACL) of a file whose name is given by *path* or referenced by the open file descriptor *fd*. The *flag* argument specifies whether a trivial ACL should be retrieved. When the *flag* argument is `ACL_NO_TRIVIAL`, only ACLs that are not trivial will be retrieved. The ACL is returned in the *aclp* argument.

The `acl_set()` and `facl_set()` functions are used for setting an ACL of a file whose name is given by *path* or referenced by the open file descriptor *fd*. The *aclp* argument specifies the ACL to set.

The `acl_get()` and `acl_set()` functions support multiple types of ACLs. When possible, the `acl_set()` function translates an ACL to the target file's style of ACL. Currently this is only possible when translating from a POSIX-draft ACL such as on UFS to a file system that supports NFSv4 ACL semantics such as ZFS or NFSv4.

Return Values Upon successful completion, `acl_get()` and `facl_get()` return 0 and *aclp* is non-`NULL`. The *aclp* argument can be `NULL` after successful completion if the file had a trivial ACL and the *flag* argument was `ACL_NO_TRIVIAL`. Otherwise, -1 is returned and `errno` is set to indicate the error.

Upon successful completion, `acl_set()` and `facl_set()` return 0. Otherwise, -1 is returned and `errno` is set to indicate the error.

Errors These functions will fail if:

`EACCES` The caller does not have access to a component of *path*.

`EIO` A disk I/O error has occured while retrieving the ACL.

`ENOENT` A component of the *path* does not exist.

`ENOSYS` The file system does not support ACLs.

`ENOTSUP` The ACL supplied could not be translated to an NFSv4 ACL.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also chmod(1), acl(2), acl(5), attributes(5)

Name aclsort – sort an ACL

Synopsis

```
cc [ flag ... ] file ... -lsec [ library ... ]
#include <sys/acl.h>

int aclsort(int nentries, int calclass, aclent_t *aclbufp);
```

Description The *aclbufp* argument points to a buffer containing ACL entries. The *nentries* argument specifies the number of ACL entries in the buffer. The *calclass* argument, if non-zero, indicates that the `CLASS_OBJ` (ACL mask) permissions should be recalculated. The union of the permission bits associated with all ACL entries in the buffer other than `CLASS_OBJ`, `OTHER_OBJ`, and `USER_OBJ` is calculated. The result is copied to the permission bits associated with the `CLASS_OBJ` entry.

The `aclsort()` function sorts the contents of the ACL buffer as follows:

- Entries will be in the order `USER_OBJ`, `USER`, `GROUP_OBJ`, `GROUP`, `CLASS_OBJ` (ACL mask), `OTHER_OBJ`, `DEF_USER_OBJ`, `DEF_USER`, `DEF_GROUP_OBJ`, `DEF_GROUP`, `DEF_CLASS_OBJ` (default ACL mask), and `DEF_OTHER_OBJ`.
- Entries of type `USER`, `GROUP`, `DEF_USER`, and `DEF_GROUP` will be sorted in increasing order by ID.

The `aclsort()` function will succeed if all of the following are true:

- There is exactly one entry each of type `USER_OBJ`, `GROUP_OBJ`, `CLASS_OBJ` (ACL mask), and `OTHER_OBJ`.
- There is exactly one entry each of type `DEF_USER_OBJ`, `DEF_GROUP_OBJ`, `DEF_CLASS_OBJ` (default ACL mask), and `DEF_OTHER_OBJ` if there are any default entries.
- Entries of type `USER`, `GROUP`, `DEF_USER`, or `DEF_GROUP` may not contain duplicate entries. A duplicate entry is one of the same type containing the same numeric ID.

Return Values Upon successful completion, the function returns `0`. Otherwise, it returns `−1`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Unsafe

See Also `acl(2)`, `aclcheck(3SEC)`, `attributes(5)`

Name acl_strip – remove all ACLs from a file

Synopsis

```
cc [ flag... ] file... -lsec [ library... ]
#include <sys/acl.h>

int acl_strip(const char *path, uid_t uid, gid_t gid, mode_t mode);
```

Description The `acl_strip()` function removes all ACLs from a file and replaces them with a trivial ACL based on the *mode* argument. After replacing the ACL, the owner and group of the file are set to the values specified by the *uid* and *gid* arguments.

Return Values Upon successful completion, `acl_strip()` returns 0. Otherwise it returns –1 and sets `errno` to indicate the error.

Errors The `acl_strip()` function will fail if:

`EACCES`	Search permission is denied on a component of the path prefix of *path*.
`EFAULT`	The *path* argument points to an illegal address.
`EINVAL`	The *uid* or *gid* argument is out of range.
`EIO`	A disk I/O error has occurred while storing or retrieving the ACL.
`ELOOP`	A loop exists in symbolic links encountered during the resolution of the *path* argument.
`ENAMETOOLONG`	The length of the *path* argument exceeds {`PATH_MAX`}, or the length of a path component exceeds {`NAME_MAX`} while `_POSIX_NO_TRUNC` is in effect.
`ENOENT`	A component of *path* does not exist.
`ENOTDIR`	A component of the prefix of *path* is not a directory.
`EPERM`	The effective user ID does not match the owner of the file and the process does not have appropriate privileges.
`EROFS`	The file system is mounted read-only.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also `acl_get(3SEC)`, `acl_trivial(3SEC)`, `acl(5)`, `attributes(5)`

Name acltomode, aclfrommode – convert an ACL to or from permission bits

Synopsis

```
cc [ flag... ] file... -lsec [ library... ]
#include <sys/types.h>
#include <sys/acl.h>

int acltomode(aclent_t *aclbufp, int nentries, mode_t *modep);

int aclfrommode(aclent_t *aclbufp, int nentries, mode_t *modep);
```

Description The `acltomode()` function converts an ACL pointed to by *aclbufp* into the permission bits buffer pointed to by *modep*. If the `USER_OBJ` ACL entry, `GROUP_OBJ` ACL entry, or the `OTHER_OBJ` ACL entry cannot be found in the ACL buffer, then the function fails with `errno` set to `EINVAL`.

The `USER_OBJ` ACL entry permission bits are copied to the file owner class bits in the permission bits buffer. The `OTHER_OBJ` ACL entry permission bits are copied to the file other class bits in the permission bits buffer. If there is a `CLASS_OBJ` (ACL mask) entry, the `CLASS_OBJ` ACL entry permission bits are copied to the file group class bits in the permission bits buffer. Otherwise, the `GROUP_OBJ` ACL entry permission bits are copied to the file group class bits in the permission bits buffer.

The `aclfrommode()` function converts the permission bits pointed to by *modep* into an ACL pointed to by *aclbufp*. If the `USER_OBJ` ACL entry, `GROUP_OBJ` ACL entry, or the `OTHER_OBJ` ACL entry cannot be found in the ACL buffer, the function fails with `errno` set to `EINVAL`.

The file owner class bits from the permission bits buffer are copied to the `USER_OBJ` ACL entry. The file other class bits from the permission bits buffer are copied to the `OTHER_OBJ` ACL entry. If there is a `CLASS_OBJ` (ACL mask) entry, the file group class bits from the permission bits buffer are copied to the `CLASS_OBJ` ACL entry, and the `GROUP_OBJ` ACL entry is not modified. Otherwise, the file group class bits from the permission bits buffer are copied to the `GROUP_OBJ` ACL entry.

The *nentries* argument represents the number of ACL entries in the buffer pointed to by *aclbufp*.

Return Values Upon successful completion, the function returns `0`. Otherwise, it returns `−1` and sets `errno` to indicate the error.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also acl(2), attributes(5)

Name acl_totext, acl_fromtext – convert internal representation to or from external representation

Synopsis

```
cc [ flag... ] file... -lsec [ library... ]
#include <sys/acl.h>

char *acl_totext(acl_t *aclp, int flags);

int acl_fromtext(char *acltextp, acl_t **aclp);
```

Description The `acl_totext()` function converts an internal ACL representation pointed to by *aclp* into an external ACL representation. The memory for the external text string is obtained using `malloc(3C)`. The caller is responsible for freeing the memory upon completion.

The format of the external ACL is controlled by the *flags* argument. Values for *flags* are constructed by a bitwise-inclusive-OR of *flags* from the following list, defined in `<sys/acl.h>`.

`ACL_COMPACT_FMT` For NFSv4 ACLs, the ACL entries will be formatted using the compact ACL format detailed in `ls(1)` for the `-V` option.

`ACL_APPEND_ID` Append the `uid` or `gid` for additional user or group entries. This flag is used to construt ACL entries in a manner that is suitable for archive utilities such as `tar(1)`. When the ACL is translated from the external format to internal representation using `acl_fromtext()`, the appended ID will be used to populate the `uid` or `gid` field of the ACL entry when the user or group name does not exist on the host system. The appended id will be ignored when the user or group name does exist on the system.

`ACL_SID_FMT` For NFSv4 ACLs, the ACL entries for user or group entries will use the `usersid` or `groupsid` format when the "id" field in the ACL entry is an ephemeral `uid` or `gid`. The raw `sid` format will only be used when the "id" cannot be resolved to a windows name.

The `acl_fromtext()` function converts an external ACL representation pointed to by *acltextp* into an internal ACL representation. The memory for the list of ACL entries is obtained using `malloc(3C)`. The caller is responsible for freeing the memory upon completion. Depending on type of ACLs a file system supports, one of two external external representations are possible. For POSIX draft file systems such as ufs, the external representation is described in `acltotext(3SEC)`. The external ACL representation For NFSv4–style ACLs is detailed as follows.

Each `acl_entry` contains one ACL entry. The external representation of an ACL entry contains three, four or five colon separated fields. The first field contains the ACL entry type. The entry type keywords are defined as:

`everyone@` This ACL entry specifies the access granted to any user or group that does not match any previous ACL entry.

`group` This ACL entry with a GID specifies the access granted to a additional group of the object.

`group@` This ACL entry with no GID specified in the ACL entry field specifies the access granted to the owning group of the object.

`groupsid` This ACL entry with a SID or Windows name specifies the access granted to a Windows group. This type of entry is for a CIFS server created file.

`owner@` This ACL entry with no UID specified in the ACL entry field specifies the access granted to the owner of the object.

`sid` This ACL entry with a SID or Windows name when the entry could be either a group or a user.

`user` This ACL entry with a UID specifies the access granted to a additional user of the object.

`usersid` This ACL entry with a SID or Windows name specifies the access granted to a Windows user. This type of entry is for a CIFS server created file.

The second field contains the ACL entry ID, and is used only for user or group ACL entries. This field is not used for `owner@`, `group@`, or `everyone@` entries.

`uid` This field contains a user-name or user-ID. If the user-name cannot be resolved to a UID, then the entry is assumed to be a numeric UID.

`gid` This field contains a group-name or group-ID. If the group-name can't be resolved to a GID, then the entry is assumed to be a numeric GID.

The third field contains the discretionary access permissions. The format of the permissions depends on whether `ACL_COMPACT_FMT` is specified. When the *flags* field does not request `ACL_COMPACT_FMT`, the following format is used with a forward slash (/) separating the permissions.

`add_file` Add a file to a directory.

`add_subdirectory` Add a subdirectory.

`append` Append data.

`delete` Delete.

`delete_child` Delete child.

`execute` Execute permission.

`list_directory` List a directory.

`read_acl` Read ACL.

`read_data` Read permission.

`read_attributes`	Read attributes.
`read_xattr`	Read named attributes.
`synchronize`	Synchronize.
`write_acl`	Write ACL.
`write_attributes`	Write attributes.
`write_data`	Write permission.
`write_owner`	Write owner.
`write_xattr`	Write named attributes.

This format allows permissions to be specified as, for example: `read_data/read_xattr/read_attributes`.

When `ACL_COMPACT_FMT` is specified, the permissions consist of 14 unique letters. A hyphen (-) character is used to indicate that the permission at that position is not specified.

`a`	read attributes
`A`	write attributes
`c`	read ACL
`C`	write ACL
`d`	delete
`D`	delete child
`o`	write owner
`p`	append
`r`	read_data
`R`	read named attributes
`s`	synchronize
`w`	write_data
`W`	write named attributes
`x`	execute

This format allows compact permissions to be represented as, for example: `rw--d-a-------`

The fourth field is optional when `ACL_COMPACT_FMT` is not specified, in which case the field will be present only when the ACL entry has inheritance flags set. The following is the list of inheritance flags separated by a slash (/) character.

dir_inherit	ACE_DIRECTORY_INHERIT_ACE
file_inherit	ACE_FILE_INHERIT_ACE
inherit_only	ACE_INHERIT_ONLY_ACE
no_propagate	ACE_NO_PROPAGATE_INHERIT_ACE

When ACL_COMPACT_FMT is specified the inheritance will always be present and is represented as positional arguments. A hyphen (-) character is used to indicate that the inheritance flag at that position is not specified.

d	dir_inherit
f	file_inherit
F	failed access (not currently supported)
i	inherit_only
n	no_propagate
S	successful access (not currently supported)

The fifth field contains the type of the ACE (allow or deny):

allow	The mask specified in field three should be allowed.
deny	The mask specified in field three should be denied.

Return Values Upon successful completion, the acl_totext() function returns a pointer to a text string. Otherwise, it returns NULL.

Upon successful completion, the acl_fromtext() function returns 0. Otherwise, the return value is set to one of the following:

EACL_FIELD_NOT_BLANK	A field that should be blank is not blank.
EACL_FLAGS_ERROR	An invalid ACL flag was specified.
EACL_INHERIT_ERROR	An invalid inheritance field was specified.
EACL_INVALID_ACCESS_TYPE	An invalid access type was specified.
EACL_INVALID_STR	The string is NULL.
EACL_INVALID_USER_GROUP	The required user or group name not found.
EACL_MISSING_FIELDS	The ACL needs more fields to be specified.
EACL_PERM_MASK_ERROR	The permission mask is invalid.
EACL_UNKNOWN_DATA	Unknown data was found in the ACL.

Examples

EXAMPLE 1 Examples of permissions when ACL_COMPACT_FMT is not specified.

```
user:joe:read_data/write_data:file_inherit/dir_inherit:allow

owner@:read_acl:allow,user:tom:read_data:file_inherit/inherit_only:deny
```

EXAMPLE 2 Examples of permissions when ACL_COMPACT_FMT is specified.

```
user:joe:rw------------:fd----:allow

owner@:----------c---:------allow,user:tom:r-------------:f-i---:deny
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also

ls(1), tar(1), acl(2), malloc(3C), aclfromtext(3SEC), acl(5), attributes(5)

Name acltotext, aclfromtext – convert internal representation to or from external representation

Synopsis

```
cc [ flag... ] file... -lsec [ library... ]
#include <sys/acl.h>

char *acltotext(aclent_t *aclbufp, int aclcnt);

aclent_t *aclfromtext(char *acltextp, int *aclcnt);
```

Description The `acltotext()` function converts an internal ACL representation pointed to by *aclbufp* into an external ACL representation. The space for the external text string is obtained using `malloc(3C)`. The caller is responsible for freeing the space upon completion..

The `aclfromtext()` function converts an external ACL representation pointed to by *acltextp* into an internal ACL representation. The space for the list of ACL entries is obtained using `malloc(3C)`. The caller is responsible for freeing the space upon completion. The *aclcnt* argument indicates the number of ACL entries found.

An external ACL representation is defined as follows:

<acl_entry>[,<acl_entry>] . . .

Each <acl_entry> contains one ACL entry. The external representation of an ACL entry contains two or three colon-separated fields. The first field contains the ACL entry tag type. The entry type keywords are defined as:

`user`	This ACL entry with no UID specified in the ACL entry ID field specifies the access granted to the owner of the object. Otherwise, this ACL entry specifies the access granted to a specific user-name or user-id number.
`group`	This ACL entry with no GID specified in the ACL entry ID field specifies the access granted to the owning group of the object. Otherwise, this ACL entry specifies the access granted to a specific group-name or group-id number.
`other`	This ACL entry specifies the access granted to any user or group that does not match any other ACL entry.
`mask`	This ACL entry specifies the maximum access granted to user or group entries.
`default:user`	This ACL entry with no uid specified in the ACL entry ID field specifies the default access granted to the owner of the object. Otherwise, this ACL entry specifies the default access granted to a specific user-name or user-ID number.
`default:group`	This ACL entry with no gid specified in the ACL entry ID field specifies the default access granted to the owning group of the object. Otherwise, this ACL entry specifies the default access granted to a specific group-name or group-ID number.

`default:other` This ACL entry specifies the default access for other entry.

`default:mask` This ACL entry specifies the default access for mask entry.

The second field contains the ACL entry ID, as follows:

`uid` This field specifies a user-name, or user-ID if there is no user-name associated with the user-ID number.

`gid` This field specifies a group-name, or group-ID if there is no group-name associated with the group-ID number.

`empty` This field is used by the user and group ACL entry types.

The third field contains the following symbolic discretionary access permissions:

`r` read permission

`w` write permission

`x` execute/search permission

– no access

Return Values Upon successful completion, the `acltotext()` function returns a pointer to a text string. Otherwise, it returns `NULL`.

Upon successful completion, the `aclfromtext()` function returns a pointer to a list of ACL entries. Otherwise, it returns `NULL`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Unsafe

See Also `acl(2)`, `malloc(3C)`, `attributes(5)`

Name acl_trivial – determine whether a file has a trivial ACL

Synopsis

```
cc [ flag... ] file... -lsec [ library... ]
#include <sys/acl.h>

int acl_trivial(char *path);
```

Description The `acl_trivial()` function is used to determine whether a file has a trivial ACL. Whether an ACL is trivial depends on the type of the ACL. A POSIX draft ACL is trivial if it has greater than `MIN_ACL_ENTRIES`. An NFSv4/ZFS-style ACL is trivial if it either has entries other than `owner@`, `group@`, and `everyone@`, has inheritance flags set, or is not ordered in a manner that meets POSIX access control requirements.

Return Values Upon successful completion, `acl_trivial()` returns 0 if the file's ACL is trivial and 1 if the file's ACL is not trivial. If it could not be determined whether a file's ACL is trivial, -1 is returned and `errno` is set to indicate the error.

Errors The `acl_trivial()` function will fail if:

`EACCES` A file's ACL could not be read.

`ENOENT` A component of *path* does not name an existing file or *path* is an empty string.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also `acl(5)`, `attributes(5)`

Name blcompare, blequal, bldominates, blstrictdom, blinrange – compare binary labels

Synopsis

```
cc [flag...] file... -ltsol [library...]
#include <tsol/label.h>

int blequal(const m_label_t *label1, const m_label_t *label2);

int bldominates(const m_label_t *label1, const m_label_t *label2);

int blstrictdom(const m_label_t *label1, const m_label_t *label2);

int blinrange(const m_label_t *label, const brange_t *range);
```

Description These functions compare binary labels for meeting a particular condition.

The `blequal()` function compares two labels for equality.

The `bldominates()` function compares label *label1* for dominance over label *label2*.

The `blstrictdom()` function compares label *label1* for strict dominance over label *label2*.

The `blinrange()` function compares label *label* for dominance over *range→lower_bound* and *range→upper_bound* for dominance over level *label*.

Return Values These functions return non-zero if their respective conditions are met, otherwise zero is returned.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also `getplabel(3TSOL)`, `label_to_str(3TSOL)`, `libtsol(3LIB)`, `ucred_getlabel(3C)`, `label_encodings(4)`, `attributes(5)`, `labels(5)`

"Determining the Relationship Between Two Labels" in *Solaris Trusted Extensions Developer's Guide*

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name blminmax, blmaximum, blminimum – bound of two labels

Synopsis

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

#include <tsol/label.h>

void blmaximum(m_label_t *maximum_label,
     const m_label_t *bounding_label);

void blminimum(m_label_t *minimum_label,
     const m_label_t *bounding_label);
```

Description The `blmaximum()` function replaces the contents of label *maximum_label* with the least upper bound of the labels *maximum_label* and *bounding_label*. The least upper bound is the greater of the classifications and all of the compartments of the two labels. This is the least label that dominates both of the original labels.

The `blminimum()` function replaces the contents of label *minimum_label* with the greatest lower bound of the labels *minimum_label* and *bounding_label*. The greatest lower bound is the lower of the classifications and only the compartments that are contained in both labels. This is the greatest label that is dominated by both of the original labels.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also `label_to_str(3TSOL)`, `libtsol(3LIB)`, `sbltos(3TSOL)`, `attributes(5)`

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name bltocolor, bltocolor_r – get character-coded color name of label

Synopsis

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

#include <tsol/label.h>

char *bltocolor(const m_label_t *label);

char *bltocolor_r(const m_label_t *label, const int size,
     char *color_name);
```

Description The `bltocolor()` and `bltocolor_r()` functions get the character-coded color name associated with the binary label *label*.

The calling process must have `PRIV_SYS_TRANS_LABEL` in its set of effective privileges to get color names of labels that dominate the current process's sensitivity label.

Return Values The `bltocolor()` function returns a pointer to a statically allocated string that contains the character-coded color name specified for the *label* or returns `(char *)0` if, for any reason, no character-coded color name is available for this binary label.

The `bltocolor_r()` function returns a pointer to the *color_name* string which contains the character-coded color name specified for the *label* or returns `(char *)0` if, for any reason, no character-coded color name is available for this binary label. *color_name* must provide for a string of at least *size* characters.

Files `/etc/security/tsol/label_encodings`
The label encodings file contains the classification names, words, constraints, and values for the defined labels of this system.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete
MT-Level	MT-Safe with exceptions

These functions are obsolete and retained for ease of porting. They might be removed in a future Solaris Trusted Extensions release. Use the `label_to_str(3TSOL)` function instead.

The `bltocolor()` function returns a pointer to a statically allocated string. Subsequent calls to it will overwrite that string with a new character-coded color name. It is not MT-Safe. The `bltocolor_r()` function should be used in multithreaded applications.

See Also `label_to_str(3TSOL)`, `libtsol(3LIB)`, `attributes(5)`

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

If *label* includes a specified word or words, the character-coded color name associated with the first word specified in the label encodings file is returned. Otherwise, if no character-coded color name is specified for *label*, the first character-coded color name specified in the label encodings file with the same classification as the binary label is returned.

Name bltos, bsltos, bcleartos – translate binary labels to character coded labels

Synopsis

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

#include <tsol/label.h>

int bsltos(const m_label_t *label, char **string,
     const int str_len, const int flags);

int bcleartos(const m_label_t *label, char **string,
     const int str_len, const int flags);
```

Description These functions translate binary labels into strings controlled by the value of the *flags* parameter.

The `bsltos()` function translates a binary sensitivity label into a string. The applicable *flags* are `LONG_CLASSIFICATION` or `SHORT_CLASSIFICATION`, `LONG_WORDS` or `SHORT_WORDS`, `VIEW_EXTERNAL` or `VIEW_INTERNAL`, and `NO_CLASSIFICATION`. A *flags* value `0` is equivalent to (`SHORT_CLASSIFICATION` | `LONG_WORDS`).

The `bcleartos()` function translates a binary clearance into a string. The applicable *flags* are `LONG_CLASSIFICATION` or `SHORT_CLASSIFICATION`, `LONG_WORDS` or `SHORT_WORDS`, `VIEW_EXTERNAL` or `VIEW_INTERNAL`, and `NO_CLASSIFICATION`. A *flags* value 0 is equivalent to (`SHORT_CLASSIFICATION` | `LONG_WORDS`). The translation of a clearance might not be the same as the translation of a sensitivity label. These functions use different `label_encodings` file tables that might contain different words and constraints.

The calling process must have `PRIV_SYS_TRANS_LABEL` in its set of effective privileges to perform label translation on labels that dominate the current process's sensitivity label.

The generic form of an output character-coded label is:

```
CLASSIFICATION WORD1 WORD2 WORD3/WORD4 SUFFIX PREFIX WORD5/WORD6
```

Capital letters are used to display all `CLASSIFICATION` names and `WORD`s. The ' ' (space) character separates classifications and words from other words in all character-coded labels except where multiple words that require the same `PREFIX` or `SUFFIX` are present, in which case the multiple words are separated from each other by the '`/`' (slash) character.

The *string* argument can point to either a pointer to pre-allocated memory, or the value `(char *)0`. If *string* points to a pointer to pre-allocated memory, then *str_len* indicates the size of that memory. If *string* points to the value `(char *)0`, memory is allocated using `malloc()` to contain the translated character-coded labels. The translated *label* is copied into allocated or pre-allocated memory.

The *flags* argument is `0` or the logical sum of the following:

`LONG_WORDS` Translate using long names of words defined in *label*.

SHORT_WORDS	Translate using short names of words defined in *label*. If no short name is defined in the label_encodings file for a word, the long name is used.
LONG_CLASSIFICATION	Translate using long name of classification defined in *label*.
SHORT_CLASSIFICATION	Translate using short name of classification defined in *label*.
ACCESS_RELATED	Translate only *access-related* entries defined in information label *label*.
VIEW_EXTERNAL	Translate ADMIN_LOW and ADMIN_HIGH labels to the lowest and highest labels defined in the label_encodings file.
VIEW_INTERNAL	Translate ADMIN_LOW and ADMIN_HIGH labels to the admin low name and admin high name strings specified in the label_encodings file. If no strings are specified, the strings "ADMIN_LOW" and "ADMIN_HIGH" are used.
NO_CLASSIFICATION	Do not translate classification defined in *label*.

Process Attributes

If the VIEW_EXTERNAL or VIEW_INTERNAL flags are not specified, translation of ADMIN_LOW and ADMIN_HIGH labels is controlled by the label view process attribute flags. If no label view process attribute flags are defined, their translation is controlled by the label view configured in the label_encodings file. A value of External specifies that ADMIN_LOW and ADMIN_HIGH labels are mapped to the lowest and highest labels defined in the label_encodings file. A value of Internal specifies that the ADMIN_LOW and ADMIN_HIGH labels are translated to the admin low and admin high name strings specified in the label_encodings file. If no such names are specified, the strings "ADMIN_LOW" and "ADMIN_HIGH" are used.

Return Values

Upon successful completion, the bsltos() and bcleartos() functions return the length of the character-coded label, including the NULL terminator.

If the label is not of the valid defined required type, if the label is not dominated by the process sensitivity label and the process does not have PRIV_SYS_TRANS_LABEL in its set of effective privileges, or if the label_encodings file is inaccessible, these functions return −1.

If memory cannot be allocated for the return string or if the pre-allocated return string memory is insufficient to hold the string, these functions return 0. The value of the pre-allocated string is set to the NULL string (*string[0]='\\00';).

Files

/etc/security/tsol/label_encodings
: The label encodings file contains the classification names, words, constraints, and values for the defined labels of this system.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete
MT-Level	MT-Safe with exceptions

The `bsltos()` and `bcleartos()` functions are Obsolete. Use the `label_to_str`(3TSOL) function instead.

See Also `free`(3C), `label_to_str`(3TSOL), `libtsol`(3LIB), `malloc`(3C), `label_encodings`(4), `attributes`(5)

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

If memory is allocated by these functions, the caller must free the memory with `free`(3C) when the memory is no longer in use.

Name btohex, bsltoh, bcleartoh, bsltoh_r, bcleartoh_r, h_alloc, h_free – convert binary label to hexadecimal

Synopsis

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

#include <tsol/label.h>

char *bsltoh(const m_label_t *label);

char *bcleartoh(const m_label_t *clearance);

char *bsltoh_r(const m_label_t *label, char *hex);

char *bcleartoh_r(const m_label_t *clearance, char *hex);

char *h_alloc(const unsigned char type);

void h_free(char *hex);
```

Description These functions convert binary labels into hexadecimal strings that represent the internal value.

The `bsltoh()` and `bsltoh_r()` functions convert a binary sensitivity label into a string of the form:

`[0x`*sensitivity_label_hexadecimal_value*`]`

The `bcleartoh()` and `bcleartoh_r()` functions convert a binary clearance into a string of the form:

`0x`*clearance_hexadecimal_value*

The `h_alloc()` function allocates memory for the hexadecimal value *type* for use by `bsltoh_r()` and `bcleartoh_r()`.

Valid values for *type* are:

`SUN_SL_ID` *label* is a binary sensitivity label.

`SUN_CLR_ID` *label* is a binary clearance.

The `h_free()` function frees memory allocated by `h_alloc()`.

Return Values These functions return a pointer to a string that contains the result of the translation, or `(char *)0` if the parameter is not of the required type.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe with exceptions

The `bsltoh()`, `bcleartoh()`, `bsltoh_r()`, `bcleartoh_r()`, `h_alloc()`, and `h_free()` functions are Obsolete. Use the `label_to_str`(3TSOL) function instead.

The `bsltoh()` and `bcleartoh()` functions share the same statically allocated string storage. They are not MT-Safe. Subsequent calls to any of these functions will overwrite that string with the newly translated string. The `bsltoh_r()` and `bcleartoh_r()` functions should be used in multithreaded applications.

See Also `atohexlabel`(1M), `hextoalabel`(1M), `label_to_str`(3TSOL), `libtsol`(3LIB), `attributes`(5), `labels`(5)

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name cpl_complete_word, cfc_file_start, cfc_literal_escapes, cfc_set_check_fn, cpl_add_completion, cpl_file_completions, cpl_last_error, cpl_list_completions, cpl_recall_matches, cpl_record_error, del_CplFileConf, cpl_check_exe, del_WordCompletion, new_CplFileConf, new_WordCompletion – look up possible completions for a word

Synopsis

```
cc [ flag... ] file... -ltecla [ library... ]
#include <stdio.h>
#include <libtecla.h>

WordCompletion *new_WordCompletion(void);

WordCompletion *del_WordCompletion(WordCompletion *cpl);

CPL_MATCH_FN(cpl_file_completions);

CplFileConf *new_CplFileConf(void);

void cfc_file_start((CplFileConf *cfc, int start_index);

void cfc_literal_escapes(CplFileConf *cfc, int literal);

void cfc_set_check_fn(CplFileConf *cfc, CplCheckFn *chk_fn,
     void *chk_data);

CPL_CHECK_FN(cpl_check_exe);

CplFileConf *del_CplFileConf(CplFileConf *cfc);

CplMatches *cpl_complete_word(WordCompletion *cpl, const char *line,
     int word_end, void *data, CplMatchFn *match_fn);

CplMatches *cpl_recall_matches(WordCompletion *cpl);

int cpl_list_completions(CplMatches *result, FILE *fp, int term_width);

int cpl_add_completion(WordCompletion *cpl, const char *line,
     int word_start, int word_end, const char *suffix,
     const char *type_suffix, const char *cont_suffix);

void cpl_record_error(WordCompletion *cpl, const char *errmsg);

const char *cpl_last_error(WordCompletion *cpl);
```

Description The `cpl_complete_word()` function is part of the `libtecla(3LIB)` library. It is usually called behind the scenes by `gl_get_line(3TECLA)`, but can also be called separately.

Given an input line containing an incomplete word to be completed, it calls a user-provided callback function (or the provided file-completion callback function) to look up all possible completion suffixes for that word. The callback function is expected to look backward in the line, starting from the specified cursor position, to find the start of the word to be completed, then to look up all possible completions of that word and record them, one at a time, by calling `cpl_add_completion()`.

The new_WordCompletion() function creates the resources used by the cpl_complete_word() function. In particular, it maintains the memory that is used to return the results of calling cpl_complete_word().

The del_WordCompletion() function deletes the resources that were returned by a previous call to new_WordCompletion(). It always returns NULL (that is, a deleted object). It takes no action if the *cpl* argument is NULL.

The callback functions that look up possible completions should be defined with the CPL_MATCH_FN() macro, which is defined in <libtecla.h>. Functions of this type are called by cpl_complete_word(), and all of the arguments of the callback are those that were passed to said function. In particular, the *line* argument contains the input line containing the word to be completed, and *word_end* is the index of the character that follows the last character of the incomplete word within this string. The callback is expected to look backwards from *word_end* for the start of the incomplete word. What constitutes the start of a word clearly depends on the application, so it makes sense for the callback to take on this responsibility. For example, the builtin filename completion function looks backwards until it encounters an unescaped space or the start of the line. Having found the start of the word, the callback should then lookup all possible completions of this word, and record each completion with separate calls to cpl_add_completion(). If the callback needs access to an application-specific symbol table, it can pass it and any other data that it needs using the *data* argument. This removes any need for global variables.

The callback function should return 0 if no errors occur. On failure it should return 1 and register a terse description of the error by calling cpl_record_error().

The last error message recorded by calling cpl_record_error() can subsequently be queried by calling cpl_last_error().

The cpl_add_completion() function is called zero or more times by the completion callback function to record each possible completion in the specified WordCompletion object. These completions are subsequently returned by cpl_complete_word(). The *cpl*, *line*, and *word_end* arguments should be those that were passed to the callback function. The *word_start* argument should be the index within the input line string of the start of the word that is being completed. This should equal *word_end* if a zero-length string is being completed. The *suffix* argument is the string that would have to be appended to the incomplete word to complete it. If this needs any quoting (for example, the addition of backslashes before special charaters) to be valid within the displayed input line, this should be included. A copy of the suffix string is allocated internally, so there is no need to maintain your copy of the string after cpl_add_completion() returns.

In the array of possible completions that the cpl_complete_word() function returns, the suffix recorded by cpl_add_completion() is listed along with the concatentation of this suffix with the word that lies between *word_start* and *word_end* in the input line.

The *type_suffix* argument specifies an optional string to be appended to the completion if it is displayed as part of a list of completions by *cpl_list_completions*. The intention is that this indicate to the user the type of each completion. For example, the file completion function places a directory separator after completions that are directories, to indicate their nature to the user. Similary, if the completion were a function, you could indicate this to the user by setting *type_suffix* to "()". Note that the *type_suffix* string is not copied, so if the argument is not a literal string between speech marks, be sure that the string remains valid for at least as long as the results of `cpl_complete_word()` are needed.

The *cont_suffix* argument is a continuation suffix to append to the completed word in the input line if this is the only completion. This is something that is not part of the completion itself, but that gives the user an indication about how they might continue to extend the token. For example, the file-completion callback function adds a directory separator if the completed word is a directory. If the completed word were a function name, you could similarly aid the user by arranging for an open parenthesis to be appended.

The `cpl_complete_word()` is normally called behind the scenes by `gl_get_line(3TECLA)`, but can also be called separately if you separately allocate a `WordCompletion` object. It performs word completion, as described at the beginning of this section. Its first argument is a resource object previously returned by `new_WordCompletion()`. The *line* argument is the input line string, containing the word to be completed. The *word_end* argument contains the index of the character in the input line, that just follows the last character of the word to be completed. When called by `gl_get_line()`, this is the character over which the user pressed TAB. The *match_fn* argument is the function pointer of the callback function which will lookup possible completions of the word, as described above, and the *data* argument provides a way for the application to pass arbitrary data to the callback function.

If no errors occur, the `cpl_complete_word()` function returns a pointer to a `CplMatches` container, as defined below. This container is allocated as part of the *cpl* object that was passed to `cpl_complete_word()`, and will thus change on each call which uses the same *cpl* argument.

```
typedef struct {
    char *completion;        /* A matching completion */
                             /* string */
    char *suffix;            /* The part of the */
                             /* completion string which */
                             /* would have to be */
                             /* appended to complete the */
                             /* original word. */
    const char *type_suffix; /* A suffix to be added when */
                             /* listing completions, to */
                             /* indicate the type of the */
                             /* completion. */
} CplMatch;

typedef struct {
```

```
    char *suffix;            /* The common initial part */
                             /* of all of the completion */
                             /* suffixes. */
    const char *cont_suffix; /* Optional continuation */
                             /* string to be appended to */
                             /* the sole completion when */
                             /* nmatch==1. */
    CplMatch *matches;       /* The array of possible */
                             /* completion strings, */
                             /* sorted into lexical */
                             /* order. */
    int nmatch;              /* The number of elements in */
                             /* the above matches[] */
                             /* array. */
} CplMatches;
```

If an error occurs during completion, `cpl_complete_word()` returns `NULL`. A description of the error can be acquired by calling the `cpl_last_error()` function.

The `cpl_last_error()` function returns a terse description of the error which occurred on the last call to `cpl_com plete_word()` or `cpl_add_completion()`.

As a convenience, the return value of the last call to `cpl_complete_word()` can be recalled at a later time by calling `cpl_recall_matches()`. If `cpl_complete_word()` returned `NULL`, so will `cpl_recall_matches()`.

When the `cpl_complete_word()` function returns multiple possible completions, the `cpl_list_completions()` function can be called upon to list them, suitably arranged across the available width of the terminal. It arranges for the displayed columns of completions to all have the same width, set by the longest completion. It also appends the *type_suffix* strings that were recorded with each completion, thus indicating their types to the user.

Builtin Filename completion Callback

By default the `gl_get_line()` function, passes the `CPL_MATCH_FN(cps_file_completions)` completion callback function to `cpl_complete_word()`. This function can also be used separately, either by sending it to `cpl_complete_word()`, or by calling it directly from your own completion callback function.

```
#define CPL_MATCH_FN(fn) int (fn)(WordCompletion *cpl, \
                                  void *data, const char *line, \
                                  int word_end)

typedef CPL_MATCH_FN(CplMatchFn);

CPL_MATCH_FN(cpl_file_completions);
```

Certain aspects of the behavior of this callback can be changed via its *data* argument. If you are happy with its default behavior you can pass `NULL` in this argument. Otherwise it should be a pointer to a `CplFileConf` object, previously allocated by calling `new_CplFileConf()`.

CplFileConf objects encapsulate the configuration parameters of cpl_file_completions(). These parameters, which start out with default values, can be changed by calling the accessor functions described below.

By default, the cpl_file_completions() callback function searches backwards for the start of the filename being completed, looking for the first unescaped space or the start of the input line. If you wish to specify a different location, call cfc_file_start() with the index at which the filename starts in the input line. Passing *start_index*=-1 reenables the default behavior.

By default, when cpl_file_completions() looks at a filename in the input line, each lone backslash in the input line is interpreted as being a special character which removes any special significance of the character which follows it, such as a space which should be taken as part of the filename rather than delimiting the start of the filename. These backslashes are thus ignored while looking for completions, and subsequently added before spaces, tabs and literal back slashes in the list of completions. To have unescaped back slashes treated as normal characters, call cfc_literal_escapes() with a non-zero value in its *literal* argument.

By default, cpl_file_completions() reports all files whose names start with the prefix that is being completed. If you only want a selected subset of these files to be reported in the list of completions, you can arrange this by providing a callback function which takes the full pathname of a file, and returns 0 if the file should be ignored, or 1 if the file should be included in the list of completions. To register such a function for use by cpl_file_completions(), call cfc_set_check_fn(), and pass it a pointer to the function, together with a pointer to any data that you would like passed to this callback whenever it is called. Your callback can make its decisions based on any property of the file, such as the filename itself, whether the file is readable, writable or executable, or even based on what the file contains.

```
#define CPL_CHECK_FN(fn) int (fn)(void *data, \
                                  const char *pathname)

typedef CPL_CHECK_FN(CplCheckFn);

void cfc_set_check_fn(CplFileConf *cfc, CplCheckFn *chk_fn, \
                                        void *chk_data);
```

The cpl_check_exe() function is a provided callback of the above type, for use with cpl_file_completions(). It returns non-zero if the filename that it is given represents a normal file that the user has execute permission to. You could use this to have cpl_file_completions() only list completions of executable files.

When you have finished with a CplFileConf variable, you can pass it to the del_CplFileConf() destructor function to reclaim its memory.

Thread Safety

It is safe to use the facilities of this module in multiple threads, provided that each thread uses a separately allocated WordCompletion object. In other words, if two threads want to do word completion, they should each call new_WordCompletion() to allocate their own completion objects.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also ef_expand_file(3TECLA), gl_get_line(3TECLA), libtecla(3LIB), pca_lookup_file(3TECLA), attributes(5)

Name ef_expand_file, del_ExpandFile, ef_last_error, ef_list_expansions, new_ExpandFile – expand filename and wildcard expressions

Synopsis

```
cc [ flag... ] file... -ltecla [ library... ]
#include <libtecla.h>

ExpandFile *ef_expand_file(void);

ExpandFile *del_ExpandFile(ExpandFile *ef);

FileExpansion *ef_last_error(ExpandFile *ef, const char *path,
     int pathlen);

int ef_list_expansions(FileExpansion *result, FILE *fp, int term_width);

const char *new_ExpandFile(ExpandFile *ef);
```

Description The `ef_expand_file()` function is part of the `libtecla(3LIB)` library. It expands a specified filename, converting ~user/ and ~/ expressions at the start of the filename to the corresponding home directories, replacing $`envvar` with the value of the corresponding environment variable, and then, if there are any wildcards, matching these against existing filenames. Backslashes in the input filename are interpreted as escaping any special meanings of the characters that follow them. Only backslashes that are themselves preceded by backslashes are preserved in the expanded filename.

In the presence of wildcards, the returned list of filenames includes only the names of existing files which match the wildcards. Otherwise, the original filename is returned after expansion of tilde and dollar expressions, and the result is not checked against existing files. This mimics the file-globbing behavior of the UNIX `tcsh` shell.

The supported wildcards and their meanings are:

*	Match any sequence of zero or more characters.
?	Match any single character.
[*chars*]	Match any single character that appears in *chars*. If *chars* contains an expression of the form a-b, then any character between a and b, including a and b, matches. The '-' character loses its special meaning as a range specifier when it appears at the start of the sequence of characters. The ']' character also looses its significance as the terminator of the range expression if it appears immediately after the opening '[', at which point it is treated one of the characters of the range. If you want both '-' and ']' to be part of the range, the '-' should come first and the ']' second.
[^*chars*]	The same as [*chars*] except that it matches any single character that does not appear in *chars*.

Note that wildcards never match the initial dot in filenames that start with '.'. The initial '.' must be explicitly specified in the filename. This again mimics the globbing behavior of most

UNIX shells, and its rational is based in the fact that in UNIX, files with names that start with '.' are usually hidden configuration files, which are not listed by default by the `ls(1)` command.

The `new_ExpandFile()` function creates the resources used by the `ef_expand_file()` function. In particular, it maintains the memory that is used to record the array of matching file names that is returned by `ef_expand_file()`. This array is expanded as needed, so there is no builtin limit to the number of files that can be matched.

The `del_ExpandFile()` function deletes the resources that were returned by a previous call to `new_ExpandFile()`. It always returns NULL (that is, a deleted object). It does nothing if the *ef* argument is NULL.

The `ef_expand_file()` function performs filename expansion. Its first argument is a resource object returned by `new_ExpandFile()`. A pointer to the start of the filename to be matched is passed by the *path* argument. This must be a normal null-terminated string, but unless a length of -1 is passed in *pathlen*, only the first *pathlen* characters will be used in the filename expansion. If the length is specified as -1, the whole of the string will be expanded. A container of the following type is returned by `ef_expand_file()`.

```
typedef struct {
    int exists;   /* True if the files in files[] exist */
    int nfile;    /* The number of files in files[] */
    char **files; /* An array of 'nfile' filenames. */
} FileExpansion;
```

The `ef_expand_file()` function returns a pointer to a container whose contents are the results of the expansion. If there were no wildcards in the filename, the *nfile* member will be 1, and the *exists* member should be queried if it is important to know if the expanded file currently exists. If there were wild cards, then the contained *files*[] array will contain the names of the *nfile* existing files that matched the wild-carded filename, and the *exists* member will have the value 1. Note that the returned container belongs to the specified ef object, and its contents will change on each call, so if you need to retain the results of more than one call to `ef_expand_file()`, you should either make a private copy of the returned results, or create multiple file-expansion resource objects with multiple calls to `new_ExpandFile()`.

On error, `NULL` is returned, and an explanation of the error can be determined by calling `ef_last_error`(*ef*).

The `ef_last_error()` function returns the message which describes the error that occurred on the last call to `ef_expand_file()`, for the given (`ExpandFile` **ef*) resource object.

The `ef_list_expansions()` function provides a convenient way to list the filename expansions returned by `ef_expand_file()`. Like the `ls` utility, it arranges the filenames into equal width columns, each column having the width of the largest file. The number of columns used is thus determined by the length of the longest filename, and the specified terminal width. Beware that filenames that are longer than the specified terminal width are

printed without being truncated, so output longer than the specified terminal width can occur. The list is written to the `stdio` stream specified by the *fp* argument.

Thread Safety

It is safe to use the facilities of this module in multiple threads, provided that each thread uses a separately allocated `ExpandFile` object. In other words, if two threads want to do file expansion, they should each call `new_ExpandFile()` to allocate their own file-expansion objects.

Examples

EXAMPLE 1 Use of file expansion function.

The following is a complete example of how to use the file expansion function.

```
#include <stdio.h>
#include <libtecla.h>

int main(int argc, char *argv[])
{
    ExpandFile *ef;      /* The expansion resource object */
    char *filename;      /* The filename being expanded */
    FileExpansion *expn; /* The results of the expansion */
    int i;

    ef = new_ExpandFile();
    if(!ef)
        return 1;

    for(arg = *(argv++); arg; arg = *(argv++)) {
       if((expn = ef_expand_file(ef, arg, -1)) == NULL) {
          fprintf(stderr, "Error expanding %s (%s).\n", arg,
              ef_last_error(ef));
       } else {
          printf("%s matches the following files:\n", arg);
          for(i=0; i<expn->nfile; i++)
              printf(" %s\n", expn->files[i]);
       }
    }

    ef = del_ExpandFile(ef);
    return 0;
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also cpl_complete_word(3TECLA), gl_get_line(3TECLA), libtecla(3LIB), pca_lookup_file(3TECLA), attributes(5)

Name getauthattr, getauthnam, free_authattr, setauthattr, endauthattr, chkauthattr – get authorization entry

Synopsis

```
cc [ flag... ] file... –lsecdb –lsocket –lnsl [ library... ]
#include <auth_attr.h>
#include <secdb.h>

authattr_t *getauthattr(void);

authattr_t *getauthnam(const char *name);

void free_authattr(authattr_t *auth);

void setauthattr(void);

void endauthattr(void);

int chkauthattr(const char *authname, const char *username);
```

Description The `getauthattr()` and `getauthnam()` functions each return an `auth_attr(4)` entry. Entries can come from any of the sources specified in the `nsswitch.conf(4)` file.

The `getauthattr()` function enumerates `auth_attr` entries. The `getauthnam()` function searches for an `auth_attr` entry with a given authorization name *name*. Successive calls to these functions return either successive `auth_attr` entries or `NULL`.

Th internal representation of an `auth_attr` entry is an `authattr_t` structure defined in <`auth_attr.h`> with the following members:

```
char   *name;        /* name of the authorization */
char   *res1;        /* reserved for future use */
char   *res2;        /* reserved for future use */
char   *short_desc;  /* short description */
char   *long_desc;   /* long description */
kva_t  *attr;        /* array of key-value pair attributes */
```

The `setauthattr()` function "rewinds" to the beginning of the enumeration of `auth_attr` entries. Calls to `getauthnam()` can leave the enumeration in an indeterminate state. Therefore, `setauthattr()` should be called before the first call to `getauthattr()`.

The `endauthattr()` function may be called to indicate that `auth_attr` processing is complete; the system may then close any open `auth_attr` file, deallocate storage, and so forth.

The `chkauthattr()` function verifies whether or not a user has a given authorization. It first reads the `AUTHS_GRANTED` key in the `/etc/security/policy.conf` file and returns 1 if it finds a match for the given authorization. If `chkauthattr()` does not find a match and the *username* is the name of the "console user", defined as the owner of `/dev/console`, it first reads the `CONSOLE_USER` key in `/etc/security/policy.conf` and returns 1 if the given authorization is in any of the profiles specified in the `CONSOLE_USER` keyword, then reads the `PROFS_GRANTED` key in `/etc/security/policy.conf` and returns 1 if the given authorization is in any profiles specified with the `PROFS_GRANTED` keyword. If a match is not found from the

default authorizations and default profiles, `chkauthattr()` reads the `user_attr(4)` database. If it does not find a match in `user_attr`, it reads the `prof_attr(4)` database, using the list of profiles assigned to the user, and checks if any of the profiles assigned to the user has the given authorization. The `chkauthattr()` function returns 0 if it does not find a match in any of the three sources or if the user does not exist.

A user is considered to have been assigned an authorization if either of the following are true:

- The authorization name matches exactly any authorization assigned in the `user_attr` or `prof_attr` databases (authorization names are case-sensitive).
- The authorization name suffix is not the key word `grant` and the authorization name matches any authorization up to the asterisk (*) character assigned in the `user_attr` or `prof_attr` databases.

The examples in the following table illustrate the conditions under which a user is assigned an authorization.

`Authorization name`	`/etc/security/policy.conf` **or** `user_attr` **or** `prof_attr` **entry**	**Is user authorized?**
solaris.printer.postscript	solaris.printer.postscript	Yes
solaris.printer.postscript	solaris.printer.*	Yes
solaris.printer.grant	solaris.printer.*	No

The `free_authattr()` function releases memory allocated by the `getauthnam()` and `getauthattr()` functions.

Return Values The `getauthattr()` function returns a pointer to an `authattr_t` if it successfully enumerates an entry; otherwise it returns `NULL`, indicating the end of the enumeration.

The `getauthnam()` function returns a pointer to an `authattr_t` if it successfully locates the requested entry; otherwise it returns `NULL`.

The `chkauthattr()` function returns 1 if the user is authorized and 0 if the user does not exist or is not authorized.

Usage The `getauthattr()` and `getauthnam()` functions both allocate memory for the pointers they return. This memory should be deallocated with the `free_authattr()` call.

Individual attributes in the `attr` structure can be referred to by calling the `kva_match(3SECDB)` function.

Warnings Because the list of legal keys is likely to expand, code must be written to ignore unknown key-value pairs without error.

Files

`/etc/nsswitch.conf`	configuration file lookup information for the name server switch
`/etc/user_attr`	extended user attributes
`/etc/security/auth_attr`	authorization attributes
`/etc/security/policy.conf`	policy definitions
`/etc/security/prof_attr`	profile information

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `getexecattr(3SECDB)`, `getprofattr(3SECDB)`, `getuserattr(3SECDB)`, `auth_attr(4)`, `nsswitch.conf(4)`, `prof_attr(4)`, `user_attr(4)`, `attributes(5)`, `rbac(5)`

Name getdevicerange – get the label range of a device

Synopsis

```
cc [flag...] file... -lbsm -ltsol [library...]

#include <tsol/label.h>

blrange_t *getdevicerange(const char *device);
```

Description The `getdevicerange()` function returns the label range of a user-allocatable device.

If the label range is not specified for *device*, `getdevicerange()` returns the default values of `ADMIN_LOW` for the lower bound and `ADMIN_HIGH` for the upper bound of *device*.

From the command line, `list_devices(1)` can be used to see the label range of *device*.

Return Values The `getdevicerange()` function returns `NULL` on failure and sets `errno`. On successful completion, it returns a pointer to a `blrange_t` structure which must be freed by the caller, as follows:

```
blrange_t *range;
      ...
      m_label_free(range->lower_bound);
      m_label_free(range->upper_bound);
      free(range);
```

Errors The `getdevicerange()` function will fail if:

`EAGAIN` There is not enough memory available to allocate the required bytes. The application could try later.

`ENOMEM` The physical limits of the system are exceeded by size bytes of memory which cannot be allocated.

`ENOTSUP` Invalid upper or lower bound for device.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also `list_devices(1)`, `free(3C)`, `libtsol(3LIB)`, `m_label_free(3TSOL)`, `attributes(5)`

"Validating the Label Request Against the Printer's Label Range" in *Solaris Trusted Extensions Developer's Guide*

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name getexecattr, free_execattr, setexecattr, endexecattr, getexecuser, getexecprof, match_execattr – get execution profile entry

Synopsis

```
cc [ flag... ] file... -lsecdb -lsocket -lnsl [ library... ]
#include <exec_attr.h>
#include <secdb.h>

execattr_t *getexecattr(void);

void free_execattr(execattr_t *ep);

void setexecattr(void);

void endexecattr(void);

execattr_t *getexecuser(const char *username, const char *type,
     const char *id, int search_flag);

execattr_t *getexecprof(const char *profname, const char *type,
     const char *id, int search_flag);

execattr_t *match_execattr(execattr_t *ep, char *profname,
     char *type, char *id);
```

Description The `getexecattr()` function returns a single `exec_attr(4)` entry. Entries can come from any of the sources specified in the `nsswitch.conf(4)` file.

Successive calls to `getexecattr()` return either successive `exec_attr` entries or `NULL`. Because `getexecattr()` always returns a single entry, the `next` pointer in the `execattr_t` data structure points to `NULL`.

The internal representation of an `exec_attr` entry is an `execattr_t` structure defined in <exec_attr.h> with the following members:

```
char              *name;   /* name of the profile */
char              *type;   /* type of profile */
char              *policy; /* policy under which the attributes are */
                           /* relevant*/
char              *res1;   /* reserved for future use */
char              *res2;   /* reserved for future use */
char              *id;     /* unique identifier */
kva_t             *attr;   /* attributes */
struct execattr_s *next;   /* optional pointer to next profile */
```

The `free_execattr()` function releases memory. It follows the `next` pointers in the `execattr_t` structure so that the entire linked list is released.

The `setexecattr()` function "rewinds" to the beginning of the enumeration of `exec_attr` entries. Calls to `getexecuser()` can leave the enumeration in an indeterminate state. Therefore, `setexecattr()` should be called before the first call to `getexecattr()`.

The endexecattr() function can be called to indicate that exec_attr processing is complete; the library can then close any open exec_attr file, deallocate any internal storage, and so forth.

The getexecuser() function returns a linked list of entries that match the *type* and *id* arguments and have a profile that has been assigned to the user specified by *username*, as described in passwd(4). Profiles for the user are obtained from the list of default profiles in /etc/security/policy.conf (see policy.conf(4)) and the user_attr(4) database. Only entries in the name service scope for which the corresponding profile entry is found in the prof_attr(4) database are returned.

The getexecprof() function returns a linked list of entries that match the *type* and *id* arguments and have the profile specified by the *profname* argument. Only entries in the name service scope for which the corresponding profile entry is found in the prof_attr database are returned.

Using getexecuser() and getexecprof(), programmers can search for any *type* argument, such as the manifest constant KV_COMMAND. The arguments are logically AND-ed together so that only entries exactly matching all of the arguments are returned. Wildcard matching applies if there is no exact match for an ID. Any argument can be assigned the NULL value to indicate that it is not used as part of the matching criteria. The search_flag controls whether the function returns the first match (GET_ONE), setting the next pointer to NULL or all matching entries (GET_ALL), using the next pointer to create a linked list of all entries that meet the search criteria. See EXAMPLES.

Once a list of entries is returned by getexecuser() or getexecprof(), the convenience function match_execattr() can be used to identify an individual entry. It returns a pointer to the individual element with the same profile name (*profname*), type name (*type*), and *id*. Function parameters set to NULL are not used as part of the matching criteria. In the event that multiple entries meet the matching criteria, only a pointer to the first entry is returned. The kva_match(3SECDB) function can be used to look up a key in a key-value array.

Return Values Those functions returning data only return data related to the active policy. The getexecattr() function returns a pointer to a execattr_t if it successfully enumerates an entry; otherwise it returns NULL, indicating the end of the enumeration.

Usage The getexecattr(), getexecuser(), and getexecprof() functions all allocate memory for the pointers they return. This memory should be deallocated with the free_execattr() call. The match_execattr()(function does not allocate any memory. Therefore, pointers returned by this function should not be deallocated.

Individual attributes may be referenced in the attr structure by calling the kva_match(3SECDB) function.

Examples

EXAMPLE 1 Find all profiles that have the `ping` command.

```
if ((execprof=getexecprof(NULL, KV_COMMAND, "/usr/sbin/ping",
    GET_ONE)) == NULL) {
        /* do error */
}
```

EXAMPLE 2 Find the entry for the `ping` command in the Network Administration Profile.

```
if ((execprof=getexecprof("Network Administration", KV_COMMAND,
    "/usr/sbin/ping", GET_ALL))==NULL) {
        /* do error */
}
```

EXAMPLE 3 Tell everything that can be done in the Filesystem Security profile.

```
if ((execprof=getexecprof("Filesystem Security", KV_NULL, NULL,
    GET_ALL))==NULL)) {
        /* do error */
}
```

EXAMPLE 4 Tell if the `tar` utility is in a profile assigned to user wetmore. If there is no exact profile entry, the wildcard (*), if defined, is returned.

The following tells if the `tar` utility is in a profile assigned to user wetmore. If there is no exact profile entry, the wildcard (*), if defined, is returned.

```
if ((execprof=getexecuser("wetmore", KV_COMMAND, "/usr/bin/tar",
    GET_ONE))==NULL) {
        /* do error */
}
```

Files

`/etc/nsswitch.conf`	configuration file lookup information for the name server switch
`/etc/user_attr`	extended user attributes
`/etc/security/exec_attr`	execution profiles
`/etc/security/policy.conf`	policy definitions

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also

`getauthattr(3SECDB)`, `getuserattr(3SECDB)`, `kva_match(3SECDB)`, `exec_attr(4)`, `passwd(4)`, `policy.conf(4)`, `prof_attr(4)`, `user_attr(4)`, `attributes(5)`

Name getpathbylabel – return the zone pathname

Synopsis

```
cc [flags...] file... -ltsol [library...]

#include <tsol/label.h>

char *getpathbylabel(const char *path, char *resolved_path,
     size_t bufsize, const m_label_t *sl);
```

Description The `getpathbylabel()` function expands all symbolic links and resolves references to '/./', '/../', extra '/' characters, and stores the zone pathname in the buffer named by *resolved_path*. The *bufsize* argument specifies the size in bytes of this buffer. The resulting path will have no symbolic links components, nor any '/./', '/. ./'. This function can only be called from the global zone.

The zone pathname is relative to the sensitivity label *sl*. To specify a sensitivity label for a zone name which does not exist, the process must assert either the `PRIV_FILE_UPGRADE_SL` or `PRIV_FILE_DOWNGRADE_SL` privilege depending on whether the specified sensitivity label dominates or does not dominate the process sensitivity label.

Return Values The `getpathbylabel()` function returns a pointer to the *resolved_path* on success. Otherwise it returns `NULL` and sets `errno` to indicate the error.

Errors The `getpathbylabel()` function will fail if:

`EACCES`	Search permission is denied for a component of the path prefix of *path*.
`EFAULT`	*resolved_path* extends outside the process's allocated address space or beyond *bufsize* bytes.
`EINVAL`	*path* or *resolved_path* was `NULL`, current zone is not the global zone, or *sl* is invalid.
`EIO`	An I/O error occurred while reading from or writing to the file system.
`ELOOP`	Too many symbolic links were encountered in translating *path*.
`ENAMETOOLONG`	The length of the path argument exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX` (see `sysconf(3C)`) while `_POSIX_NO_TRUNC` is in effect (see `pathconf(2)`).
`ENOENT`	The named file does not exist.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `readlink(2)`, `getzonerootbyid(3TSOL)`, `libtsol(3LIB)`, `attributes(5)`, `labels(5)`

Warnings The `getpathbylabel()` function indirectly invokes the `readlink(2)` system call, and hence inherits the possibility of hanging due to inaccessible file system resources.

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name getplabel – get process label

Synopsis

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

#include <tsol/label.h>

int getplabel(m_label_t *label_p);
```

Description The `getplabel()` function obtains the sensitivity label of the calling process.

Return Values Upon successful completion, `getplabel()` returns 0. Otherwise it returns -1, *label_p* is unchanged, and `errno` is set to indicate the error.

Errors The `getplabel()` function fails and *label_p* does not refer to a valid sensitivity label if:

`EFAULT` *label_p* points to an invalid address.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also `ucred_getlabel(3C)`, `libtsol(3LIB)`, `m_label_alloc(3TSOL)`, `m_label_free(3TSOL)`, `attributes(5)`

"Obtaining a Process Label" in *Solaris Trusted Extensions Developer's Guide*

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

This function returns different values for system processes than `ucred_getlabel(3C)` returns.

Name getprofattr, getprofnam, free_profattr, setprofattr, endprofattr, getproflist, free_proflist – get profile description and attributes

Synopsis

```
cc [ flag... ] file... -lsecdb -lsocket -lnsl [ library... ]
#include <prof_attr.h>

profattr_t *getprofattr(void);

profattr_t *getprofnam(const char *name);

void free_profattr(profattr_t *pd);

void setprofattr(void);

void endprofattr(void);

void getproflist(const char *profname, char **proflist, int *profcnt);

void free_proflist(char **proflist, int profcnt);
```

Description The `getprofattr()` and `getprofnam()` functions each return a `prof_attr` entry. Entries can come from any of the sources specified in the `nsswitch.conf(4)` file.

The `getprofattr()` function enumerates `prof_attr` entries. The `getprofnam()` function searches for a `prof_attr` entry with a given *name*. Successive calls to these functions return either successive `prof_attr` entries or `NULL`.

The internal representation of a `prof_attr` entry is a `profattr_t` structure defined in <`prof_attr.h`> with the following members:

```
char     *name;   /* Name of the profile */
char     *res1;   /* Reserved for future use */
char     *res2;   /* Reserved for future use */
char     *desc;   /* Description/Purpose of the profile */
kva_t *attr;   /* Profile attributes */
```

The `free_profattr()` function releases memory allocated by the `getprofattr()` and `getprofnam()` functions.

The `setprofattr()` function "rewinds" to the beginning of the enumeration of `prof_attr` entries. Calls to `getprofnam()` can leave the enumeration in an indeterminate state. Therefore, `setprofattr()` should be called before the first call to `getprofattr()`.

The `endprofattr()` function may be called to indicate that `prof_attr` processing is complete; the system may then close any open `prof_attr` file, deallocate storage, and so forth.

The `getproflist()` function searches for the list of sub-profiles found in the given *profname* and allocates memory to store this list in *proflist*. The given *profname* will be included in the list of sub-profiles. The *profcnt* argument indicates the number of items currently valid in *proflist*. Memory allocated by `getproflist()` should be freed using the `free_proflist()` function.

The `free_proflist()` function frees memory allocated by the `getproflist()` function. The *profcnt* argument specifies the number of items to free from the *proflist* argument.

Return Values The `getprofattr()` function returns a pointer to a `profattr_t` if it successfully enumerates an entry; otherwise it returns `NULL`, indicating the end of the enumeration.

The `getprofnam()` function returns a pointer to a `profattr_t` if it successfully locates the requested entry; otherwise it returns `NULL`.

Usage Individual attributes in the `prof_attr_t` structure can be referred to by calling the `kva_match(3SECDB)` function.

Because the list of legal keys is likely to expand, any code must be written to ignore unknown key-value pairs without error.

The `getprofattr()` and `getprofnam()` functions both allocate memory for the pointers they return. This memory should be deallocated with the `free_profattr()` function.

Files `/etc/security/prof_attr` profiles and their descriptions

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `auths(1)`, `profiles(1)`, `getexecattr(3SECDB)`, `getauthattr(3SECDB)`, `prof_attr(4)`

Name getuserattr, getusernam, getuseruid, free_userattr, setuserattr, enduserattr, fgetuserattr – get user_attr entry

Synopsis

```
cc [ flag... ] file... -lsecdb -lsocket -lnsl [ library... ]
#include <user_attr.h>

userattr_t *getuserattr(void);

userattr_t *getusernam(const char *name);

userattr_t *getuseruid(uid_t uid);

void free_userattr(userattr_t *userattr);

void setuserattr(void);

void enduserattr(void);

userattr_t *fgetuserattr(FILE *f);
```

Description The `getuserattr()`, `getusernam()`, and `getuseruid()` functions each return a `user_attr(4)` entry. Entries can come from any of the sources specified in the `nsswitch.conf(4)` file. The `getuserattr()` function enumerates `user_attr` entries. The `getusernam()` function searches for a `user_attr` entry with a given user name *name*. The `getuseruid()` function searches for a `user_attr` entry with a given user ID *uid*. Successive calls to these functions return either successive `user_attr` entries or `NULL`.

The `fgetuserattr()` function does not use `nsswitch.conf` but reads and parses the next line from the stream *f*. This stream is assumed to have the format of the `user_attr` files.

The `free_userattr()` function releases memory allocated by the `getusernam()`, `getuserattr()`, and `fgetuserattr()` functions.

The internal representation of a `user_attr` entry is a `userattr_t` structure defined in <`user_attr.h`> with the following members:

```
char  *name;       /* name of the user */
char  *qualifier;  /* reserved for future use */
char  *res1;       /* reserved for future use */
char  *res2;       /* reserved for future use */
kva_t *attr;       /* list of attributes */
```

The `setuserattr()` function "rewinds" to the beginning of the enumeration of `user_attr` entries. Calls to `getusernam()` may leave the enumeration in an indeterminate state, so `setuserattr()` should be called before the first call to `getuserattr()`.

The `enduserattr()` function may be called to indicate that `user_attr` processing is complete; the library may then close any open `user_attr` file, deallocate any internal storage, and so forth.

Return Values The `getuserattr()` function returns a pointer to a `userattr_t` if it successfully enumerates an entry; otherwise it returns `NULL`, indicating the end of the enumeration.

The `getusernam()` function returns a pointer to a `userattr_t` if it successfully locates the requested entry; otherwise it returns `NULL`.

Usage The `getuserattr()` and `getusernam()` functions both allocate memory for the pointers they return. This memory should be deallocated with the `free_userattr()` function.

Individual attributes can be referenced in the `attr` structure by calling the `kva_match(3SECDB)` function.

Warinings Because the list of legal keys is likely to expand, code must be written to ignore unknown key-value pairs without error.

Files

`/etc/user_attr`	extended user attributes
`/etc/nsswitch.conf`	configuration file lookup information for the name server switch

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `getauthattr(3SECDB)`, `getexecattr(3SECDB)`, `getprofattr(3SECDB)`, `user_attr(4)`, `attributes(5)`

Name getuserrange – get the label range of a user

Synopsis

```
cc [flags...] file... -ltsol [library...]

#include <tsol/label.h>

m_range_t *getuserrange(const char *username);
```

Description The `getuserrange()` function returns the label range of *username*. The lower bound in the range is used as the initial workspace label when a user logs into a multilevel desktop. The upper bound, or clearance, is used as an upper limit to the available labels that a user can assign to labeled workspaces.

The default value for a user's label range is specified in `label_encodings(4)`. Overriding values for individual users are specified in `user_attr(4)`.

Return Values The `getuserrange()` function returns `NULL` if the memory allocation fails. Otherwise, the function returns a structure which must be freed by the caller, as follows:

```
m_range_t  *range;
    ...
    m_label_free(range->lower_bound);
    m_label_free(range->upper_bound);
    free(range);
```

Errors The `getuserrange()` function will fail if:

`ENOMEM` The physical limits of the system are exceeded by size bytes of memory which cannot be allocated.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	See below.
MT-Level	MT-Safe

The `getuserrange()` function is Committed for systems that implement the Defense Intelligence Agency (DIA) MAC policy of `label_encodings(4)`. Other policies might exist in a future release of Trusted Extensions that might make obsolete or supplement `label_encodings`.

See Also `free(3C)`, `libtsol(3LIB)`, `m_label_free(3TSOL)`, `label_encodings(4)`, `user_attr(4)`, `attributes(5)`

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name getzonelabelbyid, getzonelabelbyname, getzoneidbylabel – map between zones and labels

Synopsis

```
cc [flags...] file... -ltsol [library...]

#include <tsol/label.h>

m_label_t *getzonelabelbyid(zoneid_t zoneid);

m_label_t *getzonelabelbyname(const char *zonename);

zoneid_t *getzoneidbylabel(const m_label_t *label);
```

Description The `getzonelabelbyid()` function returns the mandatory access control (MAC) label of *zoneid*.

The `getzonelabelbyname()` function returns the MAC label of the zone whose name is *zonename*.

The `getzoneidbylabel()` function returns the zone ID of the zone whose label is *label*.

All of these functions require that the specified zone's state is at least `ZONE_IS_READY`. The zone of the calling process must dominate the specified zone's label, or the calling process must be in the global zone.

Return Values On successful completion, the `getzonelabelbyid()` and `getzonelabelbyname()` functions return a pointer to a sensitivity label that is allocated within these functions. To free the storage, use `m_label_free(3TSOL)`. If the zone does not exist, `NULL` is returned.

On successful completion, the `getzoneidbylabel()` function returns the zone ID with the matching label. If there is no matching zone, the function returns `-1`.

Errors The `getzonelabelbyid()` and `getzonelabelbyname()` functions will fail if:

`ENOENT` The specified zone does not exist.

The `getzonelabelbyid()` function will fail if:

`ENOENT` No zone corresponds to the specified label.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `Intro(2)`, `getzonenamebyid(3C)`, `getzoneidbyname(3C)`, `libtsol(3LIB)`, `m_label_free(3TSOL)`, `attributes(5)`, `labels(5)`

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name getzonerootbyid, getzonerootbylabel, getzonerootbyname – map between zone root pathnames and labels

Synopsis

```
cc [flags...] file... -ltsol [library...]

#include <tsol/label.h>

char *getzonerootbyid(zoneid_t zoneid);

char *getzonerootbylabel(const m_label_t *label);

char *getzonerootbyname(const char *zonename);
```

Description The `getzonerootbyid()` function returns the root pathname of *zoneid*.

The `getzonerootbylabel()` function returns the root pathname of the zone whose label is *label*.

The `getzonerootbyname()` function returns the root pathname of *zonename*.

All of these functions require that the specified zone's state is at least `ZONE_IS_READY`. The zone of the calling process must dominate the specified zone's label, or the calling process must be in the global zone. The returned pathname is relative to the root path of the caller's zone.

Return Values On successful completion, the `getzonerootbyid()`, `getzonerootbylabel()`, and `getzonerootbyname()` functions return a pointer to a pathname that is allocated within these functions. To free the storage, use `free(3C)`. On failure, these functions return `NULL` and set `errno` to indicate the error.

Errors These functions will fail if:

`EFAULT` Invalid argument; pointer location is invalid.

`EINVAL` *zoneid* invalid, or zone not found or not ready.

`ENOENT` Zone does not exist.

`ENOMEM` Unable to allocate pathname.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `Intro(2)`, `free(3C)`, `getzonenamebyid(3C)`, `libtsol(3LIB)`, `attributes(5)`, `labels(5)`

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name gl_get_line, new_GetLine, del_GetLine, gl_customize_completion, gl_change_terminal, gl_configure_getline, gl_load_history, gl_save_history, gl_group_history, gl_show_history, gl_watch_fd, gl_inactivity_timeout, gl_terminal_size, gl_set_term_size, gl_resize_history, gl_limit_history, gl_clear_history, gl_toggle_history, gl_lookup_history, gl_state_of_history, gl_range_of_history, gl_size_of_history, gl_echo_mode, gl_replace_prompt, gl_prompt_style, gl_ignore_signal, gl_trap_signal, gl_last_signal, gl_completion_action, gl_register_action, gl_display_text, gl_return_status, gl_error_message, gl_catch_blocked, gl_list_signals, gl_bind_keyseq, gl_erase_terminal, gl_automatic_history, gl_append_history, gl_query_char, gl_read_char – allow the user to compose an input line

Synopsis

```
cc [ flag... ] file... -ltecla [ library... ]
#include <stdio.h>
#include <libtecla.h>

GetLine *new_GetLine(size_t linelen, size_t histlen);

GetLine *del_GetLine(GetLine *gl);

char *gl_get_line(GetLine *gl, const char *prompt,
     const char *start_line, int start_pos);

int gl_query_char(GetLine *gl, const char *prompt, char defchar);

int gl_read_char(GetLine *gl);

int gl_customize_completion(GetLine *gl, void *data,
     CplMatchFn *match_fn);

int gl_change_terminal(GetLine *gl, FILE *input_fp,
     FILE *output_fp, const char *term);

int gl_configure_getline(GetLine *gl, const char *app_string,
     const char *app_file, const char *user_file);

int gl_bind_keyseq(GetLine *gl, GlKeyOrigin origin,
     const char *keyseq, const char *action);

int gl_save_history(GetLine *gl, const char *filename,
     const char *comment, int max_lines);

int gl_load_history(GetLine *gl, const char *filename,
     const char *comment);

int gl_watch_fd(GetLine *gl, int fd, GlFdEvent event,
     GlFdEventFn *callback, void *data);

int gl_inactivity_timeout(GetLine *gl, GlTimeoutFn *callback,
     void *data, unsigned long sec, unsigned long nsec);

int gl_group_history(GetLine *gl, unsigned stream);

int gl_show_history(GetLine *gl, FILE *fp, const char *fmt,
     int all_groups, int max_lines);

int gl_resize_history(GetLine *gl, size_t bufsize);
```

```
void gl_limit_history(GetLine *gl, int max_lines);

void gl_clear_history(GetLine *gl, int all_groups);

void gl_toggle_history(GetLine *gl, int enable);

GlTerminalSize gl_terminal_size(GetLine *gl, int def_ncolumn,
     int def_nline);

int gl_set_term_size(GetLine *gl, int ncolumn, int nline);

int gl_lookup_history(GetLine *gl, unsigned long id,
     GlHistoryLine *hline);

void gl_state_of_history(GetLine *gl, GlHistoryState *state);

void gl_range_of_history(GetLine *gl, GlHistoryRange *range);

void gl_size_of_history(GetLine *gl, GlHistorySize *size);

void gl_echo_mode(GetLine *gl, int enable);

void gl_replace_prompt(GetLine *gl, const char *prompt);

void gl_prompt_style(GetLine *gl, GlPromptStyle style);

int gl_ignore_signal(GetLine *gl, int signo);

int gl_trap_signal(GetLine *gl, int signo, unsigned flags,
     GlAfterSignal after, int errno_value);

int gl_last_signal(GetLine *gl);

int gl_completion_action(GetLine *gl, void *data,
     CplMatchFn *match_fn, int list_only, const char *name,
     const char *keyseq);

int gl_register_action(GetLine *gl, void *data, GlActionFn *fn,
     const char *name, const char *keyseq);

int gl_display_text(GetLine *gl, int indentation,
     const char *prefix, const char *suffix, int fill_char,
     int def_width, int start, const char *string);

GlReturnStatus gl_return_status(GetLine *gl);

const char *gl_error_message(GetLine *gl, char *buff, size_t n);

void gl_catch_blocked(GetLine *gl);

int gl_list_signals(GetLine *gl, sigset_t *set);

int gl_append_history(GetLine *gl, const char *line);

int gl_automatic_history(GetLine *gl, int enable);

int gl_erase_terminal(GetLine *gl);
```

Description The `gl_get_line()` function is part of the libtecla(3LIB) library. If the user is typing at a terminal, each call prompts them for an line of input, then provides interactive editing facilities, similar to those of the UNIX `tcsh` shell. In addition to simple command-line editing, it supports recall of previously entered command lines, TAB completion of file names, and in-line wild-card expansion of filenames. Documentation of both the user-level command-line editing features and all user configuration options can be found on the tecla(5) manual page.

An Example The following shows a complete example of how to use the `gl_get_line()` function to get input from the user:

```
#include <stdio.h>
#include <locale.h>
#include <libtecla.h>

int main(int argc, char *argv[])
{
  char *line;    /* The line that the user typed */
  GetLine *gl;   /* The gl_get_line() resource object */

  setlocale(LC_CTYPE, ""); /* Adopt the user's choice */
                           /* of character set. */

  gl = new_GetLine(1024, 2048);
  if(!gl)
    return 1;
  while((line=gl_get_line(gl, "$ ", NULL, -1)) != NULL &&
        strcmp(line, "exit\n") != 0)
    printf("You typed: %s\n", line);

  gl = del_GetLine(gl);
  return 0;
}
```

In the example, first the resources needed by the `gl_get_line()` function are created by calling `new_GetLine()`. This allocates the memory used in subsequent calls to the `gl_get_line()` function, including the history buffer for recording previously entered lines. Then one or more lines are read from the user, until either an error occurs, or the user types exit. Then finally the resources that were allocated by `new_GetLine()`, are returned to the system by calling `del_GetLine()`. Note the use of the `NULL` return value of `del_GetLine()` to make *gl* `NULL`. This is a safety precaution. If the program subsequently attempts to pass *gl* to `gl_get_line()`, said function will complain, and return an error, instead of attempting to use the deleted resource object.

The Functions Used In The Example The `new_GetLine()` function creates the resources used by the `gl_get_line()` function and returns an opaque pointer to the object that contains them. The maximum length of an input line is specified by the *linelen* argument, and the number of bytes to allocate for storing history lines is set by the *histlen* argument. History lines are stored back-to-back in a single buffer of

this size. Note that this means that the number of history lines that can be stored at any given time, depends on the lengths of the individual lines. If you want to place an upper limit on the number of lines that can be stored, see the description of the `gl_limit_history()` function. If you do not want history at all, specify *histlen* as zero, and no history buffer will be allocated.

On error, a message is printed to `stderr` and `NULL` is returned.

The `del_GetLine()` function deletes the resources that were returned by a previous call to `new_GetLine()`. It always returns `NULL` (for example, a deleted object). It does nothing if the *gl* argument is `NULL`.

The `gl_get_line()` function can be called any number of times to read input from the user. The gl argument must have been previously returned by a call to `new_GetLine()`. The *prompt* argument should be a normal null-terminated string, specifying the prompt to present the user with. By default prompts are displayed literally, but if enabled with the `gl_prompt_style()` function, prompts can contain directives to do underlining, switch to and from bold fonts, or turn highlighting on and off.

If you want to specify the initial contents of the line for the user to edit, pass the desired string with the *start_line* argument. You can then specify which character of this line the cursor is initially positioned over by using the *start_pos* argument. This should be -1 if you want the cursor to follow the last character of the start line. If you do not want to preload the line in this manner, send *start_line* as `NULL`, and set *start_pos* to -1.

The `gl_get_line()` function returns a pointer to the line entered by the user, or `NULL` on error or at the end of the input. The returned pointer is part of the specified *gl* resource object, and thus should not be freed by the caller, or assumed to be unchanging from one call to the next. When reading from a user at a terminal, there will always be a newline character at the end of the returned line. When standard input is being taken from a pipe or a file, there will similarly be a newline unless the input line was too long to store in the internal buffer. In the latter case you should call `gl_get_line()` again to read the rest of the line. Note that this behavior makes `gl_get_line()` similar to `fgets(3C)`. When `stdin` is not connected to a terminal, `gl_get_line()` simply calls `fgets()`.

The Return Status Of `gl_get_line()`

The `gl_get_line()` function has two possible return values: a pointer to the completed input line, or `NULL`. Additional information about what caused `gl_get_line()` to return is available both by inspecting `errno` and by calling the `gl_return_status()` function.

The following are the possible enumerated values returned by `gl_return_status()`:

`GLR_NEWLINE` The last call to `gl_get_line()` successfully returned a completed input line.

`GLR_BLOCKED` The `gl_get_line()` function was in non-blocking server mode, and returned early to avoid blocking the process while waiting for terminal I/O. The `gl_pending_io()` function can be used to see what type of I/O `gl_get_line()` was waiting for. See the `gl_io_mode(3TECLA)`.

GLR_SIGNAL — A signal was caught by gl_get_line() that had an after-signal disposition of GLS_ABORT. See gl_trap_signal().

GLR_TIMEOUT — The inactivity timer expired while gl_get_line() was waiting for input, and the timeout callback function returned GLTO_ABORT. See gl_inactivity_timeout() for information about timeouts.

GLR_FDABORT — An application I/O callback returned GLFD_ABORT. Ssee gl_watch_fd().

GLR_EOF — End of file reached. This can happen when input is coming from a file or a pipe, instead of the terminal. It also occurs if the user invokes the list-or-eof or del-char-or-list-or-eof actions at the start of a new line.

GLR_ERROR — An unexpected error caused gl_get_line() to abort (consult errno and/or gl_error_message() for details.

When gl_return_status() returns GLR_ERROR and the value of errno is not sufficient to explain what happened, you can use the gl_error_message() function to request a description of the last error that occurred.

The return value of gl_error_message() is a pointer to the message that occurred. If the *buff* argument is NULL, this will be a pointer to a buffer within *gl* whose value will probably change on the next call to any function associated with gl_get_line(). Otherwise, if a non-null *buff* argument is provided, the error message, including a '\0' terminator, will be written within the first *n* elements of this buffer, and the return value will be a pointer to the first element of this buffer. If the message will not fit in the provided buffer, it will be truncated to fit.

Optional Prompt Formatting

Whereas by default the prompt string that you specify is displayed literally without any special interpretation of the characters within it, the gl_prompt_style() function can be used to enable optional formatting directives within the prompt.

The *style* argument, which specifies the formatting style, can take any of the following values:

GL_FORMAT_PROMPT — In this style, the formatting directives described below, when included in prompt strings, are interpreted as follows:

- %B — Display subsequent characters with a bold font.
- %b — Stop displaying characters with the bold font.
- %F — Make subsequent characters flash.
- %f — Turn off flashing characters.
- %U — Underline subsequent characters.
- %u — Stop underlining characters.
- %P — Switch to a pale (half brightness) font.
- %p — Stop using the pale font.

`%S` Highlight subsequent characters (also known as standout mode).

`%s` Stop highlighting characters.

`%V` Turn on reverse video.

`%v` Turn off reverse video.

`%%` Display a single % character.

For example, in this mode, a prompt string like "%UOK%u$" would display the prompt "OK$", but with the OK part underlined.

Note that although a pair of characters that starts with a % character, but does not match any of the above directives is displayed literally, if a new directive is subsequently introduced which does match, the displayed prompt will change, so it is better to always use %% to display a literal %.

Also note that not all terminals support all of these text attributes, and that some substitute a different attribute for missing ones.

`GL_LITERAL_PROMPT` In this style, the prompt string is printed literally. This is the default style.

Alternate Configuration Sources

By default users have the option of configuring the behavior of `gl_get_line()` with a configuration file called `.teclarc` in their home directories. The fact that all applications share this same configuration file is both an advantage and a disadvantage. In most cases it is an advantage, since it encourages uniformity, and frees the user from having to configure each application separately. In some applications, however, this single means of configuration is a problem. This is particularly true of embedded software, where there's no filesystem to read a configuration file from, and also in applications where a radically different choice of keybindings is needed to emulate a legacy keyboard interface. To cater for such cases, the `gl_configure_getline()` function allows the application to control where configuration information is read from.

The `gl_configure_getline()` function allows the configuration commands that would normally be read from a user's `~/.teclarc` file, to be read from any or none of, a string, an application specific configuration file, and/or a user-specific configuration file. If this function is called before the first call to `gl_get_line()`, the default behavior of reading `~/.teclarc` on the first call to `gl_get_line()` is disabled, so all configurations must be achieved using the configuration sources specified with this function.

If *app_string* != `NULL`, then it is interpreted as a string containing one or more configuration commands, separated from each other in the string by embedded newline characters. If *app_file* != `NULL` then it is interpreted as the full pathname of an application-specific

configuration file. If user_file != NULL then it is interpreted as the full path name of a user-specific configuration file, such as ~/.teclarc. For example, in the call

```
gl_configure_getline(gl, "edit-mode vi \
 nobeep",
                     "/usr/share/myapp/teclarc", "~/.teclarc");
```

The *app_string* argument causes the calling application to start in vi(1) edit-mode, instead of the default emacs mode, and turns off the use of the terminal bell by the library. It then attempts to read system-wide configuration commands from an optional file called /usr/share/myapp/teclarc, then finally reads user-specific configuration commands from an optional .teclarc file in the user's home directory. Note that the arguments are listed in ascending order of priority, with the contents of *app_string* being potentially over riden by commands in *app_file*, and commands in *app_file* potentially being overriden by commands in *user_file*.

You can call this function as many times as needed, the results being cumulative, but note that copies of any file names specified with the *app_file* and *user_file* arguments are recorded internally for subsequent use by the read-init-files key-binding function, so if you plan to call this function multiple times, be sure that the last call specifies the filenames that you want re-read when the user requests that the configuration files be re-read.

Individual key sequences can also be bound and unbound using the gl_bind_keyseq() function. The *origin* argument specifies the priority of the binding, according to whom it is being established for, and must be one of the following two values.

GL_USER_KEY The user requested this key-binding.

GL_APP_KEY This is a default binding set by the application.

When both user and application bindings for a given key sequence have been specified, the user binding takes precedence. The application's binding is subsequently reinstated if the user's binding is later unbound with either another call to this function, or a call to gl_configure_getline().

The *keyseq* argument specifies the key sequence to be bound or unbound, and is expressed in the same way as in a ~/.teclarc configuration file. The *action* argument must either be a string containing the name of the action to bind the key sequence to, or it must be NULL or "" to unbind the key sequence.

Customized Word Completion

If in your application you would like to have TAB completion complete other things in addition to or instead of filenames, you can arrange this by registering an alternate completion callback function with a call to the gl_customize_completion() function.

The *data* argument provides a way for your application to pass arbitrary, application-specific information to the callback function. This is passed to the callback every time that it is called. It might for example point to the symbol table from which possible completions are to be

sought. The *match_fn* argument specifies the callback function to be called. The *CplMatchFn* function type is defined in <libtecla.h>, as is a CPL_MATCH_FN() macro that you can use to declare and prototype callback functions. The declaration and responsibilities of callback functions are described in depth on the cpl_complete_word(3TECLA) manual page.

The callback function is responsible for looking backwards in the input line from the point at which the user pressed TAB, to find the start of the word being completed. It then must lookup possible completions of this word, and record them one by one in the WordCompletion object that is passed to it as an argument, by calling the cpl_add_completion() function. If the callback function wants to provide filename completion in addition to its own specific completions, it has the option of itself calling the builtin filename completion callback. This also is documented on the cpl_complete_word(3TECLA) manual page.

If you would like gl_get_line() to return the current input line when a successful completion is been made, you can arrange this when you call cpl_add_completion() by making the last character of the continuation suffix a newline character. The input line will be updated to display the completion, together with any contiuation suffix up to the newline character, and gl_get_line() will return this input line.

If your callback function needs to write something to the terminal, it must call gl_normal_io() before doing so. This will start a new line after the input line that is currently being edited, reinstate normal terminal I/O, and notify gl_get_line() that the input line will need to be redrawn when the callback returns.

Adding Completion Actions

In the previous section the ability to customize the behavior of the only default completion action, complete-word, was described. In this section the ability to install additional action functions, so that different types of word completion can be bound to different key sequences, is described. This is achieved by using the gl_completion_action() function.

The *data* and *match_fn* arguments are as described on the cpl_complete_word(3TECLA) manual page, and specify the callback function that should be invoked to identify possible completions. The *list_only* argument determines whether the action that is being defined should attempt to complete the word as far as possible in the input line before displaying any possible ambiguous completions, or whether it should simply display the list of possible completions without touching the input line. The former option is selected by specifying a value of 0, and the latter by specifying a value of 1. The *name* argument specifies the name by which configuration files and future invocations of this function should refer to the action. This must either be the name of an existing completion action to be changed, or be a new unused name for a new action. Finally, the *keyseq* argument specifies the default key sequence to bind the action to. If this is NULL, no new key sequence will be bound to the action.

Beware that in order for the user to be able to change the key sequence that is bound to actions that are installed in this manner, you shouldcall gl_completion_action() to install a given action for the first time between calling new_GetLine() and the first call to gl_get_line().

Otherwise, when the user's configuration file is read on the first call to `gl_get_line()`, the name of the your additional action will not be known, and any reference to it in the configuration file will generate an error.

As discussed for `gl_customize_completion()`, if your callback function needs to write anything to the terminal, it must call `gl_normal_io()` before doing so.

Defining Custom Actions

Although the built-in key-binding actions are sufficient for the needs of most applications, occasionally a specialized application may need to define one or more custom actions, bound to application-specific key sequences. For example, a sales application would benefit from having a key sequence that displayed the part name that corresponded to a part number preceding the cursor. Such a feature is clearly beyond the scope of the built-in action functions. So for such special cases, the `gl_register_action()` function is provided.

The `gl_register_action()` function lets the application register an external function, *fn*, that will thereafter be called whenever either the specified key sequence, *keyseq*, is entered by the user, or the user enters any other key sequence that the user subsequently binds to the specified action name, *name*, in their configuration file. The *data* argument can be a pointer to anything that the application wants to have passed to the action function, *fn*, whenever that function is invoked.

The action function, *fn*, should be declared using the `GL_ACTION_FN()` macro, which is defined in <`libtecla.h`>.

```
#define GL_ACTION_FN(fn) GlAfterAction (fn)(GetLine *gl, \
                       void *data, int count, size_t curpos, \
                       const char *line)
```

The *gl* and *data* arguments are those that were previously passed to `gl_register_action()` when the action function was registered. The *count* argument is a numeric argument which the user has the option of entering using the digit-argument action, before invoking the action. If the user does not enter a number, then the *count* argument is set to 1. Nominally this argument is interpreted as a repeat count, meaning that the action should be repeated that many times. In practice however, for some actions a repeat count makes little sense. In such cases, actions can either simply ignore the *count* argument, or use its value for a different purpose.

A copy of the current input line is passed in the read-only *line* argument. The current cursor position within this string is given by the index contained in the *curpos* argument. Note that direct manipulation of the input line and the cursor position is not permitted because the rules dictated by various modes (such as `vi` mode versus `emacs` mode, no-echo mode, and insert mode versus overstrike mode) make it too complex for an application writer to write a conforming editing action, as well as constrain future changes to the internals of `gl_get_line()`. A potential solution to this dilemma would be to allow the action function to edit the line using the existing editing actions. This is currently under consideration.

If the action function wishes to write text to the terminal without this getting mixed up with the displayed text of the input line, or read from the terminal without having to handle raw terminal I/O, then before doing either of these operations, it must temporarily suspend line editing by calling the `gl_normal_io()` function. This function flushes any pending output to the terminal, moves the cursor to the start of the line that follows the last terminal line of the input line, then restores the terminal to a state that is suitable for use with the C `stdio` facilities. The latter includes such things as restoring the normal mapping of \n to \r\n, and, when in server mode, restoring the normal blocking form of terminal I/O. Having called this function, the action function can read from and write to the terminal without the fear of creating a mess. It is not necessary for the action function to restore the original editing environment before it returns. This is done automatically by `gl_get_line()` after the action function returns. The following is a simple example of an action function which writes the sentence "Hello world" on a new terminal line after the line being edited. When this function returns, the input line is redrawn on the line that follows the "Hello world" line, and line editing resumes.

```
static GL_ACTION_FN(say_hello_fn)
{
    if(gl_normal_io(gl))   /* Temporarily suspend editing */
        return GLA_ABORT;
    printf("Hello world\n");
    return GLA_CONTINUE;
}
```

Action functions must return one of the following values, to tell `gl_get_line()` how to proceed.

`GLA_ABORT`	Cause `gl_get_line()` to return `NULL`.
`GLA_RETURN`	Cause `gl_get_line()` to return the completed input line
`GLA_CONTINUE`	Resume command-line editing.

Note that the *name* argument of `gl_register_action()` specifies the name by which a user can refer to the action in their configuration file. This allows them to re-bind the action to an alternate key-seqeunce. In order for this to work, it is necessary to call `gl_register_action()` between calling `new_GetLine()` and the first call to `gl_get_line()`.

History Files

To save the contents of the history buffer before quitting your application and subsequently restore them when you next start the application, the `gl_save_history()` and `gl_load_history()` functions are provided.

The *filename* argument specifies the name to give the history file when saving, or the name of an existing history file, when loading. This may contain home directory and environment variable expressions, such as `~/.myapp_history` or `$HOME/.myapp_history`.

Along with each history line, additional information about it, such as its nesting level and when it was entered by the user, is recorded as a comment preceding the line in the history file. Writing this as a comment allows the history file to double as a command file, just in case you wish to replay a whole session using it. Since comment prefixes differ in different languages, the comment argument is provided for specifying the comment prefix. For example, if your application were a UNIX shell, such as the Bourne shell, you would specify "#" here. Whatever you choose for the comment character, you must specify the same prefix to `gl_load_history()` that you used when you called `gl_save_history()` to write the history file.

The *max_lines* argument must be either -1 to specify that all lines in the history list be saved, or a positive number specifying a ceiling on how many of the most recent lines should be saved.

Both fuctions return non-zero on error, after writing an error message to `stderr`. Note that `gl_load_history()` does not consider the non-existence of a file to be an error.

Multiple History Lists

If your application uses a single `GetLine` object for entering many different types of input lines, you might want `gl_get_line()` to distinguish the different types of lines in the history list, and only recall lines that match the current type of line. To support this requirement, `gl_get_line()` marks lines being recorded in the history list with an integer identifier chosen by the application. Initially this identifier is set to 0 by `new_GetLine()`, but it can be changed subsequently by calling `gl_group_history()`.

The integer identifier ID can be any number chosen by the application, but note that `gl_save_history()` and `gl_load_history()` preserve the association between identifiers and historical input lines between program invocations, so you should choose fixed identifiers for the different types of input line used by your application.

Whenever `gl_get_line()` appends a new input line to the history list, the current history identifier is recorded with it, and when it is asked to recall a historical input line, it only recalls lines that are marked with the current identifier.

Displaying History

The history list can be displayed by calling `gl_show_history()`. This function displays the current contents of the history list to the `stdio` output stream *fp*. If the *max_lines* argument is greater than or equal to zero, then no more than this number of the most recent lines will be displayed. If the *all_groups* argument is non-zero, lines from all history groups are displayed. Otherwise only those of the currently selected history group are displayed. The format string argument, *fmt*, determines how the line is displayed. This can contain arbitrary characters which are written verbatim, interleaved with any of the following format directives:

`%D` The date on which the line was originally entered, formatted like 2001-11-20.

`%T` The time of day when the line was entered, formatted like 23:59:59.

`%N` The sequential entry number of the line in the history buffer.

`%G` The number of the history group which the line belongs to.

`%%` A literal % character.

`%H` The history line itself.

Thus a format string like "%D %T %H0" would output something like:

```
2001-11-20 10:23:34  Hello world
```

Note the inclusion of an explicit newline character in the format string.

Looking Up History

The `gl_lookup_history()` function allows the calling application to look up lines in the history list.

The *id* argument indicates which line to look up, where the first line that was entered in the history list after `new_GetLine()` was called is denoted by 0, and subsequently entered lines are denoted with successively higher numbers. Note that the range of lines currently preserved in the history list can be queried by calling the `gl_range_of_history()` function. If the requested line is in the history list, the details of the line are recorded in the variable pointed to by the *hline* argument, and 1 is returned. Otherwise 0 is returned, and the variable pointed to by *hline* is left unchanged.

Beware that the string returned in *hline->line* is part of the history buffer, so it must not be modified by the caller, and will be recycled on the next call to any function that takes *gl* as its argument. Therefore you should make a private copy of this string if you need to keep it.

Manual History Archival

By default, whenever a line is entered by the user, it is automatically appended to the history list, just before `gl_get_line()` returns the line to the caller. This is convenient for the majority of applications, but there are also applications that need finer-grained control over what gets added to the history list. In such cases, the automatic addition of entered lines to the history list can be turned off by calling the `gl_automatic_history()` function.

If this function is called with its *enable* argument set to 0, `gl_get_line()` will not automatically archive subsequently entered lines. Automatic archiving can be reenabled at a later time by calling this function again, with its *enable* argument set to 1. While automatic history archiving is disabled, the calling application can use the `gl_append_history()` to append lines to the history list as needed.

The *line* argument specifies the line to be added to the history list. This must be a normal '\0 ' terminated string. If this string contains any newline characters, the line that gets archived in the history list will be terminated by the first of these. Otherwise it will be terminated by the '\0 ' terminator. If the line is longer than the maximum input line length that was specified when `new_GetLine()` was called, it will be truncated to the actual `gl_get_line()` line length when the line is recalled.

If successful, `gl_append_history()` returns 0. Otherwise it returns non-zero and sets `errno` to one of the following values.

`EINVAL` One of the arguments passed to `gl_append_history()` was `NULL`.

`ENOMEM` The specified line was longer than the allocated size of the history buffer (as specified when `new_GetLine()` was called), so it could not be archived.

A textual description of the error can optionally be obtained by calling `gl_error_message()`. Note that after such an error, the history list remains in a valid state to receive new history lines, so there is little harm in simply ignoring the return status of `gl_append_history()`.

Miscellaneous History Configuration

If you wish to change the size of the history buffer that was originally specified in the call to `new_GetLine()`, you can do so with the `gl_resize_history()` function.

The *histlen* argument specifies the new size in bytes, and if you specify this as 0, the buffer will be deleted.

As mentioned in the discussion of `new_GetLine()`, the number of lines that can be stored in the history buffer, depends on the lengths of the individual lines. For example, a 1000 byte buffer could equally store 10 lines of average length 100 bytes, or 20 lines of average length 50 bytes. Although the buffer is never expanded when new lines are added, a list of pointers into the buffer does get expanded when needed to accomodate the number of lines currently stored in the buffer. To place an upper limit on the number of lines in the buffer, and thus a ceiling on the amount of memory used in this list, you can call the `gl_limit_history()` function.

The *max_lines* should either be a positive number >= 0, specifying an upper limit on the number of lines in the buffer, or be -1 to cancel any previously specified limit. When a limit is in effect, only the *max_lines* most recently appended lines are kept in the buffer. Older lines are discarded.

To discard lines from the history buffer, use the `gl_clear_history()` function.

The *all_groups* argument tells the function whether to delete just the lines associated with the current history group (see `gl_group_history()`) or all historical lines in the buffer.

The `gl_toggle_history()` function allows you to toggle history on and off without losing the current contents of the history list.

Setting the *enable* argument to 0 turns off the history mechanism, and setting it to 1 turns it back on. When history is turned off, no new lines will be added to the history list, and history lookup key-bindings will act as though there is nothing in the history buffer.

Querying History Information

The configured state of the history list can be queried with the `gl_history_state()` function. On return, the status information is recorded in the variable pointed to by the *state* argument.

The `gl_range_of_history()` function returns the number and range of lines in the history list. The return values are recorded in the variable pointed to by the range argument. If the *nlines* member of this structure is greater than zero, then the oldest and newest members report the range of lines in the list, and *newest=oldest+nlines*-1. Otherwise they are both zero.

The `gl_size_of_history()` function returns the total size of the history buffer and the amount of the buffer that is currently occupied.

On return, the size information is recorded in the variable pointed to by the *size* argument.

Changing Terminals

The `new_GetLine()` constructor function assumes that input is to be read from `stdin` and output written to `stdout`. The following function allows you to switch to different input and output streams.

The *gl* argument is the object that was returned by `new_GetLine()`. The *input_fp* argument specifies the stream to read from, and *output_fp* specifies the stream to be written to. Only if both of these refer to a terminal, will interactive terminal input be enabled. Otherwise `gl_get_line()` will simply call `fgets()` to read command input. If both streams refer to a terminal, then they must refer to the same terminal, and the type of this terminal must be specified with the *term* argument. The value of the *term* argument is looked up in the terminal information database (`terminfo` or `termcap`), in order to determine which special control sequences are needed to control various aspects of the terminal. `new_GetLine()` for example, passes the return value of `getenv`("TERM") in this argument. Note that if one or both of *input_fp* and *output_fp* do not refer to a terminal, then it is legal to pass `NULL` instead of a terminal type.

Note that if you want to pass file descriptors to `gl_change_terminal()`, you can do this by creating `stdio` stream wrappers using the POSIX `fdopen(3C)` function.

External Event Handling

By default, `gl_get_line()` does not return until either a complete input line has been entered by the user, or an error occurs. In programs that need to watch for I/O from other sources than the terminal, there are two options.

- Use the functions described in the `gl_io_mode(3TECLA)` manual page to switch `gl_get_line()` into non-blocking server mode. In this mode, `gl_get_line()` becomes a non-blocking, incremental line-editing function that can safely be called from an external event loop. Although this is a very versatile method, it involves taking on some responsibilities that are normally performed behind the scenes by `gl_get_line()`.
- While `gl_get_line()` is waiting for keyboard input from the user, you can ask it to also watch for activity on arbitrary file descriptors, such as network sockets or pipes, and have it call functions of your choosing when activity is seen. This works on any system that has the select system call, which is most, if not all flavors of UNIX.

Registering a file descriptor to be watched by `gl_get_line()` involves calling the `gl_watch_fd()` function. If this returns non-zero, then it means that either your arguments are invalid, or that this facility is not supported on the host system.

The *fd* argument is the file descriptor to be watched. The event argument specifies what type of activity is of interest, chosen from the following enumerated values:

`GLFD_READ` Watch for the arrival of data to be read.

`GLFD_WRITE` Watch for the ability to write to the file descriptor without blocking.

`GLFD_URGENT` Watch for the arrival of urgent out-of-band data on the file descriptor.

The *callback* argument is the function to call when the selected activity is seen. It should be defined with the following macro, which is defined in libtecla.h.

```
#define GL_FD_EVENT_FN(fn) GlFdStatus (fn)(GetLine *gl, \\
                                   void *data, int fd, GlFdEvent event)
```

The data argument of the `gl_watch_fd()` function is passed to the callback function for its own use, and can point to anything you like, including `NULL`. The file descriptor and the event argument are also passed to the callback function, and this potentially allows the same callback function to be registered to more than one type of event and/or more than one file descriptor. The return value of the callback function should be one of the following values.

`GLFD_ABORT` Tell `gl_get_line()` to abort. When this happens, `gl_get_line()` returns `NULL`, and a following call to `gl_return_status()` will return `GLR_FDABORT`. Note that if the application needs `errno` always to have a meaningful value when `gl_get_line()` returns `NULL`, the callback function should set `errno` appropriately.

`GLFD_REFRESH` Redraw the input line then continue waiting for input. Return this if your callback wrote to the terminal.

`GLFD_CONTINUE` Continue to wait for input, without redrawing the line.

Note that before calling the callback, `gl_get_line()` blocks most signals and leaves its own signal handlers installed, so if you need to catch a particular signal you will need to both temporarily install your own signal handler, and unblock the signal. Be sure to re-block the signal (if it was originally blocked) and reinstate the original signal handler, if any, before returning.

Your callback should not try to read from the terminal, which is left in raw mode as far as input is concerned. You can write to the terminal as usual, since features like conversion of newline to carriage-return/linefeed are re-enabled while the callback is running. If your callback function does write to the terminal, be sure to output a newline first, and when your callback returns, tell `gl_get_line()` that the input line needs to be redrawn, by returning the `GLFD_REFRESH` status code.

To remove a callback function that you previously registered for a given file descriptor and event, simply call `gl_watch_fd()` with the same *fd* and *event* arguments, but with a *callback* argument of 0. The *data* argument is ignored in this case.

Setting An Inactivity Timeout

The `gl_inactivity_timeout()` function can be used to set or cancel an inactivity timeout. Inactivity in this case refers both to keyboard input, and to I/O on any file descriptors registered by prior and subsequent calls to `gl_watch_fd()`.

The timeout is specified in the form of an integral number of seconds and an integral number of nanoseconds, specified by the *sec* and *nsec* arguments, respectively. Subsequently, whenever no activity is seen for this time period, the function specified by the *callback* argument is called. The *data* argument of `gl_inactivity_timeout()` is passed to this callback function whenever it is invoked, and can thus be used to pass arbitrary application-specific information to the callback. The following macro is provided in `<libtecla.h>` for applications to use to declare and prototype timeout callback functions.

```
#define GL_TIMEOUT_FN(fn) GlAfterTimeout (fn)(GetLine *gl, void *data)
```

On returning, the application's callback is expected to return one of the following enumerators to tell `gl_get_line()` how to procede after the timeout has been handled by the callback.

`GLTO_ABORT` Tell `gl_get_line()` to abort. When this happens, `gl_get_line()` will return `NULL`, and a following call to `gl_return_status()` will return `GLR_TIMEOUT`. Note that if the application needs `errno` always to have a meaningful value when `gl_get_line()` returns `NULL`, the callback function should set `errno` appropriately.

`GLTO_REFRESH` Redraw the input line, then continue waiting for input. You should return this value if your callback wrote to the terminal.

`GLTO_CONTINUE` In normal blocking-I/O mode, continue to wait for input, without redrawing the user's input line. In non-blocking server I/O mode (see `gl_io_mode(3TECLA)`), `gl_get_line()` acts as though I/O blocked. This means that `gl_get_line()` will immediately return `NULL`, and a following call to `gl_return_status()` will return `GLR_BLOCKED`.

Note that before calling the callback, `gl_get_line()` blocks most signals and leaves its own signal handlers installed, so if you need to catch a particular signal you will need to both temporarily install your own signal handler and unblock the signal. Be sure to re-block the signal (if it was originally blocked) and reinstate the original signal handler, if any, before returning.

Your callback should not try to read from the terminal, which is left in raw mode as far as input is concerned. You can however write to the terminal as usual, since features like conversion of newline to carriage-return/linefeed are re-enabled while the callback is running.

If your callback function does write to the terminal, be sure to output a newline first, and when your callback returns, tell `gl_get_line()` that the input line needs to be redrawn, by returning the `GLTO_REFRESH` status code.

Finally, note that although the timeout arguments include a nanosecond component, few computer clocks presently have resolutions that are finer than a few milliseconds, so asking for less than a few milliseconds is equivalent to requesting zero seconds on many systems. If this would be a problem, you should base your timeout selection on the actual resolution of the host clock (for example, by calling `sysconf(_SC_CLK_TCK)`).

To turn off timeouts, simply call `gl_inactivity_timeout()` with a *callback* argument of 0. The *data* argument is ignored in this case.

Signal Handling Defaults

By default, the `gl_get_line()` function intercepts a number of signals. This is particularly important for signals that would by default terminate the process, since the terminal needs to be restored to a usable state before this happens. This section describes the signals that are trapped by default and how `gl_get_line()` responds to them. Changing these defaults is the topic of the following section.

When the following subset of signals are caught, `gl_get_line()` first restores the terminal settings and signal handling to how they were before `gl_get_line()` was called, resends the signal to allow the calling application's signal handlers to handle it, then, if the process still exists, returns `NULL` and sets `errno` as specified below.

`SIGINT` This signal is generated both by the keyboard interrupt key (usually `^C`), and the keyboard break key. The `errno` value is `EINTR`.

`SIGHUP` This signal is generated when the controlling terminal exits. The `errno` value is `ENOTTY`.

`SIGPIPE` This signal is generated when a program attempts to write to a pipe whose remote end is not being read by any process. This can happen for example if you have called `gl_change_terminal()` to redirect output to a pipe hidden under a pseudo terminal. The `errno` value is `EPIPE`.

`SIGQUIT` This signal is generated by the keyboard quit key (usually `^\\`). The `errno` value is `EINTR`.

`SIGABRT` This signal is generated by the standard C, abort function. By default it both terminates the process and generates a core dump. The `errno` value is `EINTR`.

`SIGTERM` This is the default signal that the UNIX kill command sends to processes. The `errno` value is `EINTR`.

Note that in the case of all of the above signals, POSIX mandates that by default the process is terminated, with the addition of a core dump in the case of the `SIGQUIT` signal. In other words,

if the calling application does not override the default handler by supplying its own signal handler, receipt of the corresponding signal will terminate the application before `gl_get_line()` returns.

If `gl_get_line()` aborts with `errno` set to `EINTR`, you can find out what signal caused it to abort, by calling the `gl_last_signal()` function. This returns the numeric code (for example, `SIGINT`) of the last signal that was received during the most recent call to `gl_get_line()`, or -1 if no signals were received.

On systems that support it, when a `SIGWINCH` (window change) signal is received, `gl_get_line()` queries the terminal to find out its new size, redraws the current input line to accomodate the new size, then returns to waiting for keyboard input from the user. Unlike other signals, this signal is not resent to the application.

Finally, the following signals cause `gl_get_line()` to first restore the terminal and signal environment to that which prevailed before `gl_get_line()` was called, then resend the signal to the application. If the process still exists after the signal has been delivered, then `gl_get_line()` then re-establishes its own signal handlers, switches the terminal back to raw mode, redisplays the input line, and goes back to awaiting terminal input from the user.

`SIGCONT` This signal is generated when a suspended process is resumed.

`SIGPOLL` On SVR4 systems, this signal notifies the process of an asynchronous I/O event. Note that under 4.3+BSD, `SIGIO` and `SIGPOLL` are the same. On other systems, `SIGIO` is ignored by default, so `gl_get_line()` does not trap it by default.

`SIGPWR` This signal is generated when a power failure occurs (presumably when the system is on a UPS).

`SIGALRM` This signal is generated when a timer expires.

`SIGUSR1` An application specific signal.

`SIGUSR2` Another application specific signal.

`SIGVTALRM` This signal is generated when a virtual timer expires. See `setitimer(2)`.

`SIGXCPU` This signal is generated when a process exceeds its soft CPU time limit.

`SIGXFSZ` This signal is generated when a process exceeds its soft file-size limit.

`SIGTSTP` This signal is generated by the terminal suspend key, which is usually `^Z`, or the delayed terminal suspend key, which is usually `^Y`.

`SIGTTIN` This signal is generated if the program attempts to read from the terminal while the program is running in the background.

`SIGTTOU` This signal is generated if the program attempts to write to the terminal while the program is running in the background.

Obviously not all of the above signals are supported on all systems, so code to support them is conditionally compiled into the tecla library.

Note that if SIGKILL or SIGPOLL, which by definition cannot be caught, or any of the hardware generated exception signals, such as SIGSEGV, SIGBUS, and SIGFPE, are received and unhandled while gl_get_line() has the terminal in raw mode, the program will be terminated without the terminal having been restored to a usable state. In practice, job-control shells usually reset the terminal settings when a process relinquishes the controlling terminal, so this is only a problem with older shells.

Customized Signal Handling

The previous section listed the signals that gl_get_line() traps by default, and described how it responds to them. This section describes how to both add and remove signals from the list of trapped signals, and how to specify how gl_get_line() should respond to a given signal.

If you do not need gl_get_line() to do anything in response to a signal that it normally traps, you can tell to gl_get_line() to ignore that signal by calling gl_ignore_signal().

The *signo* argument is the number of the signal (for example, SIGINT) that you want to have ignored. If the specified signal is not currently one of those being trapped, this function does nothing.

The gl_trap_signal() function allows you to either add a new signal to the list that gl_get_line() traps or modify how it responds to a signal that it already traps.

The *signo* argument is the number of the signal that you want to have trapped. The *flags* argument is a set of flags that determine the environment in which the application's signal handler is invoked. The *after* argument tells gl_get_line() what to do after the application's signal handler returns. The *errno_value* tells gl_get_line() what to set errno to if told to abort.

The *flags* argument is a bitwise OR of zero or more of the following enumerators:

GLS_RESTORE_SIG	Restore the caller's signal environment while handling the signal.
GLS_RESTORE_TTY	Restore the caller's terminal settings while handling the signal.
GLS_RESTORE_LINE	Move the cursor to the start of the line following the input line before invoking the application's signal handler.
GLS_REDRAW_LINE	Redraw the input line when the application's signal handler returns.
GLS_UNBLOCK_SIG	Normally, if the calling program has a signal blocked (see sigprocmask(2)), gl_get_line() does not trap that signal. This flag tells gl_get_line() to trap the signal and unblock it for the duration of the call to gl_get_line().

GLS_DONT_FORWARD — If this flag is included, the signal will not be forwarded to the signal handler of the calling program.

Two commonly useful flag combinations are also enumerated as follows:

GLS_RESTORE_ENV — GLS_RESTORE_SIG | GLS_RESTORE_TTY |GLS_REDRAW_LINE

GLS_SUSPEND_INPUT — GLS_RESTORE_ENV | GLS_RESTORE_LINE

If your signal handler, or the default system signal handler for this signal, if you have not overridden it, never either writes to the terminal, nor suspends or terminates the calling program, then you can safely set the *flags* argument to 0.

- The cursor does not get left in the middle of the input line.
- So that the user can type in input and have it echoed.
- So that you do not need to end each output line with \r\n, instead of just \n.

The GL_RESTORE_ENV combination is the same as GL_SUSPEND_INPUT, except that it does not move the cursor. If your signal handler does not read or write anything to the terminal, the user will not see any visible indication that a signal was caught. This can be useful if you have a signal handler that only occasionally writes to the terminal, where using GL_SUSPEND_LINE would cause the input line to be unnecessarily duplicated when nothing had been written to the terminal. Such a signal handler, when it does write to the terminal, should be sure to start a new line at the start of its first write, by writing a new line before returning. If the signal arrives while the user is entering a line that only occupies a signal terminal line, or if the cursor is on the last terminal line of a longer input line, this will have the same effect as GL_SUSPEND_INPUT. Otherwise it will start writing on a line that already contains part of the displayed input line. This does not do any harm, but it looks a bit ugly, which is why the GL_SUSPEND_INPUT combination is better if you know that you are always going to be writting to the terminal.

The *after* argument, which determines what gl_get_line() does after the application's signal handler returns (if it returns), can take any one of the following values:

GLS_RETURN — Return the completed input line, just as though the user had pressed the return key.

GLS_ABORT — Cause gl_get_line() to abort. When this happens, gl_get_line() returns NULL, and a following call to gl_return_status() will return GLR_SIGNAL. Note that if the application needs errno always to have a meaningful value when gl_get_line() returns NULL, the callback function should set errno appropriately.

GLS_CONTINUE — Resume command line editing.

The *errno_value* argument is intended to be combined with the GLS_ABORT option, telling gl_get_line() what to set the standard errno variable to before returning NULL to the calling

program. It can also, however, be used with the GL_RETURN option, in case you want to have a way to distinguish between an input line that was entered using the return key, and one that was entered by the receipt of a signal.

Reliable Signal Handling

Signal handling is suprisingly hard to do reliably without race conditions. In gl_get_line() a lot of care has been taken to allow applications to perform reliable signal handling around gl_get_line(). This section explains how to make use of this.

As an example of the problems that can arise if the application is not written correctly, imagine that one's application has a SIGINT signal handler that sets a global flag. Now suppose that the application tests this flag just before invoking gl_get_line(). If a SIGINT signal happens to be received in the small window of time between the statement that tests the value of this flag, and the statement that calls gl_get_line(), then gl_get_line() will not see the signal, and will not be interrupted. As a result, the application will not be able to respond to the signal until the user gets around to finishing entering the input line and gl_get_line() returns. Depending on the application, this might or might not be a disaster, but at the very least it would puzzle the user.

The way to avoid such problems is to do the following.

1. If needed, use the gl_trap_signal() function to configure gl_get_line() to abort when important signals are caught.
2. Configure gl_get_line() such that if any of the signals that it catches are blocked when gl_get_line() is called, they will be unblocked automatically during times when gl_get_line() is waiting for I/O. This can be done either on a per signal basis, by calling the gl_trap_signal() function, and specifying the GLS_UNBLOCK attribute of the signal, or globally by calling the gl_catch_blocked() function. This function simply adds the GLS_UNBLOCK attribute to all of the signals that it is currently configured to trap.
3. Just before calling gl_get_line(), block delivery of all of the signals that gl_get_line() is configured to trap. This can be done using the POSIX sigprocmask function in conjunction with the gl_list_signals() function. This function returns the set of signals that it is currently configured to catch in the set argument, which is in the form required by sigprocmask(2).
4. In the example, one would now test the global flag that the signal handler sets, knowing that there is now no danger of this flag being set again until gl_get_line() unblocks its signals while performing I/O.
5. Eventually gl_get_line() returns, either because a signal was caught, an error occurred, or the user finished entering their input line.
6. Now one would check the global signal flag again, and if it is set, respond to it, and zero the flag.
7. Use sigprocmask() to unblock the signals that were blocked in step 3.

The same technique can be used around certain POSIX signal-aware functions, such as sigsetjmp(3C) and sigsuspend(2), and in particular, the former of these two functions can be used in conjunction with siglongjmp(3C) to implement race-condition free signal handling around other long-running system calls. The gl_get_line() function manages to reliably trap signals around calls to functions like read(2) and select(3C) without race conditions.

The gl_get_line() function first uses the POSIX sigprocmask() function to block the delivery of all of the signals that it is currently configured to catch. This is redundant if the application has already blocked them, but it does no harm. It undoes this step just before returning.

Whenever gl_get_line() needs to call read or select to wait for input from the user, it first calls the POSIX sigsetjmp() function, being sure to specify a non-zero value for its *savemask* argument.

If sigsetjmp() returns zero, gl_get_line() then does the following.

1. It uses the POSIX sigaction(2) function to register a temporary signal handler to all of the signals that it is configured to catch. This signal handler does two things.
   a. It records the number of the signal that was received in a file-scope variable.
   b. It then calls the POSIX siglongjmp() function using the buffer that was passed to sigsetjmp() for its first argument and a non-zero value for its second argument.

   When this signal handler is registered, the *sa_mask* member of the struct sigaction *act* argument of the call to sigaction() is configured to contain all of the signals that gl_get_line() is catching. This ensures that only one signal will be caught at once by our signal handler, which in turn ensures that multiple instances of our signal handler do not tread on each other's toes.
2. Now that the signal handler has been set up, gl_get_line() unblocks all of the signals that it is configured to catch.
3. It then calls the read() or select() function to wait for keyboard input.
4. If this function returns (that is, no signal is received), gl_get_line() blocks delivery of the signals of interest again.
5. It then reinstates the signal handlers that were displaced by the one that was just installed.

Alternatively, if sigsetjmp() returns non-zero, this means that one of the signals being trapped was caught while the above steps were executing. When this happens, gl_get_line() does the following.

First, note that when a call to siglongjmp() causes sigsetjmp() to return, provided that the *savemask* argument of sigsetjmp() was non-zero, the signal process mask is restored to how it was when sigsetjmp() was called. This is the important difference between sigsetjmp()

and the older problematic setjmp(3C), and is the essential ingredient that makes it possible to avoid signal handling race conditions. Because of this we are guaranteed that all of the signals that we blocked before calling sigsetjmp() are blocked again as soon as any signal is caught. The following statements, which are then executed, are thus guaranteed to be executed without any further signals being caught.

1. If so instructed by the gl_get_line() configuration attributes of the signal that was caught, gl_get_line() restores the terminal attributes to the state that they had when gl_get_line() was called. This is particularly important for signals that suspend or terminate the process, since otherwise the terminal would be left in an unusable state.
2. It then reinstates the application's signal handlers.
3. Then it uses the C standard-library raise(3C) function to re-send the application the signal that was caught.
4. Next it unblocks delivery of the signal that we just sent. This results in the signal that was just sent by raise() being caught by the application's original signal handler, which can now handle it as it sees fit.
5. If the signal handler returns (that is, it does not terminate the process), gl_get_line() blocks delivery of the above signal again.
6. It then undoes any actions performed in the first of the above steps and redisplays the line, if the signal configuration calls for this.
7. gl_get_line() then either resumes trying to read a character, or aborts, depending on the configuration of the signal that was caught.

What the above steps do in essence is to take asynchronously delivered signals and handle them synchronously, one at a time, at a point in the code where gl_get_line() has complete control over its environment.

The Terminal Size

On most systems the combination of the TIOCGWINSZ ioctl and the SIGWINCH signal is used to maintain an accurate idea of the terminal size. The terminal size is newly queried every time that gl_get_line() is called and whenever a SIGWINCH signal is received.

On the few systems where this mechanism is not available, at startup new_GetLine() first looks for the LINES and COLUMNS environment variables. If these are not found, or they contain unusable values, then if a terminal information database like terminfo or termcap is available, the default size of the terminal is looked up in this database. If this too fails to provide the terminal size, a default size of 80 columns by 24 lines is used.

Even on systems that do support ioctl(TIOCGWINSZ), if the terminal is on the other end of a serial line, the terminal driver generally has no way of detecting when a resize occurs or of querying what the current size is. In such cases no SIGWINCH is sent to the process, and the dimensions returned by ioctl(TIOCGWINSZ) are not correct. The only way to handle such

instances is to provide a way for the user to enter a command that tells the remote system what the new size is. This command would then call the `gl_set_term_size()` function to tell `gl_get_line()` about the change in size.

The *ncolumn* and *nline* arguments are used to specify the new dimensions of the terminal, and must not be less than 1. On systems that do support ioctl(`TIOCGWINSZ`), this function first calls ioctl(`TIOCSWINSZ`) to tell the terminal driver about the change in size. In non-blocking server-I/O mode, if a line is currently being input, the input line is then redrawn to accomodate the changed size. Finally the new values are recorded in *gl* for future use by `gl_get_line()`.

The `gl_terminal_size()` function allows you to query the current size of the terminal, and install an alternate fallback size for cases where the size is not available. Beware that the terminal size will not be available if reading from a pipe or a file, so the default values can be important even on systems that do support ways of finding out the terminal size.

This function first updates `gl_get_line()`'s fallback terminal dimensions, then records its findings in the return value.

The *def_ncolumn* and *def_nline* arguments specify the default number of terminal columns and lines to use if the terminal size cannot be determined by ioctl(`TIOCGWINSZ`) or environment variables.

Hiding What You Type

When entering sensitive information, such as passwords, it is best not to have the text that you are entering echoed on the terminal. Furthermore, such text should not be recorded in the history list, since somebody finding your terminal unattended could then recall it, or somebody snooping through your directories could see it in your history file. With this in mind, the `gl_echo_mode()` function allows you to toggle on and off the display and archival of any text that is subsequently entered in calls to `gl_get_line()`.

The *enable* argument specifies whether entered text should be visible or not. If it is 0, then subsequently entered lines will not be visible on the terminal, and will not be recorded in the history list. If it is 1, then subsequent input lines will be displayed as they are entered, and provided that history has not been turned off with a call to `gl_toggle_history()`, then they will also be archived in the history list. Finally, if the enable argument is -1, then the echoing mode is left unchanged, which allows you to non-destructively query the current setting through the return value. In all cases, the return value of the function is 0 if echoing was disabled before the function was called, and 1 if it was enabled.

When echoing is turned off, note that although tab completion will invisibly complete your prefix as far as possible, ambiguous completions will not be displayed.

Single Character Queries

Using `gl_get_line()` to query the user for a single character reply, is inconvenient for the user, since they must hit the enter or return key before the character that they typed is returned to the program. Thus the `gl_query_char()` function has been provided for single character queries like this.

This function displays the specified prompt at the start of a new line, and waits for the user to type a character. When the user types a character, `gl_query_char()` displays it to the right of the prompt, starts a newline, then returns the character to the calling program. The return value of the function is the character that was typed. If the read had to be aborted for some reason, EOF is returned instead. In the latter case, the application can call the previously documented `gl_return_status()`, to find out what went wrong. This could, for example, have been the reception of a signal, or the optional inactivity timer going off.

If the user simply hits enter, the value of the *defchar* argument is substituted. This means that when the user hits either newline or return, the character specified in *defchar*, is displayed after the prompt, as though the user had typed it, as well as being returned to the calling application. If such a replacement is not important, simply pass '\n' as the value of *defchar*.

If the entered character is an unprintable character, it is displayed symbolically. For example, control-A is displayed as `^A`, and characters beyond 127 are displayed in octal, preceded by a backslash.

As with `gl_get_line()`, echoing of the entered character can be disabled using the `gl_echo_mode()` function.

If the calling process is suspended while waiting for the user to type their response, the cursor is moved to the line following the prompt line, then when the process resumes, the prompt is redisplayed, and `gl_query_char()` resumes waiting for the user to type a character.

Note that in non-blocking server mode, if an incomplete input line is in the process of being read when `gl_query_char()` is called, the partial input line is discarded, and erased from the terminal, before the new prompt is displayed. The next call to `gl_get_line()` will thus start editing a new line.

Reading Raw Characters

Whereas the `gl_query_char()` function visibly prompts the user for a character, and displays what they typed, the `gl_read_char()` function reads a signal character from the user, without writing anything to the terminal, or perturbing any incompletely entered input line. This means that it can be called not only from between calls to `gl_get_line()`, but also from callback functions that the application has registered to be called by `gl_get_line()`.

On success, the return value of `gl_read_char()` is the character that was read. On failure, EOF is returned, and the `gl_return_status()` function can be called to find out what went wrong. Possibilities include the optional inactivity timer going off, the receipt of a signal that is configured to abort `gl_get_line()`, or terminal I/O blocking, when in non-blocking server-I/O mode.

Beware that certain keyboard keys, such as function keys, and cursor keys, usually generate at least three characters each, so a single call to `gl_read_char()` will not be enough to identify such keystrokes.

Clearing The Terminal

The calling program can clear the terminal by calling `gl_erase_terminal()`. In non-blocking server-I/O mode, this function also arranges for the current input line to be redrawn from scratch when `gl_get_line()` is next called.

Displaying Text Dynamically

Between calls to `gl_get_line()`, the `gl_display_text()` function provides a convenient way to display paragraphs of text, left-justified and split over one or more terminal lines according to the constraints of the current width of the terminal. Examples of the use of this function may be found in the demo programs, where it is used to display introductions. In those examples the advanced use of optional prefixes, suffixes and filled lines to draw a box around the text is also illustrated.

If *gl* is not currently connected to a terminal, for example if the output of a program that uses `gl_get_line()` is being piped to another program or redirected to a file, then the value of the *def_width* parameter is used as the terminal width.

The *indentation* argument specifies the number of characters to use to indent each line of ouput. The *fill_char* argument specifies the character that will be used to perform this indentation.

The *prefix* argument can be either `NULL` or a string to place at the beginning of each new line (after any indentation). Similarly, the *suffix* argument can be either `NULL` or a string to place at the end of each line. The suffix is placed flush against the right edge of the terminal, and any space between its first character and the last word on that line is filled with the character specified by the *fill_char* argument. Normally the fill-character is a space.

The *start* argument tells `gl_display_text()` how many characters have already been written to the current terminal line, and thus tells it the starting column index of the cursor. Since the return value of `gl_display_text()` is the ending column index of the cursor, by passing the return value of one call to the start argument of the next call, a paragraph that is broken between more than one string can be composed by calling `gl_display_text()` for each successive portion of the paragraph. Note that literal newline characters are necessary at the end of each paragraph to force a new line to be started.

On error, `gl_display_text()` returns -1.

Callback Function Facilities

Unless otherwise stated, callback functions such as tab completion callbacks and event callbacks should not call any functions in this module. The following functions, however, are designed specifically to be used by callback functions.

Calling the `gl_replace_prompt()` function from a callback tells `gl_get_line()` to display a different prompt when the callback returns. Except in non-blocking server mode, it has no effect if used between calls to `gl_get_line()`. In non-blocking server mode, when used

between two calls to `gl_get_line()` that are operating on the same input line, the current input line will be re-drawn with the new prompt on the following call to `gl_get_line()`.

International Character Sets

Since libtecla(3LIB) version 1.4.0, `gl_get_line()` has been 8-bit clean. This means that all 8-bit characters that are printable in the user's current locale are now displayed verbatim and included in the returned input line. Assuming that the calling program correctly contains a call like the following,

```
setlocale(LC_CTYPE, "")
```

then the current locale is determined by the first of the environment variables `LC_CTYPE`, `LC_ALL`, and `LANG` that is found to contain a valid locale name. If none of these variables are defined, or the program neglects to call setlocale(3C), then the default C locale is used, which is US 7-bit ASCII. On most UNIX-like platforms, you can get a list of valid locales by typing the command:

```
locale -a
```

at the shell prompt. Further documentation on how the user can make use of this to enter international characters can be found in the tecla(5) man page.

Thread Safety

Unfortunately neither `terminfo` nor `termcap` were designed to be reentrant, so you cannot safely use the functions of the getline module in multiple threads (you can use the separate file-expansion and word-completion modules in multiple threads, see the corresponding man pages for details). However due to the use of POSIX reentrant functions for looking up home directories, it is safe to use this module from a single thread of a multi-threaded program, provided that your other threads do not use any `termcap` or `terminfo` functions.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also

cpl_complete_word(3TECLA), ef_expand_file(3TECLA), gl_io_mode(3TECLA), libtecla(3LIB), pca_lookup_file(3TECLA), attributes(5), tecla(5)

Name gl_io_mode, gl_raw_io, gl_normal_io, gl_tty_signals, gl_abandon_line, gl_handle_signal, gl_pending_io – use `gl_get_line()` from an external event loop

Synopsis

```
cc [ flag... ] file... -ltecla [ library... ]
#include <libtecla.h>

int gl_io_mode(GetLine *gl, GlIOMode mode);

int gl_raw_io(GetLine *gl);

int gl_normal_io(GetLine *gl);

int gl_tty_signals(void (*term_handler)(int), void (*susp_handler)(int),
     void (*cont_handler)(int), void (*size_handler)(int));

void gl_abandon_line(GetLine *gl);

void gl_handle_signal(int signo, GetLine *gl, int ngl);

GlPendingIO gl_pending_io(GetLine *gl);
```

Description The `gl_get_line(3TECLA)` function supports two different I/O modes. These are selected by calling the `gl_io_mode()` function. The *mode* argument of `gl_io_mode()` specifies the new I/O mode and must be one of the following.

`GL_NORMAL_MODE` Select the normal blocking-I/O mode. In this mode `gl_get_line()` does not return until either an error occurs of the user finishes entering a new line.

`GL_SERVER_MODE` Select non-blocking server I/O mode. In this mode, since non-blocking terminal I/O is used, the entry of each new input line typically requires many calls to `gl_get_line()` from an external I/O-driven event loop.

Newly created GetLine objects start in normal I/O mode, so to switch to non-blocking server mode requires an initial call to `gl_io_mode()`.

Server I/O Mode In non-blocking server I/O mode, the application is required to have an event loop that calls `gl_get_line()` whenever the terminal file descriptor can perform the type I/O that `gl_get_line()` is waiting for. To determine which type of I/O `gl_get_line()` is waiting for, the application calls the `gl_pending_io()` function. The return value is one of the following two enumerated values.

`GLP_READ` `gl_get_line()` is waiting to write a character to the terminal.

`GLP_WRITE` `gl_get_line()` is waiting to read a character from the keyboad.

If the application is using either the `select(3C)` or `poll(2)` function to watch for I/O on a group of file descriptors, then it should call the `gl_pending_io()` function before each call to these functions to determine which direction of I/O it should tell them to watch for, and configure their arguments accordingly. In the case of the `select()` function, this means using

the FD_SET() macro to add the terminal file descriptor either to the set of file descriptors to be watched for readability or the set to be watched for writability.

As in normal I/O mode, the return value of gl_get_line() is either a pointer to a completed input line or NULL. However, whereas in normal I/O mode a NULL return value always means that an error occurred, in non-blocking server mode, NULL is also returned when gl_get_line() cannot read or write to the terminal without blocking. Thus in non-blocking server mode, in order to determine when a NULL return value signifies that an error occurred or not, it is necessary to call the gl_return_status() function. If this function returns the enumerated value GLR_BLOCKED, gl_get_line() is waiting for I/O and no error has occurred.

When gl_get_line() returns NULL and gl_return_status() indicates that this is due to blocked terminal I/O, the application should call gl_get_line() again when the type of I/O reported by gl_pending_io() becomes possible. The *prompt*, *start_line* and *start_pos* arguments of gl_get_line() will be ignored on these calls. If you need to change the prompt of the line that is currently being edited, you can call the gl_replace_prompt(3TECLA) function between calls to gl_get_line().

Giving Up The Terminal

A complication that is unique to non-blocking server mode is that it requires that the terminal be left in raw mode between calls to gl_get_line(). If this were not the case, the external event loop would not be able to detect individual key-presses, and the basic line editing implemented by the terminal driver would clash with the editing provided by gl_get_line(). When the terminal needs to be used for purposes other than entering a new input line with gl_get_line(), it needs to be restored to a usable state. In particular, whenever the process is suspended or terminated, the terminal must be returned to a normal state. If this is not done, then depending on the characteristics of the shell that was used to invoke the program, the user could end up with a hung terminal. To this end, the gl_normal_io() function is provided for switching the terminal back to the state that it was in when raw mode was last established.

The gl_normal_io() function first flushes any pending output to the terminal, then moves the cursor to the start of the terminal line which follows the end of the incompletely entered input line. At this point it is safe to suspend or terminate the process, and it is safe for the application to read and write to the terminal. To resume entry of the input line, the application should call the gl_raw_io() function.

The gl_normal_io() function starts a new line, redisplays the partially completed input line (if any), restores the cursor position within this line to where it was when gl_normal_io() was called, then switches back to raw, non-blocking terminal mode ready to continue entry of the input line when gl_get_line() is next called.

Note that in non-blocking server mode, if gl_get_line() is called after a call to gl_normal_io(), without an intervening call to gl_raw_io(), gl_get_line() will call gl_raw_mode() itself, and the terminal will remain in this mode when gl_get_line() returns.

Signal Handling

In the previous section it was pointed out that in non-blocking server mode, the terminal must be restored to a sane state whenever a signal is received that either suspends or terminates the process. In normal I/O mode, this is done for you by `gl_get_line()`, but in non-blocking server mode, since the terminal is left in raw mode between calls to `gl_get_line()`, this signal handling has to be done by the application. Since there are many signals that can suspend or terminate a process, as well as other signals that are important to `gl_get_line()`, such as the `SIGWINCH` signal, which tells it when the terminal size has changed, the `gl_tty_signals()` function is provided for installing signal handlers for all pertinent signals.

The `gl_tty_signals()` function uses `gl_get_line()`'s internal list of signals to assign specified signal handlers to groups of signals. The arguments of this function are as follows.

term_handler This is the signal handler that is used to trap signals that by default terminate any process that receives them (for example, `SIGINT` or `SIGTERM`).

susp_handler This is the signal handler that is used to trap signals that by default suspend any process that receives them, (for example, `SIGTSTP` or `SIGTTOU`).

cont_handler This is the signal handler that is used to trap signals that are usually sent when a process resumes after being suspended (usually `SIGCONT`). Beware that there is nothing to stop a user from sending one of these signals at other times.

size_handler This signal handler is used to trap signals that are sent to processes when their controlling terminals are resized by the user (for example, `SIGWINCH`).

These arguments can all be the same, if so desired, and `SIG_IGN` (ignore this signal) or `SIG_DFL` (use the system-provided default signal handler) can be specified instead of a function where pertinent. In particular, it is rarely useful to trap `SIGCONT`, so the *cont_handler* argument will usually be `SIG_DFL` or `SIG_IGN`.

The `gl_tty_signals()` function uses the POSIX `sigaction(2)` function to install these signal handlers, and it is careful to use the *sa_mask* member of each `sigaction` structure to ensure that only one of these signals is ever delivered at a time. This guards against different instances of these signal handlers from simultaneously trying to write to common global data, such as a shared `sigsetjmp(3C)` buffer or a signal-received flag. The signal handlers installed by this function should call the `gl_handle_signal()`.

The *signo* argument tells this function which signal it is being asked to respond to, and the *gl* argument should be a pointer to the first element of an array of *ngl* `GetLine` objects. If your application has only one of these objects, pass its pointer as the *gl* argument and specify *ngl* as 1.

Depending on the signal that is being handled, this function does different things.

Process termination signals

If the signal that was caught is one of those that by default terminates any process that receives it, then `gl_handle_signal()` does the following steps.

1. First it blocks the delivery of all signals that can be blocked (ie. `SIGKILL` and `SIGSTOP` cannot be blocked).
2. Next it calls `gl_normal_io()` for each of the ngl GetLine objects. Note that this does nothing to any of the GetLine objects that are not currently in raw mode.
3. Next it sets the signal handler of the signal to its default, process-termination disposition.
4. Next it re-sends the process the signal that was caught.
5. Finally it unblocks delivery of this signal, which results in the process being terminated.

Process suspension signals

If the default disposition of the signal is to suspend the process, the same steps are executed as for process termination signals, except that when the process is later resumed, `gl_handle_signal()` continues, and does the following steps.

1. It re-blocks delivery of the signal.
2. It reinstates the signal handler of the signal to the one that was displaced when its default disposition was substituted.
3. For any of the GetLine objects that were in raw mode when `gl_handle_signal()` was called, `gl_handle_signal()` then calls `gl_raw_io()`, to resume entry of the input lines on those terminals.
4. Finally, it restores the signal process mask to how it was when `gl_handle_signal()` was called.

Note that the process is suspended or terminated using the original signal that was caught, rather than using the uncatchable `SIGSTOP` and `SIGKILL` signals. This is important, because when a process is suspended or terminated, the parent of the process may wish to use the status value returned by the wait system call to figure out which signal was responsible. In particular, most shells use this information to print a corresponding message to the terminal. Users would be rightly confused if when their process received a `SIGPIPE` signal, the program responded by sending itself a `SIGKILL` signal, and the shell then printed out the provocative statement, "Killed!".

Interrupting The Event Loop

If a signal is caught and handled when the application's event loop is waiting in `select()` or `poll()`, these functions will be aborted with `errno` set to `EINTR`. When this happens the event loop should call `gl_pending_io()` before calling `select()` or `poll()` again. It should then arrange for `select()` or `poll()` to wait for the type of I/O that `gl_pending_io()` reports. This is necessary because any signal handler that calls `gl_handle_signal()` will frequently change the type of I/O that `gl_get_line()` is waiting for.

If a signal arrives between the statements that configure the arguments of `select()` or `poll()` and the calls to these functions, the signal will not be seen by these functions, which will then not be aborted. If these functions are waiting for keyboard input from the user when the signal

is received, and the signal handler arranges to redraw the input line to accommodate a terminal resize or the resumption of the process. This redisplay will be delayed until the user presses the next key. Apart from puzzling the user, this clearly is not a serious problem. However there is a way, albeit complicated, to completely avoid this race condition. The following steps illustrate this.

1. Block all of the signals that `gl_get_line()` catches, by passing the signal set returned by `gl_list_signals()` to `sigprocmask(2)`.
2. Call `gl_pending_io()` and set up the arguments of `select()` or `poll()` accordingly.
3. Call `sigsetjmp(3C)` with a non-zero *savemask* argument.
4. Initially this `sigsetjmp()` statement will return zero, indicating that control is not resuming there after a matching call to `siglongjmp(3C)`.
5. Replace all of the handlers of the signals that `gl_get_line()` is configured to catch, with a signal handler that first records the number of the signal that was caught, in a file-scope variable, then calls `siglongjmp()` with a non-zero *val* argument, to return execution to the above `sigsetjmp()` statement. Registering these signal handlers can conveniently be done using the `gl_tty_signals()` function.
6. Set the file-scope variable that the above signal handler uses to record any signal that is caught to -1, so that we can check whether a signal was caught by seeing if it contains a valid signal number.
7. Now unblock the signals that were blocked in step 1. Any signal that was received by the process in between step 1 and now will now be delivered, and trigger our signal handler, as will any signal that is received until we block these signals again.
8. Now call `select()` or `poll()`.
9. When select returns, again block the signals that were unblocked in step 7.

   If a signal is arrived any time during the above steps, our signal handler will be triggered and cause control to return to the `sigsetjmp()` statement, where this time, `sigsetjmp()` will return non-zero, indicating that a signal was caught. When this happens we simply skip the above block of statements, and continue with the following statements, which are executed regardless of whether or not a signal is caught. Note that when `sigsetjmp()` returns, regardless of why it returned, the process signal mask is returned to how it was when `sigsetjmp()` was called. Thus the following statements are always executed with all of our signals blocked.
10. Reinstate the signal handlers that were displaced in step 5.
11. Check wether a signal was caught, by checking the file-scope variable that the signal handler records signal numbers in.
12. If a signal was caught, send this signal to the application again and unblock only this signal so that it invokes the signal handler which was just reinstated in step 10.
13. Unblock all of the signals that were blocked in step 7.

Signals Caught By `gl_get_line()`

Since the application is expected to handle signals in non-blocking server mode, `gl_get_line()` does not attempt to duplicate this when it is being called. If one of the signals that it is configured to catch is sent to the application while `gl_get_line()` is being called, `gl_get_line()` reinstates the caller's signal handlers, then immediately before returning, re-sends the signal to the process to let the application's signal handler handle it. If the process is not terminated by this signal, `gl_get_line()` returns `NULL`, and a following call to `gl_return_status()` returns the enumerated value `GLR_SIGNAL`.

Aborting Line Input

Often, rather than letting it terminate the process, applications respond to the `SIGINT` user-interrupt signal by aborting the current input line. This can be accomplished in non-blocking server-I/O mode by not calling `gl_handle_signal()` when this signal is caught, but by calling instead the `gl_abandon_line()` function. This function arranges that when `gl_get_line()` is next called, it first flushes any pending output to the terminal, discardes the current input line, outputs a new prompt on the next line, and finally starts accepting input of a new input line from the user.

Signal Safe Functions

Provided that certain rules are followed, the `gl_normal_io()`, `gl_raw_io()`, `gl_handle_signal()`, and `gl_abandon_line()` functions can be written to be safely callable from signal handlers. Other functions in this library should not be called from signal handlers. For this to be true, all signal handlers that call these functions must be registered in such a way that only one instance of any one of them can be running at one time. The way to do this is to use the POSIX `sigaction()` function to register all signal handlers, and when doing this, use the *sa_mask* member of the corresponding `sigaction` structure to indicate that all of the signals whose handlers invoke the above functions should be blocked when the current signal is being handled. This prevents two signal handlers from operating on a `GetLine` object at the same time.

To prevent signal handlers from accessing a `GetLine` object while `gl_get_line()` or any of its associated public functions are operating on it, all public functions associated with `gl_get_line()`, including `gl_get_line()` itself, temporarily block the delivery of signals when they are accessing `GetLine` objects. Beware that the only signals that they block are the signals that `gl_get_line()` is currently configured to catch, so be sure that if you call any of the above functions from signal handlers, that the signals that these handlers are assigned to are configured to be caught by `gl_get_line()`. See `gl_trap_signal(3TECLA)`.

Using Timeouts To Poll

If instead of using `select()` or `poll()` to wait for I/O your application needs only to get out of `gl_get_line()` periodically to briefly do something else before returning to accept input from the user, use the `gl_inactivity_timeout(3TECLA)` function in non-blocking server mode to specify that a callback function that returns `GLTO_CONTINUE` should be called whenever `gl_get_line()` has been waiting for I/O for more than a specified amount of time. When this callback is triggered, `gl_get_line()` will return `NULL` and a following call to `gl_return_status()` will return `GLR_BLOCKED`.

The `gl_get_line()` function will not return until the user has not typed a key for the specified interval, so if the interval is long and the user keeps typing, `gl_get_line()` might not return

for a while. There is no guarantee that it will return in the time specified.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also cpl_complete_word(3TECLA), ef_expand_file(3TECLA), gl_get_line(3TECLA), libtecla(3LIB), pca_lookup_file(3TECLA), attributes(5), tecla(5)

Name hextob, htobsl, htobclear – convert hexadecimal string to binary label

Synopsis

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

#include <tsol/label.h>

int htobsl(const char *s, m_label_t *label);

int htobclear(const char *s, m_label_t *clearance);
```

Description These functions convert hexadecimal string representations of internal label values into binary labels.

The `htobsl()` function converts into a binary sensitivity label, a hexadecimal string of the form:

`0x`*sensitivity_label_hexadecimal_value*

The `htobclear()` function converts into a binary clearance, a hexadecimal string of the form:

`0x`*clearance_hexadecimal_value*

Return Values These functions return non-zero if the conversion was successful, otherwise zero is returned.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete
MT-Level	MT-Safe

These functions are obsolete and retained for ease of porting. They might be removed in a future Solaris Trusted Extensions release. Use the `str_to_label(3TSOL)` function instead.

See Also `libtsol(3LIB)`, `str_to_label(3TSOL)`, `attributes(5)`, `labels(5)`

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name kva_match – look up a key in a key-value array

Synopsis

```
cc [ flag... ] file...– lsecdb [ library... ]
#include <secdb.h>

char *kva_match(kva_t *kva, char *key);
```

Description The `kva_match()` function searches a `kva_t` structure, which is part of the `authattr_t`, `execattr_t`, `profattr_t`, or `userattr_t` structures. The function takes two arguments: a pointer to a key value array, and a key. If the key is in the array, the function returns a pointer to the first corresponding value that matches that key. Otherwise, the function returns `NULL`.

Return Values Upon successful completion, the function returns a pointer to the value sought. Otherwise, it returns `NULL`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `getauthattr(3SECDB)`, `getexecattr(3SECDB)`, `getprofattr(3SECDB)`, `getuserattr(3SECDB)`

Notes The `kva_match()` function returns a pointer to data that already exists in the key-value array. It does not allocate its own memory for this pointer but obtains it from the key-value array that is passed as its first argument.

Name labelclipping, Xbsltos, Xbcleartos – translate a binary label and clip to the specified width

Synopsis

```
cc [flag...] file... -ltsol -lDtTsol [library...]

#include <Dt/label_clipping.h>

XmString Xbsltos(Display *display, const m_label_t *senslabel,
     Dimension width, const XmFontList fontlist, const int flags);

XmString Xbcleartos(Display *display, const m_label_t *clearance,
     Dimension width, const XmFontList fontlist, const int flags);
```

Description The calling process must have PRIV_SYS_TRANS_LABEL in its set of effective privileges to translate labels or clearances that dominate the current process' sensitivity label.

display The structure controlling the connection to an X Window System display.

senslabel The sensitivity label to be translated.

clearance The clearance to be translated.

width The width of the translated label or clearance in pixels. If the specified width is shorter than the full label, the label is clipped and the presence of clipped letters is indicated by an arrow. In this example, letters have been clipped to the right of: TS<-. See the sbltos(3TSOL) manual page for more information on the clipped indicator. If the specified width is equal to the display width (*display*), the label is not truncated, but word-wrapped using a width of half the display width.

fontlist A list of fonts and character sets where each font is associated with a character set.

flags The value of flags indicates which words in the label_encodings(4) file are used for the translation. See the bltos(3TSOL) manual page for a description of the flag values: LONG_WORDS, SHORT_WORDS, LONG_CLASSIFICATION, SHORT_CLASSIFICATION, ALL_ENTRIES, ACCESS_RELATED, VIEW_EXTERNAL, VIEW_INTERNAL, NO_CLASSIFICATION. BRACKETED is an additional flag that can be used with Xbsltos() only. It encloses the sensitivity label in square brackets as follows: [C].

Return Values These functions return a compound string that represents the character-coded form of the sensitivity label or clearance that is translated. The compound string uses the language and fonts specified in *fontlist* and is clipped to *width*. These functions return NULL if the label or clearance is not a valid, required type as defined in the label_encodings(4) file, or not dominated by the process' sensitivity label and the PRIV_SYS_TRANS_LABEL privilege is not asserted.

Files /usr/dt/include/Dt/label_clipping.h
Header file for label clipping functions

`/etc/security/tsol/label_encodings`
The label encodings file contains the classification names, words, constraints, and values for the defined labels of this system.

Examples

EXAMPLE 1 Translate and Clip a Clearance.

This example translates a clearance to text using the long words specified in the `label_encodings(4)` file, a font list, and clips the translated clearance to a width of 72 pixels.

```
xmstr = Xbcleartos(XtDisplay(topLevel),
&clearance, 72, fontlist, LONG_WORDS
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete
MT-Level	MT-Safe

The labelclipping functions, `Xbsltos()` and `Xbcleartos()`, are obsolete. Use the `label_to_str(3TSOL)` function instead.

See Also

`bltos(3TSOL)`, `label_to_str(3TSOL)`, `libtsol(3LIB)`, `label_encodings(4)`, `attributes(5)`

See `XmStringDraw`(3) and `FontList`(3) for information on the creation and structure of a font list.

Notes

The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name label_to_str – convert labels to human readable strings

Synopsis

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

#include <tsol/label.h>

int label_to_str(const m_label_t *label, char **string,
    const m_label_str_t conversion_type, uint_t flags);
```

Description `label_to_str()` is a simple function to convert various mandatory label types to human readable strings.

label is the mandatory label to convert. *string* points to memory that is allocated by `label_to_str()` that contains the converted string. The caller is responsible for calling `free(3C)` to free allocated memory.

The calling process must have mandatory read access to the resulting human readable string. Or the calling process must have the `sys_trans_label` privilege.

The *conversion_type* parameter controls the type of label conversion. Not all types of conversion are valid for all types of label:

`M_LABEL`	Converts *label* to a human readable string based on its type.
`M_INTERNAL`	Converts *label* to an internal text representation that is safe for storing in a public object. Internal conversions can later be parsed to their same value.
`M_COLOR`	Converts *label* to a string that represents the color name that the administrator has associated with the label.
`PRINTER_TOP_BOTTOM`	Converts *label* to a human readable string that is appropriate for use as the top and bottom label of banner and trailer pages in the Defense Intelligence Agency (DIA) encodings printed output schema.
`PRINTER_LABEL`	Converts *label* to a human readable string that is appropriate for use as the banner page downgrade warning in the DIA encodings printed output schema.
`PRINTER_CAVEATS`	Converts *label* to a human readable string that is appropriate for use as the banner page caveats section in the DIA encodings printed output schema.
`PRINTER_CHANNELS`	Converts *label* to a human readable string that is appropriate for use as the banner page handling channels in the DIA encodings printed output schema.

The *flags* parameter provides a hint to the label conversion:

`DEF_NAMES`	The default names are preferred.

`SHORT_NAMES` Short names are preferred where defined.

`LONG_NAMES` Long names are preferred.

Return Values Upon successful completion, the `label_to_str()` function returns 0. Otherwise, -1 is returned, `errno` is set to indicate the error and the string pointer is set to `NULL`.

Errors The `label_to_str()` function will fail if:

`EINVAL` Invalid parameter.

`ENOTSUP` The system does not support label translations.

`ENOMEM` The physical limits of the system are exceeded by size bytes of memory which cannot be allocated.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	See below.
MT-Level	MT-Safe
Standard	See below.

The `label_to_str()` function is Committed. The returned string is Not-an-Interface and is dependent on the specific `label_encodings` file. The conversion type `INTERNAL` is Uncommitted, but is always accepted as input to `str_to_label(3TSOL)`.

Conversion types that are relative to the DIA encodings schema are Standard. Standard is specified in `label_encodings(4)`.

See Also `free(3C)`, `libtsol(3LIB)`, `str_to_label(3TSOL)`, `label_encodings(4)`, `attributes(5)`, `labels(5)`

"Using the label_to_str Function" in *Solaris Trusted Extensions Developer's Guide*

Warnings A number of these conversions rely on the DIA label encodings schema. They might not be valid for other label schemata.

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name libtecla_version – query libtecla version number

Synopsis

```
cc [ flag... ] file... -ltecla [ library... ]
#include <libtecla.h>

void libtecla_version(int *major, int *minor, int *micro);
```

Description The `libtecla_version()` function queries for the version number of the library.

On return, this function records the three components of the libtecla version number in *major, *minor, *micro. The formal meaning of the three components is as follows:

major — Incrementing this number implies that a change has been made to the library's public interface that makes it binary incompatible with programs that were linked with previous shared versions of libtecla.

minor — This number is incremented by one whenever additional functionality, such as new functions or modules, are added to the library.

micro — This number is incremented whenever modifications to the library are made that make no changes to the public interface, but which fix bugs and/or improve the behind-the-scenes implementation.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also `libtecla(3LIB)`, `attributes(5)`

Name libtnfctl – library for TNF probe control in a process or the kernel

Synopsis

```
cc [ flag ... ] file ... -ltnfctl [ library ... ]
#include <tnf/tnfctl.h>
```

Description The `libtnfctl` library provides an API to control TNF ("Trace Normal Form") probes within a process or the kernel. See `tracing(3TNF)` for an overview of the Solaris tracing architecture. The client of `libtnfctl` controls probes in one of four modes:

internal mode — The target is the controlling process itself; that is, the client controls its own probes.

direct mode — The target is a separate process; a client can either `exec(2)` a program or attach to a running process for probe control. The `libtnfctl` library uses `proc(4)` on the target process for probe and process control in this mode, and additionally provides basic process control features.

indirect mode — The target is a separate process, but the controlling process is already using `proc(4)` to control the target, and hence `libtnfctl` cannot use those interfaces directly. Use this mode to control probes from within a debugger. In this mode, the client must provide a set of functions that `libtnfctl` can use to query and update the target process.

kernel mode — The target is the Solaris kernel.

A process is controlled "externally" if it is being controlled in either direct mode or indirect mode. Alternatively, a process is controlled "internally" when it uses internal mode to control its own probes.

There can be only one client at a time doing probe control on a given process. Therefore, it is not possible for a process to be controlled internally while it is being controlled externally. It is also not possible to have a process controlled by multiple external processes. Similarly, there can be only one process at a time doing kernel probe control. Note, however, that while a given target may only be controlled by one `libtnfctl` client, a single client may control an arbitrary number of targets. That is, it is possible for a process to simultaneously control its own probes, probes in other processes, and probes in the kernel.

The following tables denotes the modes applicable to all `libtnfctl` interfaces (INT = internal mode; D = direct mode; IND = indirect mode; K = kernel mode).

These interfaces create handles in the specified modes:

`tnfctl_internal_open()`	INT			
`tnfctl_exec_open()`		D		
`tnfctl_pid_open()`		D		

`tnfctl_indirect_open()`			IND	
`tnfctl_kernel_open()`				K

These interfaces are used with the specified modes:

`tnfctl_continue()`		D		
`tnfctl_probe_connect()`	INT	D	IND	
`tnfctl_probe_disconnect_all ()`	INT	D	IND	
`tnfctl_trace_attrs_get()`	INT	D	IND	K
`tnfctl_buffer_alloc()`	INT	D	IND	K
`tnfctl_register_funcs()`	INT	D	IND	K
`tnfctl_probe_apply()`	INT	D	IND	K
`tnfctl_probe_apply_ids()`	INT	D	IND	K
`tnfctl_probe_state_get ()`	INT	D	IND	K
`tnfctl_probe_enable()`	INT	D	IND	K
`tnfctl_probe_disable()`	INT	D	IND	K
`tnfctl_probe_trace()`	INT	D	IND	K
`tnfctl_probe_untrace()`	INT	D	IND	K
`tnfctl_check_libs()`	INT	D	IND	K
`tnfctl_close()`	INT	D	IND	K
`tnfctl_strerror()`	INT	D	IND	K
`tnfctl_buffer_dealloc()`				K
`tnfctl_trace_state_set()`				K
`tnfctl_filter_state_set()`				K
`tnfctl_filter_list_get()`				K
`tnfctl_filter_list_add()`				K
`tnfctl_filter_list_delete()`				K

When using `libtnfctl`, the first task is to create a handle for controlling probes. The `tnfctl_internal_open()` function creates an internal mode handle for controlling probes in the same process, as described above. The `tnfctl_pid_open()` and `tnfctl_exec_open()` functions create handles in direct mode. The `tnfctl_indirect_open()` function creates an

indirect mode handle, and the `tnfctl_kernel_open()` function creates a kernel mode handle. A handle is required for use in nearly all other `libtnfctl` functions. The `tnfctl_close()` function releases the resources associated with a handle.

The `tnfctl_continue()` function is used in direct mode to resume execution of the target process.

The `tnfctl_buffer_alloc()` function allocates a trace file or, in kernel mode, a trace buffer.

The `tnfctl_probe_apply()` and `tnfctl_probe_apply_ids()` functions call a specified function for each probe or for a designated set of probes.

The `tnfctl_register_funcs()` function registers functions to be called whenever new probes are seen or probes have disappeared, providing an opportunity to do one-time processing for each probe.

The `tnfctl_check_libs()` function is used primarily in indirect mode to check whether any new probes have appeared, that is, they have been made available by `dlopen(3C)`, or have disappeared, that is, they have disassociated from the process by `dlclose(3C)`.

The `tnfctl_probe_enable()` and `tnfctl_probe_disable()` functions control whether the probe, when hit, will be ignored.

The `tnfctl_probe_trace()` and `tnfctl_probe_untrace()` functions control whether an enabled probe, when hit, will cause an entry to be made in the trace file.

The `tnfctl_probe_connect()` and `tnfctl_probe_disconnect_all()` functions control which functions, if any, are called when an enabled probe is hit.

The `tnfctl_probe_state_get()` function returns information about the status of a probe, such as whether it is currently enabled.

The `tnfctl_trace_attrs_get()` function returns information about the tracing session, such as the size of the trace buffer or trace file.

The `tnfctl_strerror()` function maps a `tnfctl` error code to a string, for reporting purposes.

The remaining functions apply only to kernel mode.

The `tnfctl_trace_state_set()` function controls the master switch for kernel tracing. See `prex(1)` for more details.

The `tnfctl_filter_state_set()`, `tnfctl_filter_list_get()`, `tnfctl_filter_list_add()`, and `tnfctl_filter_list_delete()` functions allow a set of processes to be specified for which probes will not be ignored when hit. This prevents kernel activity caused by uninteresting processes from cluttering up the kernel's trace buffer.

The `tnfctl_buffer_dealloc()` function deallocates the kernel's internal trace buffer.

Return Values Upon successful completion, these functions return `TNFCTL_ERR_NONE`.

Errors The error codes for `libtnfctl` are:

`TNFCTL_ERR_ACCES`	Permission denied.
`TNFCTL_ERR_NOTARGET`	The target process completed.
`TNFCTL_ERR_ALLOCFAIL`	A memory allocation failure occurred.
`TNFCTL_ERR_INTERNAL`	An internal error occurred.
`TNFCTL_ERR_SIZETOOSMALL`	The requested trace size is too small.
`TNFCTL_ERR_SIZETOOBIG`	The requested trace size is too big.
`TNFCTL_ERR_BADARG`	Bad input argument.
`TNFCTL_ERR_NOTDYNAMIC`	The target is not a dynamic executable.
`TNFCTL_ERR_NOLIBTNFPROBE`	`libtnfprobe.so` not linked in target.
`TNFCTL_ERR_BUFBROKEN`	Tracing is broken in the target.
`TNFCTL_ERR_BUFEXISTS`	A buffer already exists.
`TNFCTL_ERR_NOBUF`	No buffer exists.
`TNFCTL_ERR_BADDEALLOC`	Cannot deallocate buffer.
`TNFCTL_ERR_NOPROCESS`	No such target process exists.
`TNFCTL_ERR_FILENOTFOUND`	File not found.
`TNFCTL_ERR_BUSY`	Cannot attach to process or kernel because it is already tracing.
`TNFCTL_ERR_INVALIDPROBE`	Probe no longer valid.
`TNFCTL_ERR_USR1`	Error code reserved for user.
`TNFCTL_ERR_USR2`	Error code reserved for user.
`TNFCTL_ERR_USR3`	Error code reserved for user.
`TNFCTL_ERR_USR4`	Error code reserved for user.
`TNFCTL_ERR_USR5`	Error code reserved for user.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWtnfc

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	MT-Safe with exceptions

See Also prex(1), exec(2), dlclose(3C), dlopen(3C), TNF_PROBE(3TNF), tnfctl_buffer_alloc(3TNF), tnfctl_buffer_dealloc(3TNF), tnfctl_check_libs(3TNF), tnfctl_close(3TNF), tnfctl_continue(3TNF), tnfctl_internal_open(3TNF), tnfctl_exec_open(3TNF), tnfctl_filter_list_add(3TNF), tnfctl_filter_list_delete(3TNF), tnfctl_filter_list_get(3TNF), tnfctl_filter_state_set(3TNF), tnfctl_kernel_open(3TNF), tnfctl_pid_open(3TNF), tnfctl_probe_apply(3TNF), tnfctl_probe_apply_ids(3TNF), tnfctl_probe_connect(3TNF), tnfctl_probe_disable(3TNF), tnfctl_probe_enable(3TNF), tnfctl_probe_state_get(3TNF), tnfctl_probe_trace(3TNF), tnfctl_probe_untrace(3TNF), tnfctl_indirect_open(3TNF), tnfctl_register_funcs(3TNF), tnfctl_strerror(3TNF), tnfctl_trace_attrs_get(3TNF), tnfctl_trace_state_set(3TNF), libtnfctl(3LIB), proc(4), attributes(5)

Linker and Libraries Guide

Notes This API is MT-Safe. Multiple threads may concurrently operate on independent `tnfctl` handles, which is the typical behavior expected. The `libtnfctl` library does not support multiple threads operating on the same `tnfctl` handle. If this is desired, it is the client's responsibility to implement locking to ensure that two threads that use the same `tnfctl` handle are not simultaneously in a `libtnfctl` interface.

Name media_findname – convert a supplied name into an absolute pathname that can be used to access removable media

Synopsis

```
cc [ flag ... ] file ... -lvolmgt [ library ... ]
#include <volmgt.h>

char *media_findname(char *start);
```

Description This function is obsolete. The management of removable media by the Volume Management feature, including `vold`, has been replaced by software that supports the Hardware Abstraction Layer (HAL). Programmatic support for HAL is through the HAL APIs, which are documented on the HAL web site. See `hal(5)`. The return value of this function is undefined.

`media_findname()` converts the supplied *start* string into an absolute pathname that can then be used to access a particular piece of media.

The *start* parameter can be one of the following types of specifications:

`/dev/...`	An absolute pathname in `/dev`, such as `/dev/rdiskette0`, in which case a copy of that string is returned (see `NOTES` on this page).
volume_name	The volume name for a particular volume, such as `fred` (see `fdformat(1)` for a description of how to label floppies).
volmgt_symname	The symbolic name for a device, such as `floppy0` or `cdrom2`.
media_type	The generic media type name. For example, `floppy` or `cdrom`. In this case `media_findname()` looks for the first piece of media that matches that media type, starting at 0 (zero) and continuing on until a match is found (or some fairly large maximum number is reached). In this case, if a match is found, a copy of the pathname to the volume found is returned.

Return Values The return from this function is undefined.

Errors For cases where the supplied *start* parameter is an absolute pathname, `media_findname()` can fail, returning a null string pointer, if an `lstat(2)` of that supplied pathname fails. Also, if the supplied absolute pathname is a symbolic link, `media_findname()` can fail if a `readlink(2)` of that symbolic link fails, or if a `stat(2)` of the pathname pointed to by that symbolic link fails, or if any of the following is true:

`ENXIO` The specified absolute pathname was not a character special device, and it was not a directory with a character special device in it.

Examples

EXAMPLE 1 Sample programs of the `media_findname()` function.

The following example attempts to find what the pathname is to a piece of media called fred. Notice that a `volmgt_check()` is done first (see the `NOTES` section on this page).

```
(void) volmgt_check(NULL);
if ((nm = media_findname("fred")) != NULL) {
        (void) printf("media named \"fred\" is at \"%s\"\n", nm);
} else {
      (void) printf("media named \"fred\" not found\n");
}
```

This example looks for whatever volume is in the first floppy drive, letting `media_findname()` call `volmgt_check()` if and only if no floppy is currently known to be the first floppy drive.

```
if ((nm = media_findname("floppy0")) != NULL) {
        (void) printf("path to floppy0 is \"%s\"\n", nm);
} else {
        (void) printf("nothing in floppy0\n");
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Unsafe
Interface Stability	Obsolete

See Also

`fdformat(1)`, `lstat(2)`, `readlink(2)`, `stat(2)`, `free(3C)`, `malloc(3C)`, `volmgt_check(3VOLMGT)`, `volmgt_inuse(3VOLMGT)`, `volmgt_root(3VOLMGT)`, `volmgt_running(3VOLMGT)`, `volmgt_symname(3VOLMGT)`, `attributes(5)`, `hal(5)`

Notes

If `media_findname()` cannot find a match for the supplied name, it performs a `volmgt_check(3VOLMGT)` and tries again, so it can be more efficient to perform `volmgt_check()` before calling `media_findname()`.

Upon success `media_findname()` returns a pointer to string which has been allocated; this should be freed when no longer in use (see `free(3C)`).

Name media_getattr, media_setattr – get and set media attributes

Synopsis

```
cc [ flag ... ] file ... -lvolmgt [ library ... ]
#include <volmgt.h>

char *media_getattr(char *vol_path, char *attr);

int media_setattr(char *vol_path, char *attr, char *value);
```

Description This function is obsolete. The management of removable media by the Volume Management feature, including `vold`, has been replaced by software that supports the Hardware Abstraction Layer (HAL). Programmatic support for HAL is through the HAL APIs, which are documented on the HAL web site. See `hal(5)`. The return value of this function is undefined.

`media_setattr()` and `media_getattr()` respectively set and get attribute-value pairs (called properties) on a per-volume basis.

Volume management supports system properties and user properties. System properties are ones that volume management predefines. Some of these system properties are writable, but only by the user that owns the volume being specified, and some system properties are read only:

Attribute	Writable	Value	Description
`s-access`	RO	"seq", "rand"	sequential or random access
`s-density`	RO	"low", "medium", "high"	media density
`s-parts`	RO	comma separated list of slice numbers	list of partitions on this volume
`s-location`	RO	*pathname*	volume management pathname to media
`s-mejectable`	RO	"true", "false"	whether or not media is manually ejectable
`s-rmoneject`	R/W	"true", "false"	should media access points be removed from database upon ejection
`s-enxio`	R/W	"true", "false"	if set return `ENXIO` when media access attempted

Properties can also be defined by the user. In this case the value can be any string the user wishes.

Return Values The return from this function is undefined.

Errors

Both `media_getattr()` and `media_setattr()` can fail returning a null pointer if an `open(2)` of the specified *vol_path* fails, if an `fstat(2)` of that pathname fails, or if that pathname is not a block or character special device.

`media_getattr()` can also fail if the specified attribute was not found, and `media_setattr()` can also fail if the caller doesn't have permission to set the attribute, either because it's is a system attribute, or because the caller doesn't own the specified volume.

Examples

EXAMPLE 1 Using `media_getattr()`

The following example checks to see if the volume called *fred* that volume management is managing can be ejected by means of software, or if it can only be manually ejected:

```
if (media_getattr("/rdsk/fred", "s-mejectable") != NULL) {
        (void) printf("\"fred\" must be manually ejected\n");
} else {
        (void) printf("software can eject \"fred\"\n");
}
```

This example shows setting the *s-enxio* property for the floppy volume currently in the first floppy drive:

```
int     res;
if ((res = media_setattr("/dev/aliases/floppy0", "s-enxio",
    "true")) == 0) {
        (void) printf("can't set s-enxio flag for floppy0\n");
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe
Interface Stability	Obsolete

See Also

`lstat(2)`, `open(2)`, `readlink(2)`, `stat(2)`, `free(3C)`, `malloc(3C)`, `media_findname(3VOLMGT)`, `volmgt_check(3VOLMGT)`, `volmgt_inuse(3VOLMGT)`, `volmgt_root(3VOLMGT)`, `volmgt_running(3VOLMGT)`, `volmgt_symname(3VOLMGT)`, `attributes(5)`, `hal(5)`

Name media_getid – return the id of a piece of media

Synopsis

```
cc [flag ...] file ...–lvolgmt [library ...]

#include <volmgt.h>



ulonglong_t media_getid(char *vol_path);
```

Description This function is obsolete. The management of removable media by the Volume Management feature, including `vold`, has been replaced by software that supports the Hardware Abstraction Layer (HAL). Programmatic support for HAL is through the HAL APIs, which are documented on the HAL web site. See `hal(5)`. The return value of this function is undefined.

`media_getid()` returns the *id* of a piece of media. Volume management must be running. See `volmgt_running(3VOLMGT)`.

Parameters *vol_path* Path to the block or character special device.

Return Values The return from this function is undefined.

Examples **EXAMPLE 1** Using `media_getid()`

The following example first checks if volume management is running, then checks the volume management name space for *path*, and then returns the *id* for the piece of media.

```
char *path;

    ...

if (volmgt_running()) {
     if (volmgt_ownspath(path)) {
          (void) printf("id of %s is %lld\n",
               path, media_getid(path));
     }
    }
```

If a program using `media_getid()` does not check whether or not volume management is running, then any `NULL` return value will be ambiguous, as it could mean that either volume management does not have *path* in its name space, or volume management is not running.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

See Also volmgt_ownspath(3VOLMGT),volmgt_running(3VOLMGT),attributes(5),hal(5)

Name m_label, m_label_alloc, m_label_dup, m_label_free – m_label functions

Synopsis

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

#include <tsol/label.h>

m_label_t *m_label_alloc(const m_label_type_t label_type);

int m_label_dup(m_label_t **dst, const m_label_t *src);

void m_label_free(m_label_t *label);
```

Description The `m_label_alloc()` function allocates resources for a new label. The *label_type* argument defines the type for a newly allocated label. The label type can be:

`MAC_LABEL` A Mandatory Access Control (MAC) label.

`USER_CLEAR` A user clearance.

The `m_label_dup()` function allocates resources for a new *dst* label. The function returns a pointer to the allocated label, which is an exact copy of the *src* label. The caller is responsible for freeing the allocated resources by calling `m_label_free()`.

The `m_label_free()` function frees resources that are associated with the previously allocated label.

Return Values Upon successful completion, the `m_label_alloc()` function returns a pointer to the newly allocated label. Otherwise, `m_label_alloc()` returns `NULL` and `errno` is set to indicate the error.

Upon successful completion, the `m_label_dup()` function returns 0. Otherwise, `-1` is returned and `errno` is set to indicate the error.

Errors The `m_label_alloc()` function will fail if:

`EINVAL` Invalid parameter.

`ENOMEM` The physical limits of the system are exceeded by size bytes of memory which cannot be allocated.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `label_to_str(3TSOL)`, `libtsol(3LIB)`, `str_to_label(3TSOL)`, `label_encodings(4)`, `attributes(5)`, `labels(5)`

"Determining Whether the Printing Service Is Running in a Labeled Environment" in *Solaris Trusted Extensions Developer's Guide*

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name pca_lookup_file, del_PathCache, del_PcaPathConf, new_PathCache, new_PcaPathConf, pca_last_error, pca_path_completions, pca_scan_path, pca_set_check_fn, ppc_file_start, ppc_literal_escapes – lookup a file in a list of directories

Synopsis

```
cc [ flag... ] file... -ltecla [ library... ]
#include <libtecla.h>

char *pca_lookup_file(PathCache *pc, const char *name,
     int name_len, int literal);

PathCache *del_PathCache(PathCache *pc);

PcaPathConf *del_PcaPathConf(PcaPathConf *ppc);

PathCache *new_PathCache(void);

PcaPathConf *new_PcaPathConf(PathCache *pc);

const char *pca_last_error(PathCache *pc);

CPL_MATCH_FN(pca_path_completions);

int pca_scan_path(PathCache *pc, const char *path);

void pca_set_check_fn(PathCache *pc, CplCheckFn *check_fn,
     void *data);

void ppc_file_start(PcaPathConf *ppc, int start_index);

void ppc_literal_escapes(PcaPathConf *ppc, int literal);
```

Description The `PathCache` object is part of the `libtecla(3LIB)` library. `PathCache` objects allow an application to search for files in any colon separated list of directories, such as the UNIX execution `PATH` environment variable. Files in absolute directories are cached in a `PathCache` object, whereas relative directories are scanned as needed. Using a `PathCache` object, you can look up the full pathname of a simple filename, or you can obtain a list of the possible completions of a given filename prefix. By default all files in the list of directories are targets for lookup and completion, but a versatile mechanism is provided for only selecting specific types of files. The obvious application of this facility is to provide Tab-completion and lookup of executable commands in the UNIX `PATH`, so an optional callback which rejects all but executable files, is provided.

An Example Under UNIX, the following example program looks up and displays the full pathnames of each of the command names on the command line.

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

int main(int argc, char *argv[])
{
        int i;
        /*
```

```
        * Create a cache for executable files.
        */
        PathCache *pc = new_PathCache();
        if(!pc)
          exit(1);
        /*
        * Scan the user's PATH for executables.
        */
        if(pca_scan_path(pc, getenv("PATH"))) {
          fprintf(stderr, "%s\n", pca_last_error(pc));
          exit(1);
        }
        /*
        * Arrange to only report executable files.
        */
        pca_set_check_fn(pc, cpl_check_exe, NULL);
        /*
        * Lookup and display the full pathname of each of the
        * commands listed on the command line.
        */
        for(i=1; i<argc; i++) {
          char *cmd = pca_lookup_file(pc, argv[i], -1, 0);
          printf("The full pathname of '%s' is %s\\n", argv[i],
                 cmd ? cmd : "unknown");
        }
        pc = del_PathCache(pc);  /* Clean up */
        return 0;
}
```

The following is an example of what this does on a laptop under LINUX:

```
$ ./example less more blob
The full pathname of 'less' is /usr/bin/less
The full pathname of 'more' is /bin/more
The full pathname of 'blob' is unknown
$
```

Function Descriptions

To use the facilities of this module, you must first allocate a `PathCache` object by calling the `new_PathCache()` constructor function. This function creates the resources needed to cache and lookup files in a list of directories. It returns `NULL` on error.

Populating The Cache

Once you have created a cache, it needs to be populated with files. To do this, call the `pca_scan_path()` function. Whenever this function is called, it discards the current contents of the cache, then scans the list of directories specified in its path argument for files. The path argument must be a string containing a colon-separated list of directories, such as "`/usr/bin:/home/mcs/bin:`". This can include directories specified by absolute pathnames such as "`/usr/bin`", as well as sub-directories specified by relative pathnames such as "`.`" or "`bin`". Files in the absolute directories are immediately cached in the specified `PathCache`

object, whereas subdirectories, whose identities obviously change whenever the current working directory is changed, are marked to be scanned on the fly whenever a file is looked up.

On success this function return 0. On error it returns 1, and a description of the error can be obtained by calling `pca_last_error`(*pc*).

Looking Up Files

Once the cache has been populated with files, you can look up the full pathname of a file, simply by specifying its filename to `pca_lookup_file()`.

To make it possible to pass this function a filename which is actually part of a longer string, the *name_len* argument can be used to specify the length of the filename at the start of the *name*[] argument. If you pass -1 for this length, the length of the string will be determined with *strlen*. If the *name*[] string might contain backslashes that escape the special meanings of spaces and tabs within the filename, give the *literal* argument the value 0. Otherwise, if backslashes should be treated as normal characters, pass 1 for the value of the *literal* argument.

Filename Completion

Looking up the potential completions of a filename-prefix in the filename cache is achieved by passing the provided `pca_path_completions()` callback function to the `cpl_complete_word`(3TECLA) function.

This callback requires that its data argument be a pointer to a PcaPathConf object. Configuration objects of this type are allocated by calling `new_PcaPathConf()`.

This function returns an object initialized with default configuration parameters, which determine how the `cpl_path_completions()` callback function behaves. The functions which allow you to individually change these parameters are discussed below.

By default, the `pca_path_completions()` callback function searches backwards for the start of the filename being completed, looking for the first un-escaped space or the start of the input line. If you wish to specify a different location, call `ppc_file_start()` with the index at which the filename starts in the input line. Passing *start_index*=-1 re-enables the default behavior.

By default, when `pca_path_completions()` looks at a filename in the input line, each lone backslash in the input line is interpreted as being a special character which removes any special significance of the character which follows it, such as a space which should be taken as part of the filename rather than delimiting the start of the filename. These backslashes are thus ignored while looking for completions, and subsequently added before spaces, tabs and literal backslashes in the list of completions. To have unescaped backslashes treated as normal characters, call `ppc_literal_escapes()` with a non-zero value in its literal argument.

When you have finished with a `PcaPathConf` variable, you can pass it to the `del_PcaPathConf()` destructor function to reclaim its memory.

Being Selective

If you are only interested in certain types or files, such as, for example, executable files, or files whose names end in a particular suffix, you can arrange for the file completion and lookup functions to be selective in the filenames that they return. This is done by registering a callback function with your `PathCache` object. Thereafter, whenever a filename is found which either

matches a filename being looked up or matches a prefix which is being completed, your callback function will be called with the full pathname of the file, plus any application-specific data that you provide. If the callback returns 1 the filename will be reported as a match. If it returns 0, it will be ignored. Suitable callback functions and their prototypes should be declared with the following macro. The `CplCheckFn` typedef is also provided in case you wish to declare pointers to such functions

```
#define CPL_CHECK_FN(fn) int (fn)(void *data, const char *pathname)
typedef CPL_CHECK_FN(CplCheckFn);
```

Registering one of these functions involves calling the `pca_set_check_fn()` function. In addition to the callback function passed with the *check_fn* argument, you can pass a pointer to anything with the *data* argument. This pointer will be passed on to your callback function by its own *data* argument whenever it is called, providing a way to pass application-specific data to your callback. Note that these callbacks are passed the full pathname of each matching file, so the decision about whether a file is of interest can be based on any property of the file, not just its filename. As an example, the provided `cpl_check_exe()` callback function looks at the executable permissions of the file and the permissions of its parent directories, and only returns 1 if the user has execute permission to the file. This callback function can thus be used to lookup or complete command names found in the directories listed in the user's `PATH` environment variable. The example program above provides a demonstration of this.

Beware that if somebody tries to complete an empty string, your callback will get called once for every file in the cache, which could number in the thousands. If your callback does anything time consuming, this could result in an unacceptable delay for the user, so callbacks should be kept short.

To improve performance, whenever one of these callbacks is called, the choice that it makes is cached, and the next time the corresponding file is looked up, instead of calling the callback again, the cached record of whether it was accepted or rejected is used. Thus if somebody tries to complete an empty string, and hits tab a second time when nothing appears to happen, there will only be one long delay, since the second pass will operate entirely from the cached dispositions of the files. These cached dipositions are discarded whenever `pca_scan_path()` is called, and whenever `pca_set_check_fn()` is called with changed callback function or *data* arguments.

Error Handling

If `pca_scan_path()` reports that an error occurred by returning 1, you can obtain a terse description of the error by calling `pca_last_error`(*pc*). This returns an internal string containing an error message.

Cleaning Up

Once you have finished using a `PathCache` object, you can reclaim its resources by passing it to the `del_PathCache()` destructor function. This takes a pointer to one of these objects, and always returns `NULL`.

Thread Safety

It is safe to use the facilities of this module in multiple threads, provided that each thread uses a separately allocated `PathCache` object. In other words, if two threads want to do path searching, they should each call `new_PathCache()` to allocate their own caches.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also

`cpl_complete_word(3TECLA)`, `ef_expand_file(3TECLA)`, `gl_get_line(3TECLA)`, `libtecla(3LIB)`, `attributes(5)`

Name sbltos, sbsltos, sbcleartos – translate binary labels to canonical character-coded labels

Synopsis

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

#include <tsol/label.h>

char *sbsltos(const m_label_t *label, const int len);

char *sbcleartos(const m_label_t *clearance, const int len);
```

Description These functions translate binary labels into canonical strings that are clipped to the number of printable characters specified in *len*. Clipping is required if the number of characters of the translated string is greater than *len*. Clipping is done by truncating the label on the right to two characters less than the specified number of characters. A clipped indicator, "<–", is appended to sensitivity labels and clearances. The character-coded label begins with a classification name separated with a single space character from the list of words making up the remainder of the label. The binary labels must be of the proper defined type and dominated by the process's sensitivity label. A *len* of 0 (zero) returns the entire string with no clipping.

The `sbsltos()` function translates a binary sensitivity label into a clipped string using the long form of the words and the short form of the classification name. If *len* is less than the minimum number of characters (three), the translation fails.

The `sbcleartos()` function translates a binary clearance into a clipped string using the long form of the words and the short form of the classification name. If *len* is less than the minimum number of characters (three), the translation fails. The translation of a clearance might not be the same as the translation of a sensitivity label. These functions use different tables of the `label_encodings` file which might contain different words and constraints.

The calling process must have `PRIV_SYS_TRANS_LABEL` in its set of effective privileges to perform label translation on labels that dominate the current process's sensitivity label.

Process Attributes If the `VIEW_EXTERNAL` or `VIEW_INTERNAL` flags are not specified, translation of `ADMIN_LOW` and `ADMIN_HIGH` labels is controlled by the label view process attribute flags. If no label view process attribute flags are defined, their translation is controlled by the label view configured in the `label_encodings` file. A value of External specifies that `ADMIN_LOW` and `ADMIN_HIGH` labels are mapped to the lowest and highest labels defined in the `label_encodings` file. A value of Internal specifies that the `ADMIN_LOW` and `ADMIN_HIGH` labels are translated to the `admin low name` and `admin high name` strings specified in the `label_encodings` file. If no such names are specified, the strings "`ADMIN_LOW`" and "`ADMIN_HIGH`" are used.

Return Values These functions return a pointer to a statically allocated string that contains the result of the translation, or `(char *)0` if the translation fails for any reason.

Examples

`sbsltos()` Assume that a sensitivity label is:

```
UN TOP/MIDDLE/LOWER DRAWER
```

When clipped to ten characters it is:

```
UN TOP/M<—
```

`sbcleartos()` Assume that a clearance is:

```
UN TOP/MIDDLE/LOWER DRAWER
```

When clipped to ten characters it is:

```
UN TOP/M<—
```

Files `/etc/security/tsol/label_encodings`
The label encodings file contains the classification names, words, constraints, and values for the defined labels of this system.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete
MT-Level	Unsafe

These functions are obsolete and retained for ease of porting. They might be removed in a future Solaris Trusted Extensions release. Use the `label_to_str(3TSOL)` function instead.

See Also `label_to_str(3TSOL)`, `libtsol(3LIB)`, `attributes(5)`, `labels(5)`

Warnings All these functions share the same statically allocated string storage. They are not MT-Safe. Subsequent calls to any of these functions will overwrite that string with the newly translated string.

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name scf_entry_create, scf_entry_handle, scf_entry_destroy, scf_entry_destroy_children, scf_entry_reset, scf_entry_add_value – create and manipulate transaction in the Service Configuration Facility

Synopsis

```
cc [ flag... ] file... -lscf [ library... ]
#include <libscf.h>

scf_transaction_entry_t *scf_entry_create(scf_handle_t *handle);

scf_handle_t *scf_entry_handle(scf_transaction_entry_t *entry);

void scf_entry_destroy(scf_transaction_entry_t *entry);

void scf_entry_destroy_children(scf_transaction_entry_t *entry);

void scf_entry_reset(scf_transaction_entry_t *entry);

int scf_entry_add_value(scf_transaction_entry_t *entry,
     scf_value_t *value);
```

Description The `scf_entry_create()` function allocates a new transaction entry handle. The `scf_entry_destroy()` function destroys the transaction entry handle.

The `scf_entry_handle()` function retrieves the handle associated with *entry*.

A transaction entry represents a single action on a property in a property group. If an entry is added to a transaction using `scf_transaction_property_new(3SCF)`, `scf_transaction_property_change(3SCF)`, or `scf_transaction_property_change_type(3SCF)`, `scf_entry_add_value()` can be called zero or more times to set up the set of values for that property. Each value must be set and of a compatible type to the type associated with the entry. When later retrieved from the property, the values will have the type of the entry. If the values are committed successfully with `scf_transaction_commit(3SCF)`, they will be set in the order in which they were added with `scf_entry_add_value()`.

The `scf_entry_reset()` function resets a transaction entry, disassociating it from any transaction it is a part of (invalidating the transaction in the process), and disassociating any values that were added to it.

The `scf_entry_destroy_children()` function destroys all values associated with the transaction entry. The entry itself is not destroyed.

Return Values Upon successful completion, `scf_entry_create()` returns a new `scf_transaction_entry_t`. Otherwise, it returns `NULL`.

Upon successful completion, `scf_entry_handle()` returns the handle associated with the transaction entry. Otherwise, it returns `NULL`.

Upon successful completion, `scf_entry_add_value()` returns 0. Otherwise, it returns -1.

Errors The `scf_entry_create()` function will fail if:

`SCF_ERROR_INVALID_ARGUMENT`	The *handle* argument is `NULL`.
`SCF_ERROR_NO_MEMORY`	There is not enough memory to allocate an `scf_transaction_entry_t`.

The `scf_entry_handle()` function will fail if:

`SCF_ERROR_HANDLE_DESTROYED`	The handle associated with entry has been destroyed.

The `scf_entry_add_value()` function will fail if:

`SCF_ERROR_HANDLE_MISMATCH`	The *value* and *entry* arguments are not derived from the same handle.
`SCF_ERROR_IN_USE`	The value has been added to another entry.
`SCF_ERROR_INTERNAL`	An internal error occurred.
`SCF_ERROR_INVALID_ARGUMENT`	The *value* argument is not set, or the entry was added to the transaction using `scf_transaction_property_delete(3SCF)`.
`SCF_ERROR_NOT_SET`	The transaction entry is not associated with a transaction.
`SCF_ERROR_TYPE_MISMATCH`	The type of the *value* argument does not match the type that was set using `scf_transaction_property_new()`, `scf_transaction_property_change()`, or `scf_transaction_property_change_type()`.

The `scf_error(3SCF)` function can be used to retrieve the error value.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libscf(3LIB)`, `scf_error(3SCF)`, `scf_transaction_commit(3SCF)`, `scf_transaction_property_change(3SCF)`, `scf_transaction_property_change_type(3SCF)`, `scf_transaction_property_delete(3SCF)`, `scf_transaction_property_new(3SCF)`, `scf_transaction_reset(3SCF)`, `attributes(5)`

Name scf_error, scf_strerror – error interface to Service Configuration Facility

Synopsis

```
cc [ flag... ] file... -lscf [ library... ]
#include <libscf.h>

scf_error_t scf_error(void);

const char *scf_strerror(scf_error_t error);
```

Description The `scf_error()` function returns the current `libscf(3LIB)` error value for the current thread. If the immediately previous call to a `libscf` function failed, the error value will reflect the reason for that failure.

The `scf_strerror()` function takes an error code previously returned by `scf_error()` and returns a human-readable, localized description of the error.

The error values are as follows:

`SCF_ERROR_BACKEND_ACCESS`	The storage mechanism that the repository server (`svc.configd(1M)`) chose for the operation denied access.
`SCF_ERROR_BACKEND_READONLY`	The storage mechanism that the repository server (`svc.configd`) chose for the operation is read-only. For the local filesystem storage mechanism (currently `/etc/svc/repository.db`), this usually occurs because the filesystem that contains it is mounted read-only. See `mount(1M)`
`SCF_ERROR_CONNECTION_BROKEN`	The connection to repository is broken.
`SCF_ERROR_CONSTRAINT_VIOLATED`	A required constraint was not met.
`SCF_ERROR_DELETED`	Object was deleted.
`SCF_ERROR_EXISTS`	The object already exists.
`SCF_ERROR_HANDLE_DESTROYED`	An object was bound to a destroyed handle.
`SCF_ERROR_HANDLE_MISMATCH`	Objects from different SCF handles were used.
`SCF_ERROR_IN_USE`	The object is currently in use.
`SCF_ERROR_INTERNAL`	An internal error occurred.
`SCF_ERROR_INVALID_ARGUMENT`	An argument is invalid.
`SCF_ERROR_NO_MEMORY`	No memory is available.
`SCF_ERROR_NO_RESOURCES`	The repository server is out of resources.
`SCF_ERROR_NO_SERVER`	The repository server is unavailable.
`SCF_ERROR_NONE`	No error occurred.

SCF_ERROR_NOT_BOUND	The handle is not bound.
SCF_ERROR_NOT_FOUND	Nothing of that name was found.
SCF_ERROR_NOT_SET	Cannot use unset value.
SCF_ERROR_PERMISSION_DENIED	The user lacks sufficient authority to conduct the requested operation. See smf_security(5).
SCF_ERROR_TYPE_MISMATCH	The type does not match value.
SCF_ERROR_VERSION_MISMATCH	The SCF version is incompatible.

Return Values The scf_error() function returns SCF_ERROR_NONE if there have been no calls from libscf functions from the current thread. The return value is undefined if the immediately previous call to a libscf function did not fail.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also svc.configd(1M), libscf(3LIB), attributes(5), svc.configd(1M)

Name scf_handle_create, scf_handle_destroy, scf_handle_decorate, scf_handle_bind, scf_handle_unbind, scf_myname – Service Configuration Facility handle functions

Synopsis

```
cc [ flag... ] file... -lscf [ library... ]
#include <libscf.h>

scf_handle_t *scf_handle_create(scf_version_t version);

void scf_handle_destroy(scf_handle_t *handle);

int scf_handle_decorate(scf_handle_t *handle, const char *param,
     scf_value_t *value);

int scf_handle_bind(scf_handle_t *handle);

int scf_handle_unbind(scf_handle_t *handle);

ssize_t scf_myname(scf_handle_t *handle, char *out, size_t sz);
```

Description The `scf_handle_create()` function creates a new Service Configuration Facility handle that is used as the base for all communication with the configuration repository. The version argument must be `SCF_VERSION`.

The `scf_handle_decorate()` function sets a single connection-level parameter, *param*, to the supplied value. If *value* is `SCF_DECORATE_CLEAR`, *param* is reset to its default state. Values passed to `scf_handle_decorate()` can be reset, reused, or destroyed. The values set do not take effect until `scf_handle_bind()` is called. Any invalid values will not cause errors prior to the call to `scf_handle_bind()`. The only available decorations is:

debug (count) Set the debugging flags.

The `scf_handle_bind()` function binds the handle to a running `svc.configd(1M)` daemon, using the current decorations to modify the connection. All states derived from the handle are reset immediately after a successful binding.

The `scf_handle_unbind()` function severs an existing repository connection or clears the in-client state for a broken connection.

The `scf_handle_destroy()` function destroys and frees an SCF handle. It is illegal to use the handle after calling `scf_handle_destroy()`. Actions on subordinate objects act as if the handle is unbound.

The `scf_myname()` function retrieves the FMRI for the service of which the connecting process is a part. If the full FMRI does not fit in the provided buffer, it is truncated and, if *sz* > 0, zero-terminated.

Return Values Upon successful completion, `scf_handle_create()` returns the new handle. Otherwise, it returns `NULL`.

Upon successful completion, `scf_handle_decorate()`, `scf_handle_bind()`, and `scf_handle_unbind()` return 0. Otherwise, they return -1.

The `scf_myname()` function returns the length of the full FMRI. Otherwise, it returns –1.

Errors The `scf_handle_create()` function will fail if:

`SCF_ERROR_NO_MEMORY`	There is no memory available.
`SCF_ERROR_VERSION_MISMATCH`	The version is invalid, or the application was compiled against a version of the library that is more recent than the one on the system.

The `scf_handle_decorate()` function will fail if:

`SCF_ERROR_INVALID_ARGUMENT`	The *param* argument is not a recognized parameter.
`SCF_ERROR_TYPE_MISMATCH`	The *value* argument does not match the expected type for param.
`SCF_ERROR_NOT_SET`	The *value* argument is not set.
`SCF_ERROR_IN_USE`	The handle is currently bound.
`SCF_ERROR_HANDLE_MISMATCH`	The *value* argument is not derived from *handle*.

The `scf_handle_bind()` function will fail if:

`SCF_ERROR_INVALID_ARGUMENT`	One of the decorations was invalid.
`SCF_ERROR_NO_SERVER`	The repository server is not running.
`SCF_ERROR_NO_RESOURCES`	The server does not have adequate resources for a new connection.
`SCF_ERROR_IN_USE`	The handle is already bound.

The `scf_handle_unbind()` function will fail if:

`SCF_ERROR_NOT_BOUND`	The handle is not bound.

The `scf_handle_myname()` function will fail if:

`SCF_ERROR_CONNECTION_BROKEN`	The connection to the repository was lost.
`SCF_ERROR_NOT_BOUND`	The handle is not bound.
`SCF_ERROR_NOT_SET`	This process is not marked as a SMF service.

The `scf_error(3SCF)` function can be used to retrieve the error value.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	See below.

Operations on a single handle (and the objects associated with it) are Safe. Operations on different handles are MT-Safe. Objects associated with different handles cannot be mixed, as this will lead to an SCF_ERROR_HANDLE_MISMATCH error.

See Also libscf(3LIB), scf_error(3SCF), attributes(5)

Name scf_handle_decode_fmri, scf_scope_to_fmri, scf_service_to_fmri, scf_instance_to_fmri, scf_pg_to_fmri, scf_property_to_fmri – convert between objects and FMRIs in the Service Configuration Facility

Synopsis

```
cc [ flag... ] file... -lscf [ library... ]
#include <libscf.h>

int scf_handle_decode_fmri(scf_handle_t *handle, const char *fmri,
     scf_scope_t *scope, scf_service_t *service,
     scf_instance_t *instance, scf_propertygroup_t *pg,
     scf_property_t *property, int flag);

ssize_t scf_scope_to_fmri(const scf_scope_t *object,
     char *buffer, size_t sz);

ssize_t scf_service_to_fmri(const scf_scope_t *object,
     char *buffer, size_t sz);

ssize_t scf_instance_to_fmri(const scf_instance_t *inst,
     char *buffer, size_t sz);

ssize_t scf_pg_to_fmri(const scf_propertygroup_t *pg, char *out,
     size_t sz);

ssize_t scf_property_to_fmri(const scf_scope_t *object,
     char *buffer, size_t sz);
```

Description The `scf_handle_decode_fmri()` function decodes an FMRI string into a set of repository entries. Any number of the entity handles can be `NULL`. The validation and decoding of the FMRI are determined by the *flags* argument and by those arguments that are `NULL`.

If *flags* == 0, any FMRI is accepted as long as it is well-formed and exists in the repository.

If `SCF_DECODE_FMRI_EXACT` is set in *flags*, the last part of the FMRI must match the last non-null entity handle. For example, if property is `NULL` and *pg* is non-null, the FMRI must be a property group FMRI.

If `SCF_DECODE_FMRI_TRUNCATE` is set in *flags*, there is no check for the existence of any objects specified in the FMRI that follow the last non-null entity handle. For example, if property is `NULL`, *pg* is non-null, and a property FMRI is passed in, `scf_handle_decode_fmri()` succeeds as long as the property group exists, even if the referenced property does not exist.

If `SCF_DECODE_FMRI_REQUIRE_INSTANCE` (or `SCF_FMRI_REQUIRE_NO_INSTANCE`) is set in *flags*, then the FMRI must (or must not) specify an instance.

If an error occurs, all of the entity handles that were passed to the function are reset.

The `scf_scope_to_fmri()`, `scf_service_to_fmri()`, `scf_instance_to_fmri()`, `scf_pg_to_fmri()`, and `scf_property_to_fmri()` functions convert an entity handle to an FMRI.

Return Values Upon successful completion, `scf_handle_decode_fmri()` returns 0. Otherwise, it returns -1.

Upon successful completion, `scf_scope_to_fmri()`, `scf_service_to_fmri()`, `scf_instance_to_fmri()`, `scf_pg_to_fmri()`, and `scf_property_to_fmri()` return the length of the FMRI. The buffer will be null-terminated if *sz* > 0, similar to strlcpy(3C). Otherwise, they return -1 and the contents of buffer are undefined.

Errors The `scf_handle_decode_fmri()` function will fail if:

`SCF_ERROR_BACKEND_ACCESS`	The storage mechanism that the repository server (`svc.configd(1M)`) chose for the operation denied access.
`SCF_ERROR_CONNECTION_BROKEN`	The connection to the repository was lost.
`SCF_ERROR_CONSTRAINT_VIOLATED`	The FMRI does not meet the restrictions requested in the flag argument.
`SCF_ERROR_DELETED`	The object argument refers to an object that has been deleted.
`SCF_ERROR_HANDLE_MISMATCH`	One or more of the entity handles was not derived from handle.
`SCF_ERROR_INTERNAL`	An internal error occurred.
`SCF_ERROR_INVALID_ARGUMENT`	The *fmri* argument is not a valid FMRI.
`SCF_ERROR_NO_RESOURCES`	The server does not have adequate resources to complete the request.
`SCF_ERROR_NOT_BOUND`	The handle is not currently bound.
`SCF_ERROR_NOT_FOUND`	The FMRI is well-formed but there is no object in the repository matching it.
`SCF_ERROR_NOT_SET`	Cannot use unset value.

The `scf_scope_to_fmri()`, `scf_service_to_fmri()`, `scf_instance_to_fmri()`, `scf_pg_to_fmri()`, and `scf_property_to_fmri()` functions will fail if:

`SCF_ERROR_NOT_SET`	The *object* argument is not currently set.
`SCF_ERROR_DELETED`	The object argument refers to an object that has been deleted.
`SCF_ERROR_NOT_BOUND`	The handle is not currently bound.
`SCF_ERROR_CONNECTION_BROKEN`	The connection to the repository was lost.

The `scf_error(3SCF)` function can be used to retrieve the error value.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also libscf(3LIB), scf_error(3SCF), attributes(5)

Name scf_instance_create, scf_instance_handle, scf_instance_destroy, scf_instance_get_parent, scf_instance_get_name, scf_service_get_instance, scf_service_add_instance, scf_instance_delete – create and manipulate instance handles and instances in the Service Configuration Facility

Synopsis

```
cc [ flag... ] file... -lscf [ library... ]
#include <libscf.h>

scf_instance_t *scf_instance_create(scf_handle_t *handle);

scf_handle_t *scf_instance_handle(scf_instance_t *inst);

void scf_instance_destroy(scf_instance_t *inst);

int scf_instance_get_parent(const scf_instance_t *inst,
     scf_service_t *svc);

ssize_t scf_instance_get_name(const scf_instance_t *inst,
     char *name, size_t size);

int scf_service_get_instance(const scf_service_t *svc,
     const char *name, scf_instance_t *inst);

int scf_service_add_instance(const scf_service_t *svc,
     const char *name, scf_instance_t *inst);

int scf_instance_delete(scf_instance_t *inst);
```

Description Instances form the bottom layer of the Service Configuration Facility repository tree. An instance is the child of a service and has two sets of children:

Property Groups	These hold configuration information specific to this instance. See `scf_pg_create(3SCF)`, `scf_iter_instance_pgs(3SCF)`, and `scf_iter_instance_pgs_typed(3SCF)`.
Snapshots	These are complete configuration snapshots that hold unchanging copies of all of the property groups necessary to run the instance. See `scf_snapshot_create(3SCF)` and `scf_iter_instance_snapshots(3SCF)`.

See `smf(5)` for information about instances.

An `scf_instance_t` is an opaque handle that can be set to a single instance at any given time. The `scf_instance_create()` function allocates and initializes a new `scf_instance_t` bound to *handle*. The `scf_instance_destroy()` function destroys and frees *inst*.

The `scf_instance_handle()` function retrieves the handle to which *inst* is bound.

The `scf_inst_get_parent()` function sets *svc* to the service that is the parent of *inst*.

The `scf_instance_get_name()` function retrieves the name of the instance to which *inst* is set.

The `scf_service_get_instance()` function sets *inst* to the child instance of the service *svc* specified by *name*.

The `scf_service_add_instance()` function sets *inst* to a new child instance of the service *svc* specified by *name*.

The `scf_instance_delete()` function deletes the instance to which *inst* is set, as well all of the children of the instance.

Return Values Upon successful completion, `scf_instance_create()` returns a new `scf_instance_t`. Otherwise it returns `NULL`.

Upon successful completion, `scf_instance_handle()` returns the handle to which *inst* is bound. Otherwise, it returns `NULL`.

Upon successful completion, `scf_instance_get_name()` returns the length of the string written, not including the terminating null character. Otherwise it returns -1.

Upon successful completion, `scf_instance_get_parent()`, `scf_service_get_instance()`, `scf_service_add_instance()`, and `scf_instance_delete()` functions return 0. Otherwise, they return -1.

Errors The `scf_instance_create()` function will fail if:

`SCF_ERROR_HANDLE_DESTROYED`
An object was bound to a destroyed handle.

`SCF_ERROR_INTERNAL`
An internal error occurred.

`SCF_ERROR_INVALID_ARGUMENT`
The *handle* argument is `NULL`.

`SCF_ERROR_NO_MEMORY`
There is not enough memory to allocate an `scf_instance_t`.

`SCF_ERROR_NO_RESOURCES`
The server does not have adequate resources for a new instance handle.

The `scf_instance_handle()` function will fail if:

`SCF_ERROR_HANDLE_DESTROYED`
The handle associated with *inst* has been destroyed.

The `scf_instance_get_name()`, `scf_instance_get_parent()`, and `scf_instance_delete()` functions will fail if:

`SCF_ERROR_DELETED`	The instance has been deleted.
`SCF_ERROR_NOT_SET`	The instance is not set.

SCF_ERROR_NOT_BOUND	The repository handle is not bound.
SCF_ERROR_CONNECTION_BROKEN	The connection to the repository was lost.

The scf_service_add_instance() function will fail if:

SCF_ERROR_EXISTS
An instance named *name* already exists.

SCF_ERROR_INTERNAL
An internal error occurred.

SCF_ERROR_NO_RESOURCES
The server does not have the resources to complete the request.

SCF_ERROR_NOT_BOUND
The handle is not bound.

The scf_service_get_instance() function will fail if:

SCF_ERROR_BACKEND_ACCESS	The storage mechanism that the repository server (svc.configd(1M)) chose for the operation denied access.
SCF_ERROR_INTERNAL	An internal error occurred.
SCF_ERROR_NOT_BOUND	The handle is not bound.
SCF_ERROR_NOT_FOUND	No instance specified by *name* was found.
SCF_ERROR_NO_RESOURCES	The repository server is out of resources.

The scf_service_add_instance() and scf_service_get_instance() functions will fail if:

SCF_ERROR_NOT_SET
The service is not set.

SCF_ERROR_DELETED
The service has been deleted.

SCF_ERROR_INVALID_ARGUMENT
The *name* argument is not a valid instance name.

SCF_ERROR_HANDLE_MISMATCH
The service and instance are not derived from the same handle.

SCF_ERROR_CONNECTION_BROKEN
The connection to the repository was lost.

The scf_instance_get_parent() function will fail if:

SCF_ERROR_HANDLE_MISMATCH
The *service* and *instance* arguments are not derived from the same handle.

The scf_service_add_instance() and scf_instance_delete() functions will fail if:

`SCF_ERROR_PERMISSION_DENIED`
The user does not have sufficient privileges to create or delete an instance.

`SCF_ERROR_BACKEND_READONLY`
The repository backend is read-only.

`SCF_ERROR_BACKEND_ACCESS`
The repository backend refused the modification.

The `scf_instance_delete()` function will fail if:

`SCF_ERROR_NO_RESOURCES` The server does not have adequate resources for a new instance handle.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also

`libscf(3LIB)`, `scf_error(3SCF)`, `scf_iter_instance_pgs(3SCF)`, `scf_iter_instance_pgs_typed(3SCF)`, `scf_iter_instance_snapshots(3SCF)`, `scf_pg_create(3SCF)`, `scf_snapshot_create(3SCF)`, `attributes(5)`, `smf(5)`

Notes

Instance names are of the form:

[*domain*.]*identifier*

where *domain* is either a stock ticker symbol such as SUNW or a Java-style reversed domain name such as `com.sun`. Identifiers begin with a letter or underscore and contain only letters, digits, underscores, and dashes.

Name scf_iter_create, scf_iter_handle, scf_iter_destroy, scf_iter_reset, scf_iter_handle_scopes, scf_iter_scope_services, scf_iter_service_instances, scf_iter_service_pgs, scf_iter_service_pgs_typed, scf_iter_instance_snapshots, scf_iter_snaplevel_pgs, scf_iter_snaplevel_pgs_typed, scf_iter_instance_pgs, scf_iter_instance_pgs_typed, scf_iter_instance_pgs_composed, scf_iter_instance_pgs_typed_composed, scf_iter_pg_properties, scf_iter_property_values, scf_iter_next_scope, scf_iter_next_service, scf_iter_next_instance, scf_iter_next_snapshot, scf_iter_next_pg, scf_iter_next_property, scf_iter_next_value – iterate through the Service Configuration Facility repository

Synopsis

```
cc [ flag... ] file... -lscf [ library... ]
#include <libscf.h>

scf_iter_t *scf_iter_create(scf_handle_t *handle);

scf_handle_t *scf_iter_handle(scf_iter_t *iter);

void scf_iter_destroy(scf_iter_t *iter);

void scf_iter_reset(scf_iter_t *iter);

int scf_iter_handle_scopes(scf_iter_t *iter, const scf_handle_t *h);

int scf_iter_scope_services(scf_iter_t *iter, const scf_scope_t *parent);

int scf_iter_service_instances(scf_iter_t *iter,
     const scf_service_t *parent);

int scf_iter_service_pgs(scf_iter_t *iter, const scf_service_t *parent);

int scf_iter_service_pgs_typed(scf_iter_t *iter,
     const scf_service_t *parent, const char *pgtype);

int scf_iter_instance_snapshots(scf_iter_t *iter,
     const scf_instance_t *parent);

int scf_iter_snaplevel_pgs(scf_iter_t *iter,
     const scf_snaplevel_t *parent);

int scf_iter_snaplevel_pgs_typed(scf_iter_t *iter,
     const scf_snaplevel_t *parent, const char *pgtype);

int scf_iter_instance_pgs(scf_iter_t *iter, scf_instance_t *parent);

int scf_iter_instance_pgs_typed(scf_iter_t *iter,
     scf_instance_t *parent, const char *pgtype);

int scf_iter_instance_pgs_composed(scf_iter_t *iter,
     const scf_instance_t *instance, const scf_snapshot_t *snapshot);

int scf_iter_instance_pgs_typed_composed(scf_iter_t *iter,
     const scf_instance_t *instance, const scf_snapshot_t *snapshot,
     const char *pgtype);

int scf_iter_pg_properties(scf_iter_t *iter,
     const scf_propertygroup_t *parent);
```

```
int scf_iter_property_values(scf_iter_t *iter,
     const scf_property_t *parent);

int scf_iter_next_scope(scf_iter_t *iter, scf_scope_t *out);

int scf_iter_next_service(scf_iter_t *iter, scf_service_t *out);

int scf_iter_next_instance(scf_iter_t *iter, scf_instance_t *out);

int scf_iter_next_snapshot(scf_iter_t *iter, scf_snapshot_t *out);

int scf_iter_next_pg(scf_iter_t *iter, scf_propertygroup_t  *out);

int scf_iter_next_property(scf_iter_t *iter, scf_property_t *out);

int scf_iter_next_value(scf_iter_t *iter, scf_value_t *out);
```

Description The `scf_iter_create()` function creates a new iterator associated with *handle*. The `scf_iter_destroy()` function destroys an iteration.

The `scf_iter_reset()` function releases any resources involved with an active iteration and returns the iterator to its initial state.

The `scf_iter_handle_scopes()`, `scf_iter_scope_services()`, `scf_iter_service_instances()`, `scf_iter_instance_snapshots()`, `scf_iter_service_pgs()`, `scf_iter_instance_pgs()`, `scf_iter_snaplevel_pgs()`, `scf_iter_pg_properties()`, and `scf_iter_property_values()` functions set up a new iteration of all the children *parent* of a particular type. The `scf_iter_property_values()` function will iterate over values in the order in which they were specified with `scf_entry_add_value(3SCF)`.

The `scf_iter_service_pgs_typed()`, `scf_iter_instance_pgs_typed()`, and `scf_iter_snaplevel_pgs_typed()` functions iterate over the child property groups of *parent*, but restrict them to a particular property group type.

The `scf_iter_instance_pgs_composed()` function sets up a new iteration of the composed view of instance's children at the time *snapshot* was taken. If *snapshot* is `NULL`, the current properties are used. The composed view of an instance's properties is the union of the properties of the instance and its ancestors. Properties of the instance take precedence over properties of the service with the same name, including property group name. Property groups retrieved with this iterator might not have *instance* as their parent and properties retrieved from such property groups might not have the indicated property group as their parent. If *instance* and its parent have property groups with the same name but different types, the properties in the property group of the parent are excluded. The `scf_iter_instance_pgs_typed_composed()` function behaves as `scf_iter_instance_pgs_composed()`, except the property groups of the type *pgtype* are returned.

The `scf_iter_next_scope()`, `scf_iter_next_service()`, `scf_iter_next_instance()`, `scf_iter_next_snapshot()`, `scf_iter_next_pg()`, `scf_iter_next_property()`, and `scf_iter_next_value()` functions retrieve the next element of the iteration.

Return Values Upon successful completion, `scf_iter_create()` returns a pointer to a new iterator. Otherwise, it returns `NULL`.

Upon successful completion, `scf_iter_handle()` returns the handle associated with *iter*. Otherwise it returns `NULL`.

Upon successful completion, `scf_iter_handle_scopes()`, `scf_iter_scope_services()`, `scf_iter_service_instances()`, `scf_iter_instance_snapshots()`, `scf_iter_service_pgs()`, `scf_iter_instance_pgs()`, `scf_iter_snaplevel_pgs()`, `scf_iter_pg_properties()`, `scf_iter_property_values()`, `scf_iter_service_pgs_typed()`, `scf_iter_instance_pgs_composed()`, `scf_iter_instance_pgs_typed()`, `scf_iter_instance_pgs_typed_composed()`, and `scf_iter_snaplevel_pgs_typed()` return 0. Otherwise, they return -1.

Upon successful completion, `scf_iter_next_scope()`, `scf_iter_next_service()`, `scf_iter_next_instance()`, `scf_iter_next_snapshot()`, `scf_iter_next_pg()`, `scf_iter_next_property()`, and `scf_iter_next_value()` return 1. If the iterator is complete, they return 0. Otherwise, they return -1.

Errors The `scf_iter_create()` function will fail if:

`SCF_ERROR_INVALID_ARGUMENT`	The handle argument is NULL.
`SCF_ERROR_NO_MEMORY`	There is no memory available.
`SCF_ERROR_NO_RESOURCES`	The server does not have adequate resources for a new iteration.

The `scf_iter_handle()` function will fail if:

`SCF_ERROR_HANDLE_DESTROYED`	The handle associated with *iter* has been destroyed.

The `scf_iter_next_value()` function will fail if:

`SCF_ERROR_PERMISSION_DENIED`	The value could not be read due to access restrictions.

The `scf_iter_handle_scopes()`, `scf_iter_scope_services()`, `scf_iter_service_instances()`, `scf_iter_instance_snapshots()`, `scf_iter_service_pgs()`, `scf_iter_instance_pgs()`, `scf_iter_instance_pgs_composed()`, `scf_iter_snaplevel_pgs()`, `scf_iter_pg_properties()`, `scf_iter_property_values()`, `scf_iter_service_pgs_typed()`, `scf_iter_instance_pgs_typed()`, `scf_iter_instance_pgs_typed_composed()`, and `scf_iter_snaplevel_pgs_typed()` functions will fail if:

`SCF_ERROR_DELETED`	The parent has been deleted.
`SCF_ERROR_NOT_SET`	The parent is not set.
`SCF_ERROR_NOT_BOUND`	The handle is not bound.

`SCF_ERROR_CONNECTION_BROKEN` The connection to the repository was lost.

`SCF_ERROR_HANDLE_MISMATCH` The *iter* and *parent* arguments are not derived from the same handle.

The `scf_iter_service_pgs_typed()`, `scf_iter_instance_pgs_typed()`, `scf_iter_instance_pgs_typed_composed()`, and `scf_iter_snaplevel_pgs_typed()` functions will fail if:

`SCF_ERROR_INVALID_ARGUMENT` The *pgtype* argument is not a valid property group type.

The `scf_iter_next_service()`, `scf_iter_next_instance()`, `scf_iter_next_snapshot()`, `scf_iter_next_pg()`, `scf_iter_next_property()`, and `scf_iter_next_value()` functions will fail if:

`SCF_ERROR_DELETED` The parent the iterator is attached to has been deleted.

The `scf_iter_next_scope()`, `scf_iter_next_service()`, `scf_iter_next_instance()`, `scf_iter_next_snapshot()`, `scf_iter_next_pg()`,`scf_iter_next_property()`, and `scf_iter_next_value()` functions will fail if:

`SCF_ERROR_NOT_SET` The iterator is not set.

`SCF_ERROR_INVALID_ARGUMENT` The requested object type does not match the type the iterator is walking.

`SCF_ERROR_NOT_BOUND` The handle is not bound.

`SCF_ERROR_HANDLE_MISMATCH` The *iter* and *parent* arguments are not derived from the same handle.

`SCF_ERROR_CONNECTION_BROKEN` The connection to the repository was lost.

The `scf_iter_scope_services()`, `scf_iter_service_instances()`, `scf_iter_service_pgs()`, `scf_iter_instance_snapshots()`, `scf_iter_instance_pgs()`, `scf_iter_instance_pgs_composed()`, `scf_iter_snaplevel_pgs()`, `scf_iter_pg_properties()`, `scf_iter_property_values()`, `scf_iter_service_pgs_typed()`, `scf_iter_instance_pgs_typed()`, `scf_iter_instance_pgs_typed_composed()`, `scf_iter_snaplevel_pgs_typed()`, `scf_iter_next_service()`, `scf_iter_next_instance()`, `scf_iter_next_snapshot()`, `scf_iter_next_pg()`, and `scf_iter_next_property()` functions will fail if:

`SCF_ERROR_NO_RESOURCES` The server does not have the resources to complete the request.

The `scf_error(3SCF)` function can be used to retrieve the error value.

Examples

EXAMPLE 1 Iterate over all instances under a service.

```
scf_iter_t *iter = scf_iter_create(handle);

if (iter == NULL || scf_iter_service_instances(iter, parent) == -1) {
     /* failure */
}
while ((r = scf_iter_next_instance(iter, child)) > 0) {
     /* process child */
}
if (r < 0) {
     /* failure */
}
scf_iter_destroy(iter);
```

EXAMPLE 2 Connect to the repository, walk all services and instances and print their FMRIs.

```
scf_handle_t *handle = scf_handle_create(SCF_VERSION);
scf_scope_t *scope = scf_scope_create(handle);
scf_service_t *svc = scf_service_create(handle);
scf_instance_t *inst = scf_instance_create(handle);
scf_iter_t *svc_iter = scf_iter_create(handle);
scf_iter_t *inst_iter = scf_iter_create(handle);

size_t sz = scf_limit(SCF_LIMIT_MAX_FMRI_LENGTH) + 1;
char *fmri = malloc(sz + 1);

int r;

if (handle == NULL || scope == NULL || svc == NULL ||
     inst == NULL || svc_iter == NULL || inst_iter == NULL ||
     fmri == NULL) {
         /* failure */
}
if (scf_handle_bind(handle) == -1 ||
     scf_handle_get_scope(handle, SCF_SCOPE_LOCAL, scope) == -1 ||
     scf_iter_scope_services(svc_iter, scope) == -1) {
          /* failure */
}
while ((r = scf_iter_next_service(svc_iter, svc)) > 0) {
     if (scf_service_to_fmri(svc, fmri, sz) < 0) {
          /* failure */
     }
     puts(fmri);
     if (scf_iter_service_instances(inst_iter, svc) < 0) {
          /* failure */
     }
     while ((r = scf_iter_next_instance(inst_iter, inst)) > 0) {
```

EXAMPLE 2 Connect to the repository, walk all services and instances and print their FMRIs. *(Continued)*

```
          if (scf_instance_to_fmri(inst, fmri, sz) < 0) {
               /* failure */
          }
          puts(fmri);
     }
     if (r < 0)
          break;
}
if (r < 0) {
     /* failure */
}

scf_handle_destroy(handle);
scf_scope_destroy(scope);
scf_service_destroy(svc);
scf_instance_destroy(inst);
scf_iter_destroy(svc_iter);
scf_iter_destroy(inst_iter);
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also

libscf(3LIB), scf_entry_add_value(3SCF), scf_error(3SCF), scf_handle_create(3SCF), attributes(5)

Name scf_limit – limit information for Service Configuration Facility

Synopsis

```
cc [ flag... ] file... -lscf [ library... ]
#include <libscf.h>

ssize_t scf_limit(uint32_t name);
```

Description The `scf_limit()` function returns information about implementation-defined limits in the service configuration facility. These limits are generally maximum lengths for various strings. The values returned do not change during the execution of a program, but they should not be cached between executions.

The available values for *name* are:

`SCF_LIMIT_MAX_FMRI_LENGTH`	Return the maximum length of an FMRI the service configuration facility accepts.
`SCF_LIMIT_MAX_PG_TYPE_LENGTH`	Return the maximum length for property group types in the service configuration facility.
`SCF_LIMIT_MAX_NAME_LENGTH`	Return the maximum length for names in the service configuration facility. This value does not include space for the required terminating null byte.
`SCF_LIMIT_MAX_VALUE_LENGTH`	Return the maximum string length a scf_value_t can hold, not including the terminating null byte.

Lengths do not include space for the required terminating null byte.

Return Values Upon successful completion, `scf_limit()` returns the requested value. Otherwise, it returns -1.

Errors The `scf_limit()` function will fail if:

`SCF_ERROR_INVALID_ARGUMENT` The *name* argument is not a recognized request.

The `scf_error(3SCF)` function can be used to retrieve the error value.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also `libscf(3LIB)`, `scf_error(3SCF)`, `attributes(5)`

Name scf_pg_create, scf_pg_handle, scf_pg_destroy, scf_pg_get_parent_service, scf_pg_get_parent_instance, scf_pg_get_parent_snaplevel, scf_pg_get_name, scf_pg_get_type, scf_pg_get_flags, scf_pg_update, scf_service_get_pg, scf_service_add_pg, scf_instance_get_pg, scf_instance_get_pg_composed, scf_instance_add_pg, scf_snaplevel_get_pg, scf_pg_delete, scf_pg_get_underlying_pg – create and manipulate property group handles and property groups in the Service Configuration Facility

Synopsis

```
cc [ flag... ] file... -lscf [ library... ]
#include <libscf.h>

scf_propertygroup_t *scf_pg_create(scf_handle_t *handle);

scf_handle_t *scf_pg_handle(scf_propertygroup_t *pg);

void scf_pg_destroy(scf_propertygroup_t *pg);

int scf_pg_get_parent_service(const scf_propertygroup_t *pg,
     scf_service_t *svc);

int scf_pg_get_parent_instance(const scf_propertygroup_t *pg,
     scf_instance_t *inst);

int scf_pg_get_parent_snaplevel(const scf_propertygroup_t *pg,
     scf_snaplevel_t *level);

ssize_t scf_pg_get_name(const scf_propertygroup_t *pg, char *buf,
     size_t size);

ssize_t scf_pg_get_type(const scf_propertygroup_t *pg, char *buf,
     size_t size);

int scf_pg_get_flags(const scf_propertygroup_t *pg, uint32_t *out);

int scf_pg_update(const scf_propertygroup_t *pg);

int scf_service_get_pg(const scf_service_t *svc, const char *name,
     scf_propertygroup_t *pg);

int scf_service_add_pg(const scf_service_t *svc,
     const char *name, const char *group_type,
     uint32_t flags, scf_propertygroup_t *pg);

int scf_instance_get_pg(const scf_instance_t *inst,
     const char *name, scf_propertygroup_t *pg);

int scf_instance_get_pg_composed(const scf_instance_t *inst,
     const scf_snapshot_t *snapshot, const char *name,
     scf_propertygroup_t *pg);

int scf_instance_add_pg(const scf_instance_t *inst,
     const char *name, const char *group_type, uint32_t flags,
     scf_propertygroup_t *pg);

int scf_snaplevel_get_pg(const scf_snaplevel_t *level,
     const char *name, scf_propertygroup_t *pg);
```

```
int scf_pg_delete(scf_propertygroup_t *pg);

int scf_pg_get_underlying_pg(const scf_propertygroup_t *pg,
     scf_propertygroup_t *out);
```

Description Property groups are an atomically-updated group of typed properties. Property groups of services (see `scf_service_create(3SCF)`) or instances (see `scf_instance_create(3SCF)`) are modifiable. Property groups of snaplevels (see `scf_snaplevel_create(3SCF)`) are not modifiable.

An `scf_propertygroup_t` is an opaque handle that can be set to a single property group at any given time. When an `scf_propertygroup_t` is set, it references a frozen-in-time version of the property group to which it is set. Updates to the property group will not be visible until either `scf_pg_update()` is called or the property group is set again.

This static view is propagated to the `scf_property_t`s set to children of the property group. They will not see updates, even if the `scf_propertygroup_t` is updated.

The `scf_pg_create()` function allocates and initializes a new `scf_propertygroup_t` bound to *handle*. The `scf_pg_destroy()` function destroys and frees *pg*.

The `scf_pg_handle()` function retrieves the handle to which *pg* is bound.

The `scf_pg_get_parent_service()`, `scf_pg_get_parent_instance()`, and `scf_pg_get_parent_snaplevel()` functions retrieve the property group's parent, if it is of the requested type.

The `scf_pg_get_name()` and `scf_pg_get_type()` functions retrieve the name and type, respectively, of the property group to which *pg* is set.

The `scf_pg_get_flags()` function retrieves the flags for the property group to which *pg* is set. If `SCF_PG_FLAG_NONPERSISTENT` is set, the property group is not included in snapshots and will lose its contents upon system shutdown or reboot. Non-persistent property groups are mainly used for smf-internal state. See `smf(5)`.

The `scf_pg_update()` function ensures that *pg* is attached to the most recent version of the *pg* to which it is set.

The `scf_service_get_pg()`, `scf_instance_get_pg()`, and `scf_snaplevel_get_pg()` functions set *pg* to the property group specified by *name* in the service specified by *svc*, the instance specified by *inst*, or the snaplevel specified by *level*, respectively.

The `scf_instance_get_pg_composed()` function sets *pg* to the property group specified by *name* in the composed view of *inst* at the time *snapshot* was taken. If *snapshot* is `NULL`, the current properties are used. The composed view of an instance's properties is the union of the properties of the instance and its ancestors. Properties of the instance take precedence over properties of the service with the same name (including the property group name). After a successful call to `scf_instance_get_pg_composed()`, the parent of *pg* might not be *inst*, and

the parents of properties obtained from *pg* might not be *pg*. If *inst* and its parent have property groups with the same name but different types, the properties in the property group of the parent are excluded.

The `scf_service_add_pg()` and `scf_instance_add_pg()` functions create a new property group specified by *name* whose type is *group_type*, and attach the *pg* handle (if non-null) to the new object. The *flags* argument must be either 0 or `SCF_PG_FLAG_NONPERSISTENT`.

The `scf_pg_delete()` function deletes the property group. Versions of the property group in snapshots are not affected.

The `scf_pg_get_underlying_pg()` function gets the first existing underlying property group. If the property group specified by *pg* is an instance property group, *out* is set to the property group of the same name in the instance's parent.

Applications can use a transaction to modify a property group. See `scf_transaction_create(3SCF)`.

Return Values Upon successful completion, `scf_pg_create()` returns a new `scf_propertygroup_t`. Otherwise, it returns `NULL`.

Upon successful completion, `scf_pg_handle()` returns a pointer to the handle to which *pg* is bound. Otherwise, it returns NULL.

Upon successful completion, `scf_instance_handle()` returns the handle instance with which it is associated. Otherwise, it returns `NULL`.

Upon successful completion, `scf_pg_get_name()` and `scf_pg_get_type()` return the length of the string written, not including the terminating null byte. Otherwise, they return -1.

The `scf_pg_update()` function returns 1 if the object was updated, 0 if the object was already up to date, and -1 on failure.

Upon successful completion, `scf_pg_get_parent_service()`, `scf_pg_get_parent_snaplevel()`, `scf_pg_get_flags()`, `scf_service_get_pg()`, `scf_service_add_pg()`, `scf_pg_get_parent_instance()`, `scf_instance_get_pg()`, `scf_instance_get_pg_composed()`, `scf_instance_add_pg()`, `scf_snaplevel_get_pg()`, `scf_pg_delete()`, and `scf_pg_get_underlying_pg()` return 0. Otherwise, they return -1.

Errors The `scf_pg_create()` function will fail if:

`SCF_ERROR_INVALID_ARGUMENT`
The *handle* argument is `NULL`.

`SCF_ERROR_NO_MEMORY`
There is not enough memory to allocate an `scf_propertygroup_t`.

`SCF_ERROR_NO_RESOURCES`
The server does not have adequate resources for a new property group handle.

The `scf_pg_handle()` function will fail if:

`SCF_ERROR_HANDLE_DESTROYED`
The handle associated with *pg* has been destroyed.

The `scf_pg_update()` function will fail if:

`SCF_ERROR_CONNECTION_BROKEN`
The connection to the repository was lost.

`SCF_ERROR_DELETED`
An ancestor of the property group specified by *pg* has been deleted.

`SCF_ERROR_INTERNAL`
An internal error occurred. This can happen if *pg* has been corrupted.

`SCF_ERROR_INVALID_ARGUMENT`
The *pg* argument refers to an invalid `scf_propertygroup_t`.

`SCF_ERROR_NOT_BOUND`
The handle is not bound.

`SCF_ERROR_NOT_SET`
The property group specified by *pg* is not set.

The `scf_service_get_pg()`, `scf_instance_get_pg()`, `scf_instance_get_pg_composed()`, `scf_snaplevel_get_pg()`, and `scf_pg_get_underlying_pg()` functions will fail if:

`SCF_ERROR_BACKEND_ACCESS`
The storage mechanism that the repository server (`svc.configd(1M)`) chose for the operation denied access.

`SCF_ERROR_INTERNAL`
An internal error occurred.

`SCF_ERROR_NO_RESOURCES`
The server does not have the resources to complete the request.

The `scf_pg_get_name()`, `scf_pg_get_type()`, `scf_pg_get_flags()`, `scf_pg_get_parent_service()`, `scf_pg_get_parent_snaplevel()`, and `scf_pg_get_parent_instance()` functions will fail if:

`SCF_ERROR_DELETED`
The property group specified by *pg* has been deleted.

`SCF_ERROR_NOT_SET`
The property group specified by *pg* is not set.

`SCF_ERROR_NOT_BOUND`
The handle is not bound.

SCF_ERROR_CONNECTION_BROKEN
The connection to the repository was lost.

The scf_pg_get_parent_service(), scf_pg_get_parent_snaplevel(), and scf_pg_get_parent_instance() functions will fail if:

SCF_ERROR_CONSTRAINT_VIOLATED
The requested parent type does not match the actual type of the parent of the property group specified by *pg*.

SCF_ERROR_HANDLE_MISMATCH
The property group and either the instance, the service, or the snaplevel are not derived from the same handle.

The scf_instance_get_pg(), scf_instance_get_pg_composed(), scf_service_get_pg(), scf_pg_get_underlying_pg(), and scf_snaplevel_get_pg() functions will fail if:

SCF_ERROR_NOT_FOUND
The property group specified by *name* was not found.

The scf_service_add_pg(), scf_service_get_pg(), scf_instance_add_pg(), scf_instance_get_pg(), scf_instance_get_pg_composed(), and scf_snaplevel_get_pg() functions will fail if:

SCF_ERROR_DELETED
The service or instance has been deleted.

SCF_ERROR_NOT_SET
The instance is not set.

SCF_ERROR_INVALID_ARGUMENT
The value of the *name* argument is not a valid property group name.

SCF_ERROR_HANDLE_MISMATCH
The property group and either the instance, the service, or the level are not derived from the same handle.

SCF_ERROR_NOT_BOUND
The handle is not bound.

SCF_ERROR_CONNECTION_BROKEN
The connection to the repository was lost.

The scf_service_add_pg() and scf_instance_add_pg() functions will fail if:

SCF_ERROR_PERMISSION_DENIED
The caller does not have permission to create the requested property group.

SCF_ERROR_BACKEND_READONLY
The repository backend is read-only.

`SCF_ERROR_BACKEND_ACCESS`
The repository backend refused the modification.

`SCF_ERROR_EXISTS`
A {service,instance,property group} named *name* already exists.

`SCF_ERROR_NO_RESOURCES`
The server does not have the resources to complete the request.

The `scf_pg_delete()` function will fail if:

`SCF_ERROR_BACKEND_ACCESS`
The repository backend refused the modification.

`SCF_ERROR_BACKEND_READONLY`
The repository backend is read-only.

`SCF_ERROR_CONNECTION_BROKEN`
The connection to the repository was lost.

`SCF_ERROR_DELETED`
The property group has been deleted by someone else.

`SCF_ERROR_NO_RESOURCES`
The server does not have adequate resources for a new property group handle.

`SCF_ERROR_NOT_SET`
The property group has not been set.

`SCF_ERROR_PERMISSION_DENIED`
The caller does not have permission to delete this property group.

The `scf_pg_get_underlying_pg()` function will fail if:

`SCF_ERROR_CONNECTION_BROKEN`
The connection to the repository was lost.

`SCF_ERROR_CONSTRAINT_VIOLATED`
A required constraint was not met.

`SCF_ERROR_DELETED`
The property group has been deleted.

`SCF_ERROR_HANDLE_MISMATCH`
The property group and *out* are not derived from the same handle.

`SCF_ERROR_INVALID_ARGUMENT`
An argument is invalid.

`SCF_ERROR_NOT_BOUND`
The handle is not bound.

SCF_ERROR_NOT_SET
The property group has not been set.

The scf_error(3SCF) function can be used to retrieve the error value.

Examples

EXAMPLE 1 Perform a layered lookup of *name* in *pg*.

```
int layered_lookup(scf_propertygroup_t *pg, const char *name,
scf_property_t *out) {
     scf_handle_t *handle = scf_pg_handle(out);
     scf_propertygroup_t *new_pg;
     scf_propertygroup_t *cur, *other;
     int state = 0;

     if (handle == NULL) {
          return (-1);
     }
     new_pg = scf_pg_create(handle);
     if (new_pg == NULL) {
          return (-1);
     }
     for (;;) {
          cur =   state ? pg : new_pg;
          other = state ? new_pg : pg;
          state = !state;

          if (scf_pg_get_property(cur, name, out) != -1) {
               scf_pg_destroy(new_pg);
               return (SUCCESS);
          }
          if (scf_pg_get_underlying_pg(cur, other) == -1)
               break;
     }
     scf_pg_destroy(new_pg);
     return (NOT_FOUND);
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also

libscf(3LIB), scf_error(3SCF), scf_handle_decode_fmri(3SCF), scf_instance_create(3SCF), scf_pg_to_fmri(3SCF), scf_service_create(3SCF), scf_snaplevel_create(3SCF), scf_transaction_create(3SCF), attributes(5), smf(5)

Name scf_property_create, scf_property_handle, scf_property_destroy, scf_property_get_name, scf_property_type, scf_property_is_type, scf_type_to_string, scf_string_to_type, scf_property_get_value, scf_pg_get_property – create and manipulate property handles in the Service Configuration Facility

Synopsis

```
cc [ flag... ] file... -lscf [ library... ]
#include <libscf.h>

scf_property_t *scf_property_create(scf_handle_t *handle);

scf_handle_t *scf_property_handle(scf_property_t *prop);

void scf_property_destroy(scf_property_t *prop);

ssize_t scf_property_get_name(const scf_property_t *prop,
     char *buf, size_t size);

int scf_property_type(const scf_property_t *prop,
     scf_type_t *type);

int scf_property_is_type(const scf_property_t *prop,
     scf_type_t type);

const char *scf_type_to_string(scf_type_t type);

scf_type_t scf_string_to_type(const char *type);

int scf_property_get_value(const scf_property_t *prop,
     scf_value_t *value);

int scf_pg_get_property(const scf_property_t *pg,
     const char *name, scf_property_t *prop);
```

Description Properties are named sets of values of one type. They are grouped into property groups (see `scf_pg_create(3SCF)`) that are updated atomically using transactions (see `scf_transaction_create(3SCF)`).

An `scf_property_t` is an opaque handle that can be set to a single property at any given time. When set, it inherits the point-in-time from the source `scf_propertygroup_t` and does not change until reset.

The `scf_property_create()` function allocates and initializes a new `scf_property_t` bound to *handle*. The `scf_property_destroy()` function destroys and frees *prop*.

The `scf_property_handle()` function returns the handle to which *prop* is bound.

The `scf_property_type()` function retrieves the type of the property to which *prop* is set.

The `scf_property_is_type()` function determines if the property is compatible with type. See `scf_value_create(3SCF)`.

The `scf_type_to_string()` function returns the string name of the type supplied. If the type is invalid or unknown, it returns “unknown”.

The `scf_string_to_type()` function returns the `scf_type_t` definition of the string supplied. If the string does not translate to an existing type, it returns `SCF_TYPE_INVALID`.

The `scf_property_get_value()` function retrieves the single value that the property to which *prop* is set contains. If the property has more than one value, the *value* argument is set to one of the values. To retrieve all values associated with a property, see `scf_iter_property_values(3SCF)`.

The `scf_pg_get_property()` function sets *prop* to the property specified by *name* in the property group specified by *pg*.

Return Values Upon successful completion, `scf_property_create()` returns a new `scf_property_t`. Otherwise, it returns `NULL`.

Upon successful completion, `scf_property_get_name()` function returns the length of the string written, not including the terminating null byte. Otherwise, it returns -1.

Upon successful completion, `scf_pg_get_property()`, `scf_property_type()`, `scf_property_is_type()`, and `scf_pg_get_value()` functions return 0. Otherwise, they return -1.

Upon successful completion, `scf_type_to_string()` returns a string of the type supplied.

Upon successful completion, `scf_string_to_type()` returns the `scf_type_t` definition of the string supplied

Errors The `scf_property_create()` function will fail if:

`SCF_ERROR_INVALID_ARGUMENT`	The value of the *handle* argument is `NULL`.
`SCF_ERROR_NO_MEMORY`	There is not enough memory to allocate an `scf_property_t`.
`SCF_ERROR_NO_RESOURCES`	The server does not have adequate resources for a new property handle.

The `scf_property_handle()` function will fail if:

`SCF_ERROR_HANDLE_DESTROYED`	The handle associated with *prop* has been destroyed.

The `scf_property_get_name()`, `scf_property_type()`, `scf_property_is_type()`, and `scf_property_get_value()` functions will fail if:

`SCF_ERROR_CONNECTION_BROKEN`	The connection to the repository was lost.
`SCF_ERROR_DELETED`	The property's parent property group or an ancestor has been deleted.
`SCF_ERROR_NOT_BOUND`	The handle was never bound or has been unbound.

SCF_ERROR_NOT_SET	The property is not set.

The scf_property_is_type() function will fail if:

SCF_ERROR_INVALID_ARGUMENT	The *type* argument is not a valid type.
SCF_ERROR_TYPE_MISMATCH	The *prop* argument is not of a type compatible with *type*.

The scf_pg_get_property() function will fail if:

SCF_ERROR_BACKEND_ACCESS	The storage mechanism that the repository server (svc.configd(1M)) chose for the operation denied access.
SCF_ERROR_CONNECTION_BROKEN	The connection to the repository was lost.
SCF_ERROR_DELETED	The property group or an ancestor has been deleted.
SCF_ERROR_HANDLE_MISMATCH	The property group and property are not derived from the same handle.
SCF_ERROR_INTERNAL	An internal error occurred.
SCF_ERROR_INVALID_ARGUMENT	The value of the *name* argument is not a valid property name.
SCF_ERROR_NO_RESOURCES	The server does not have the resources to complete the request.
SCF_ERROR_NOT_BOUND	The handle was never bound or has been unbound.
SCF_ERROR_NOT_FOUND	The property specified by *name* was not found.
SCF_ERROR_NOT_SET	The property group specified by *pg* is not set.

The scf_property_get_value() function will fail if:

SCF_ERROR_CONSTRAINT_VIOLATED	The property has more than one value associated with it. The *value* argument will be set to one of the values.
SCF_ERROR_HANDLE_MISMATCH	The property and value are derived from different handles.
SCF_ERROR_NOT_FOUND	The property has no values associated with it. The *value* argument will be reset.
SCF_ERROR_PERMISSION_DENIED	The value could not be read due to access restrictions.

The scf_error(3SCF) function can be used to retrieve the error value.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also libscf(3LIB), scf_error(3SCF), scf_handle_decode_fmri(3SCF), scf_iter_property_values(3SCF), scf_pg_create(3SCF), scf_property_to_fmri(3SCF), scf_transaction_create(3SCF), scf_value_create(3SCF), attributes(5)

Name scf_scope_create, scf_scope_handle, scf_scope_destroy, scf_scope_get_name, scf_handle_get_scope – create and manipulate scope handles in the Service Configuration Facility

Synopsis

```
cc [ flag... ] file... -lscf [ library... ]
#include <libscf.h>

scf_scope_t *scf_scope_create(scf_handle_t *handle);

scf_handle_t *scf_scope_handle(scf_scope_t *sc);

void scf_scope_destroy(scf_scope_t *sc);

ssize_t  scf_scope_get_name(scf_scope_t *sc, char *buf, size_t size);

int scf_handle_get_scope(scf_handle_t *handle, const char *name,
     scf_scope_t *out);
```

Description Scopes are the top level of the Service Configuration Facility's repository tree. The children of a scope are services (see `scf_service_create(3SCF)`) and can be walked using `scf_iter_scope_services(3SCF)`.

There is a distinguished scope with the name `SCF_SCOPE_LOCAL` that is the root for all available services on the local machine. In the current implementation, there are no other scopes.

An `scf_scope_t` is an opaque handle that can be set to a single scope at any given time. The `scf_scope_create()` function allocates a new `scf_scope_t` bound to *handle*. The `scf_scope_destroy()` function destroys and frees *sc*.

The `scf_scope_handle()` function retrieves the handle to which *sc* is bound.

The `scf_scope_get_name()` function retrieves the name of the scope to which *sc* is set.

The `scf_handle_get_scope()` function sets *out* to the scope specified by *name* for the repository handle specified by *handle*. The `scf_iter_handle_scopes(3SCF)` and `scf_iter_next_scope(3SCF)` calls can be used to iterate through all available scopes.

Return Values Upon successful completion, `scf_scope_create()` returns a new `scf_scope_t`. Otherwise, it returns `NULL`.

Upon successful completion, `scf_scope_handle()` returns the handle to which *sc* is bound. Otherwise, it returns `NULL`.

Upon successful completion, `scf_scope_get_name()` returns the length of the string written, not including the terminating null byte. Otherwise, it returns -1.

Upon successful completion, `scf_handle_get_scope()` returns 0. Otherwise, it returns -1.

Errors The `scf_scope_create()` function will fail if:

`SCF_ERROR_INVALID_ARGUMENT`	The value of the *handle* argument is `NULL`.
`SCF_ERROR_NO_MEMORY`	There is not enough memory to allocate an `scf_scope_t`.
`SCF_ERROR_NO_RESOURCES`	The server does not have adequate resources for a new scope handle.

The `scf_scope_handle()` function will fail if:

`SCF_ERROR_HANDLE_DESTROYED`	The handle associated with *sc* has been destroyed.

The `scf_scope_get_name()` function will fail if:

`SCF_ERROR_NOT_SET`	The scope is not set.
`SCF_ERROR_NOT_BOUND`	The handle is not bound.
`SCF_ERROR_CONNECTION_BROKEN`	The connection to the repository was lost.

The `scf_handle_get_scope()` function will fail if:

`SCF_ERROR_NOT_FOUND`	No scope named name was found.
`SCF_ERROR_INVALID_ARGUMENT`	The *name* argument is not a valid scope name.
`SCF_ERROR_NOT_BOUND`	The handle is not bound.
`SCF_ERROR_CONNECTION_BROKEN`	The connection to the repository was lost.
`SCF_ERROR_HANDLE_MISMATCH`	The value of the *out* argument is not derived from handle.

The `scf_error(3SCF)` function can be used to retrieve the error value.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also `libscf(3LIB)`, `scf_error(3SCF)`, `scf_handle_decode_fmri(3SCF)`, `scf_iter_handle_scopes(3SCF)`, `scf_iter_next_scope(3SCF)`, `scf_iter_scope_services(3SCF)`, `scf_scope_to_fmri(3SCF)`, `scf_service_create(3SCF)`, `attributes(5)`

Name scf_service_create, scf_service_handle, scf_service_destroy, scf_service_get_parent, scf_service_get_name, scf_scope_get_service, scf_scope_add_service, scf_service_delete – create and manipulate service handles and services in the Service Configuration Facility

Synopsis

```
cc [ flag... ] file... -lscf [ library... ]
#include <libscf.h>

scf_service_t *scf_service_create(scf_handle_t *handle);

scf_handle_t *scf_service_handle(scf_service_t *svc);

void scf_service_destroy(scf_service_t *svc);

int scf_service_get_parent(scf_service_t *svc, scf_scope_t *sc);

ssize_t scf_service_get_name(const scf_service_t *svc, char *buf,
     size_t size);

int scf_scope_get_service(const scf_scope_t *sc, const char *name,
     scf_service_t *svc);

int scf_scope_add_service(const scf_scope_t *sc, const char *name,
     scf_service_t *svc);

int scf_service_delete(scf_service_t *svc);
```

Description Services form the middle layer of the Service Configuration Facility repository tree. Services are children of a scope (see `scf_scope_create(3SCF)`) and have three sets of children:

Property groups	These hold configuration information shared by all of the instances of the service. See `scf_pg_create(3SCF)`, `scf_iter_service_pgs(3SCF)`, and `scf_iter_service_pgs_typed(3SCF)`.
Instances	A particular instantiation of the service. See `scf_instance_create(3SCF)`.

A service groups one or more related instances and provides a shared configuration for them.

An `scf_service_t` is an opaque handle that can be set to a single service at any given time. The `scf_service_create()` function allocates and initializes a new `scf_service_t` bound to *handle*. The `scf_service_destroy()` function destroys and frees svc.

The `scf_service_handle()` function retrieves the handle to which *svc* is bound.

The `scf_service_get_parent()` function sets *sc* to the scope that is the parent of *svc*.

The `scf_service_get_name()` function retrieves the name of the service to which *svc* is set.

The `scf_scope_get_service()` function sets *svc* to the service specified by *name* in the scope specified by *sc*.

The `scf_scope_add_service()` function sets *svc* to a new service specified by *name* in the scope specified by *sc*.

The `scf_service_delete()` function deletes the service to which *svc* is set, as well as all of its children.

Return Values Upon successful completion, `scf_service_create()` returns a new `scf_service_t`. Otherwise, it returns `NULL`.

Upon successful completion, `scf_service_handle()` returns the handle to which svc is bound. Otherwise, it returns `NULL`.

Upon successful completion, `scf_service_get_name()` returns the length of the string written, not including the terminating null byte. Otherwise, it returns -1.

Upon successful completion, `scf_service_get_parent()`, `scf_scope_get_service()`, `scf_scope_add_service()`, and `scf_service_delete()` return 0. Otherwise, it returns -1.

Errors The `scf_service_create()` function will fail if:

`SCF_ERROR_INVALID_ARGUMENT`	The value of the *handle* argument is `NULL`.
`SCF_ERROR_NO_MEMORY`	There is not enough memory to allocate an `scf_service_t`.
`SCF_ERROR_NO_RESOURCES`	The server does not have adequate resources for a new scope handle.

The `scf_service_handle()` function will fail if:

`SCF_ERROR_HANDLE_DESTROYED`	The handle associated with *svc* has been destroyed.

The `scf_service_get_name()`, `scf_service_get_parent()`, and `scf_service_delete()` functions will fail if:

`SCF_ERROR_DELETED`	The service has been deleted by someone else.
`SCF_ERROR_NOT_SET`	The service is not set.
`SCF_ERROR_NOT_BOUND`	The handle is not bound.
`SCF_ERROR_CONNECTION_BROKEN`	The connection to the repository was lost.

The `scf_service_delete()` function will fail if:

`SCF_ERROR_EXISTS`	The service contains instances.
`SCF_ERROR_NO_RESOURCES`	The server does not have adequate resources for a new scope handle.

The `scf_scope_add_service()` function will fail if:

`SCF_ERROR_EXISTS` A {service,instance,property group} named *name* already exists.

The `scf_scope_get_service()` function will fail if:

`SCF_ERROR_BACKEND_ACCESS` The storage mechanism that the repository server (`svc.configd(1M)`) chose for the operation denied access.

`SCF_ERROR_INTERNAL` An internal error occurred.

The `scf_scope_add_service()` and `scf_scope_get_service()` functions will fail if:

`SCF_ERROR_CONNECTION_BROKEN` The connection to the repository was lost.

`SCF_ERROR_DELETED` The parent entity has been deleted.

`SCF_ERROR_HANDLE_MISMATCH` The scope and service are not derived from the same handle.

`SCF_ERROR_INVALID_ARGUMENT` The value of the *name* argument is not a valid service name.

`SCF_ERROR_NO_RESOURCES` The server does not have the resources to complete the request.

`SCF_ERROR_NOT_BOUND` The handle is not bound.

`SCF_ERROR_NOT_FOUND` The service specified by *name* was not found.

`SCF_ERROR_NOT_SET` The scope is not set.

The `scf_scope_add_service()` and `scf_service_delete()` functions will fail if:

`SCF_ERROR_PERMISSION_DENIED` The user does not have sufficient privileges to create or delete a service.

`SCF_ERROR_BACKEND_READONLY` The repository backend is read-only.

`SCF_ERROR_BACKEND_ACCESS` The repository backend refused the modification.

The `scf_error(3SCF)` function can be used to retrieve the error value.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also

`libscf(3LIB)`, `scf_error(3SCF)`, `scf_handle_decode_fmri(3SCF)`, `scf_iter_service_pgs(3SCF)`, `scf_iter_service_pgs_typed(3SCF)`,

scf_instance_create(3SCF), scf_pg_create(3SCF), scf_scope_create(3SCF), scf_service_to_fmri(3SCF), attributes(5), smf(5)

Name scf_simple_prop_get, scf_simple_prop_free, scf_simple_app_props_get, scf_simple_app_props_free, scf_simple_app_props_next, scf_simple_app_props_search, scf_simple_prop_numvalues, scf_simple_prop_type, scf_simple_prop_name, scf_simple_prop_pgname, scf_simple_prop_next_boolean, scf_simple_prop_next_count, scf_simple_prop_next_integer, scf_simple_prop_next_time, scf_simple_prop_next_astring, scf_simple_prop_next_ustring, scf_simple_prop_next_opaque, scf_simple_prop_next_reset – simplified property read interface to Service Configuration Facility

Synopsis

```
cc [ flag... ] file... -lscf [ library... ]
#include <libscf.h>

scf_simple_prop_t *scf_simple_prop_get(scf_handle_t *handle,
     const char *instance, const char *pgname, const char *propname);

void scf_simple_prop_free(scf_simple_prop_t *prop);

scf_simple_app_props_t *scf_simple_app_props_get(scf_handle_t *handle,
     const char *instance);

void scf_simple_app_props_free(scf_simple_app_props_t *propblock);

const scf_simple_prop_t *scf_simple_app_props_next
     (const scf_simple_app_props_t *propblock,scf_simple_prop_t *last);

const scf_simple_prop_t *scf_simple_app_props_search
     (const scf_simple_app_props_t *propblock, const char *pgname,
     const char *propname);

ssize_t scf_simple_prop_numvalues(const scf_simple_prop_t *prop);

scf_type_t scf_simple_prop_type(const scf_simple_prop_t *prop);

const char *scf_simple_prop_name(const scf_simple_prop_t *prop);

const char *scf_simple_prop_pgname(const scf_simple_prop_t *prop);

uint8_t *scf_simple_prop_next_boolean(const scf_simple_prop_t *prop);

uint64_t *scf_simple_prop_next_count(const scf_simple_prop_t *prop);

int64_t *scf_simple_prop_next_integer(const scf_simple_prop_t *prop);

int64_t *scf_simple_prop_next_time(const scf_simple_prop_t *prop,
     int32_t *nsec);

char *scf_simple_prop_next_astring(const scf_simple_prop_t *prop);

char *scf_simple_prop_next_ustring(const scf_simple_prop_t *prop);

void *scf_simple_prop_next_opaque(const scf_simple_prop_t *prop,
     size_t *length);

void *scf_simple_prop_next_reset(const scf_simple_prop_t *prop);
```

Description The simplified read interface to the Service Configuration Facility deals with properties and blocks of properties.

The `scf_simple_prop_get()` function pulls a single property. The `scf_simple_prop_*()` functions operate on the resulting `scf_simple_prop_t`.

The application might need to get many properties or iterate through all properties. The `scf_simple_app_props_get()` function gets all properties from the service instance that are in property groups of type 'application'. Individual properties are pulled from the block using the `scf_simple_app_props_next()` function for iteration or `scf_simple_app_props_search()` to search. The pointer to the `scf_simple_prop_t` returned from iteration or searching can be acted upon using the `scf_simple_prop_*()` functions. Each `scf_*_get()` function has an accompanying `scf_*_free` function. The application does not free the pointer to the `scf_simple_prop_t` returned from the `scf_simple_app_props_next()` and `scf_simple_app_props_search()` calls. A free call is only used with a corresponding get call.

The `scf_simple_prop_*()` functions return references to the read-only in-memory copy of the property information. Any changes to this information results in unstable behavior and inaccurate results. The simplified read interface provides read access only, with no provisions to modify data in the service configuration facility repository.

The `scf_simple_prop_get()` function takes as arguments a bound handle, a service instance FMRI, and the property group and property name of a property. If *handle* is `NULL`, the library uses a temporary handle created for the purpose. If *instance* is `NULL` the library automatically finds the FMRI of the calling process. If *pgname* is `NULL`, the library uses the default application property group. The caller is responsible for freeing the returned property with `scf_simple_prop_free()`.

The `scf_simple_prop_free()` function frees the `scf_simple_prop_t` allocated by `scf_simple_prop_get()`.

The `scf_simple_app_props_get()` function takes a bound handle and a service instance FMRI and pulls all the application properties into an `scf_simple_app_props_t`. If *handle* is `NULL`, the library uses a temporary handle created for the purpose. If *instance* is `NULL`, the library looks up the instance FMRI of the process calling the function. The caller is responsible for freeing the `scf_simple_app_props_t` with `scf_simple_app_props_free()`.

The `scf_simple_app_props_free()` function frees the `scf_simple_app_props_t` allocated by `scf_simple_app_props_get()`.

The `scf_simple_app_props_next()` function iterates over each property in an `scf_simple_app_props_t`. It takes an `scf_simple_app_props_t` pointer and the last property returned from the previous call and returns the next property in the `scf_simple_app_props_t`. Because the property is a reference into the `scf_simple_app_props_t`, its lifetime extends only until that structure is freed.

The`scf_simple_app_props_search()` function queries for an exact match on a property in a property group. It takes an apps prop object, a property group name, and a property name, and returns a property pointer. Because the property is a reference into the `scf_simple_app_props_t`, its lifetime extends only until that structure is freed. If the property group name, *pgname*, is `NULL`, "application" is used.

The `scf_simple_prop_numvalues()` function takes a pointer to a property and returns the number of values in that property.

The `scf_simple_prop_type()` function takes a pointer to a property and returns the type of the property in an `scf_type_t`.

The `scf_simple_prop_name()` function takes a pointer to a property and returns a pointer to the property name string.

The `scf_simple_prop_pgname()` function takes a pointer to a property and returns a pointer to the property group name string. The `scf_simple_prop_next_boolean()`, `scf_simple_prop_next_count()`, `scf_simple_prop_next_integer()`, `scf_simple_prop_next_astring()`, and `scf_simple_prop_next_ustring()` functions take a pointer to a property and return the first value in the property. Subsequent calls iterate over all the values in the property. The property's internal iteration can be reset with `scf_simple_prop_next_reset()`.

The `scf_simple_prop_next_time()` function takes a pointer to a property and the address of an allocated `int32_t` to hold the nanoseconds field, and returns the first value in the property. Subsequent calls iterate over the property values.

The `scf_simple_prop_next_opaque()` function takes a pointer to a property and the address of an allocated integer to hold the size of the opaque buffer. It returns the first value in the property. Subsequent calls iterate over the property values, as do the `scf_simple_prop_next_*()` functions. The `scf_simple_prop_next_opaque()` function writes the size of the opaque buffer into the allocated integer.

The `scf_simple_prop_next_reset()` function resets iteration on a property, so that a call to one of the `scf_simple_prop_next_*()` functions returns the first value in the property.

Return Values Upon successful completion, `scf_simple_prop_get()` returns a pointer to an allocated `scf_simple_prop_t`. Otherwise, it returns `NULL`.

Upon successful completion, `scf_simple_app_props_get()` returns a pointer to an allocated `scf_simple_app_props_t`. Otherwise, it returns `NULL`.

Upon successful completion, `scf_simple_app_props_next()` returns a pointer to an `scf_simple_prop_t`. Otherwise, it returns `NULL`.

Upon successful completion, `scf_simple_app_props_search()` returns a pointer to an `scf_simple_prop_t`. Otherwise, it returns `NULL`.

Upon successful completion, `scf_simple_prop_numvalues()` returns the number of values in a property. Otherwise, it returns -1.

Upon successful completion, `scf_simple_prop_type()` returns an `scf_type_t`. Otherwise, it returns -1.

Upon successful completion, `scf_simple_prop_name()` and `scf_simple_prop_pgname()` return character pointers. Otherwise, they return `NULL`.

Upon successful completion, `scf_simple_prop_next_boolean()`, `scf_simple_prop_next_count()`, `scf_simple_prop_next_integer()`, `scf_simple_prop_next_time()`, `scf_simple_prop_next_astring()`, `scf_simple_prop_next_ustring()`, and `scf_simple_prop_next_opaque()` return a pointer to the next value in the property. After all values have been returned, NULL is returned and `SCF_ERROR_NONE` is set. On failure, `NULL` is returned and the appropriate error value is set.

Errors The `scf_simple_prop_get()` and `scf_simple_app_props_get()` functions will fail if:

`SCF_ERROR_CONNECTION_BROKEN` The connection to the datastore is broken.

`SCF_ERROR_INVALID_ARGUMENT` The instance FMRI is invalid or property name is `NULL`.

`SCF_ERROR_NO_MEMORY` The memory allocation failed.

`SCF_ERROR_NOT_BOUND` The connection handle is not bound.

`SCF_ERROR_NOT_FOUND` The specified instance or property does not exist.

`SCF_ERROR_PERMISSION_DENIED` The caller is not authorized to read the property's value(s).

The `scf_simple_app_props_next()` function will fail if:

`SCF_ERROR_NOT_SET` The value of the *propblock* argument is `NULL`.

The `scf_simple_app_props_search()` function will fail if:

`SCF_ERROR_NOT_FOUND` The property was not found.

`SCF_ERROR_NOT_SET` The value of the *propblock* or *propname* argument is `NULL`.

The `scf_simple_prop_numvalues()`, `scf_simple_prop_type()`, `scf_simple_prop_name()`, and `scf_simple_prop_pgname()` functions will fail if:

`SCF_ERROR_NOT_SET` The property is `NULL`.

The `scf_simple_prop_next_boolean()`, `scf_simple_prop_next_count()`, `scf_simple_prop_next_integer()`, `scf_simple_prop_next_time()`, `scf_simple_prop_next_astring()`, `scf_simple_prop_next_ustring()`, and `scf_simple_prop_next_opaque()` functions will fail if:

`SCF_ERROR_NOT_SET` The property is `NULL`.

`SCF_ERROR_TYPE_MISMATCH` The requested type does not match the property type.

Examples

EXAMPLE 1 Simple Property Get

```
/*
 * In this example, we pull the property named "size" from the
 * default property group.  We make sure that the property
 * isn't empty, and then copy it into the sizeval variable.
 */

scf_simple_prop_t       *prop;
ssize_t                 numvals;
int64_t                 *sizeval;

prop = scf_simple_prop_get(
        "svc://localhost/category/service/instance",
        NULL, "size");

numvals = scf_simple_prop_numvalues(prop);

if(numvals > 0){
        sizeval = scf_simple_prop_next_integer(prop);
}

scf_simple_prop_free(prop);
```

EXAMPLE 2 Property Iteration

```
scf_simple_prop_t             *prop;
scf_simple_app_props_t        *appprops;

appprops = scf_simple_app_props_get(
        "svc://localhost/category/service/instance");

prop = scf_simple_app_props_next(appprops, NULL);

while(prop != NULL)
{
        /*
         * This iteration will go through every property in the
         * instance's application block.  The user can use
         * the set of property functions to pull the values out
         * of prop, as seen in other examples.
         */

        (...code acting on each property...)
```

EXAMPLE 2 Property Iteration *(Continued)*

```
prop = scf_simple_app_props_next(appprops, prop);

}

scf_simple_app_props_free(appprops);
```

EXAMPLE 3 Property Searching

```
/*
 * In this example, we pull the property block from the instance,
 * and then query it.  Generally speaking, the simple get would
 * be used for an example like this, but for the purposes of
 * illustration, the non-simple approach is used.  The property
 * is a list of integers that are pulled into an array.
 * Note how val is passed back into each call, as described above.
 */

scf_simple_app_props_t         *appprops;
scf_simple_prop_t              *prop;
int                     i;
int64_t                 *intlist;
ssize_t                 numvals;

appprops = scf_simple_app_props_get(
              "svc://localhost/category/service/instance");

prop = scf_simple_app_props_search(appprops, "appname", "numlist");

if(prop != NULL){

        numvals = scf_simple_prop_numvalues(prop);

        if(numvals > 0){

        intlist = malloc(numvals * sizeof(int64_t));

        val = scf_simple_prop_next_integer(prop);

                for(i=0, i < numvals, i++){
                        intlist[i] = *val;
                        val = scf_simple_prop_next_integer(prop);
                }
        }
}

scf_simple_app_props_free(appprops);
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also libscf(3LIB), scf_error(3SCF), attributes(5)

Name scf_simple_walk_instances – observational interface for Service Configuration Facility

Synopsis

```
cc [ flag... ] file... -lscf [ library... ]
#include <libscf.h>

int scf_simple_walk_instances(uint_t flags, void *private,
     int (*inst_callback)(scf_handle_t *, scf_instance_t *, void *));
```

Description The `scf_simple_walk_instances()` function iterates over every service instance in a specified state and calls a callback function provided by the user on each specified instance.

The function takes a *flags* argument to indicate which instance states are involved in the iteration, an opaque buffer to be passed to the callback function, and a callback function with three arguments, a handle, an instance pointer, and an opaque buffer. If the callback function returns a value other than success, iteration is ended, an error is set, and the function returns -1.

The handle passed to the callback function is provided to the callback function by the library. This handle is used by the callback function for all low-level allocation involved in the function.

The simplified library provides defined constants for the *flags* argument. The user can use a bitwise OR to apply more than one flag. The `SCF_STATE_ALL` flag is a bitwise OR of all the other states.The flags are:

- `SCF_STATE_UNINIT`
- `SCF_STATE_MAINT`
- `SCF_STATE_OFFLINE`
- `SCF_STATE_DISABLED`
- `SCF_STATE_ONLINE`
- `SCF_STATE_DEGRADED`
- `SCF_STATE_ALL`

Return Values Upon successful completion, `scf_simple_walk_instances()` returns 0. Otherwise, it returns -1.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also `libscf(3LIB)`, `attributes(5)`

Name scf_snaplevel_create, scf_snaplevel_handle, scf_snaplevel_destroy, scf_snaplevel_get_parent, scf_snaplevel_get_scope_name, scf_snaplevel_get_service_name, scf_snaplevel_get_instance_name, scf_snapshot_get_base_snaplevel, scf_snaplevel_get_next_snaplevel – create and manipulate snaplevel handles in the Service Configuration Facility

Synopsis

```
cc [ flag... ] file... -lscf [ library... ]
#include <libscf.h>

scf_snaplevel_t *scf_snaplevel_create(scf_handle_t *handle);

scf_handle_t *scf_snaplevel_handle(scf_snaplevel_t *level);

void scf_snaplevel_destroy(scf_snaplevel_t *level);

int scf_snaplevel_get_parent(const scf_snaplevel_t *level,
     const scf_snapshot_t *snap);

ssize_t scf_snaplevel_get_scope_name(const scf_snaplevel_t *level,
     char *buf, size_t size);

ssize_t scf_snaplevel_get_service_name(const scf_snaplevel_t *level,
     char *buf, size_t size);

ssize_t scf_snaplevel_get_instance_name(const scf_snaplevel_t *level,
     char *buf, size_t size);

int scf_snapshot_get_base_snaplevel(const scf_snapshot_t *snap,
     scf_snaplevel_t *level);

int scf_snaplevel_get_next_snaplevel(scf_snaplevel_t *in,
     scf_snaplevel_t *out);
```

Description A snaplevel holds all of the property groups associated with either a service or an instance. Each snapshot has an ordered list of snaplevels. Snaplevels contain the names of the instance or service from which they are derived.

An `scf_snaplevel_t` is an opaque handle that can be set to a single snaplevel at any given time. When set, the `scf_snaplevel_t` inherits the point in time from the `scf_snapshot_t` from which it comes.

The `scf_snaplevel_create()` function allocates and initializes a new `scf_snaplevel_t` bound to *handle*. The `scf_snaplevel_destroy()` function destroys and frees *level*.

The `scf_snaplevel_handle()` function retrieves the handle to which *level* is bound.

The `scf_snaplevel_get_parent()` function sets *snap* to the parent snapshot of the snaplevel to which *level* is set. The snapshot specified by *snap* is attached to the same point in time as level.

The `scf_snaplevel_get_scope_name()`, `scf_snaplevel_get_service_name()`, and `scf_snaplevel_get_instance_name()` functions retrieve the name of the scope, service, and

instance for the snapshot to which *snap* is set. If the snaplevel is from an instance, all three succeed. If the snaplevel is from a service, `scf_snaplevel_get_instance_name()` fails.

The `scf_snapshot_get_base_snaplevel()` function sets *level* to the first snaplevel in the snapshot to which *snap* is set. The `scf_snaplevel_get_next_snaplevel()` function sets *out* to the next snaplevel after the snaplevel to which *in* is set. Both the *in* and *out* arguments can point to the same `scf_snaplevel_t`.

To retrieve the property groups associated with a snaplevel, see `scf_iter_snaplevel_pgs(3SCF)`, `scf_iter_snaplevel_pgs_typed(3SCF)`, and `scf_snaplevel_get_pg(3SCF)`.

Return Values Upon successful completion, `scf_snaplevel_create()` returns a new `scf_snaplevel_t`. Otherwise, it returns `NULL`.

Upon successful completion, `scf_snaplevel_get_scope_name()`, `scf_snaplevel_get_service_name()`, and `scf_snaplevel_get_instance_name()` return the length of the string written, not including the terminating null byte. Otherwise, they return -1.

Upon successful completion, `scf_snaplevel_get_parent()`, `scf_snapshot_get_base_snaplevel()`, and `scf_snaplevel_get_next_snaplevel()` return. Otherwise, they return -1.

Errors The `scf_snaplevel_create()` function will fail if:

`SCF_ERROR_INVALID_ARGUMENT`	The *handle* argument is `NULL`.
`SCF_ERROR_NO_MEMORY`	There is not enough memory to allocate an `scf_snaplevel_t`.
`SCF_ERROR_NO_RESOURCES`	The server does not have adequate resources for a new snapshot handle.

The `scf_snaplevel_get_scope_name()`, `scf_snaplevel_get_service_name()`, `scf_snaplevel_get_instance_name()`, and `scf_snaplevel_get_parent()` functions will fail if:

`SCF_ERROR_DELETED`	The object referred to by *level* has been deleted.
`SCF_ERROR_NOT_SET`	The snaplevel is not set.
`SCF_ERROR_NOT_BOUND`	The handle is not bound.
`SCF_ERROR_CONNECTION_BROKEN`	The connection to the repository was lost.

The `scf_snaplevel_get_instance_name()` function will fail if:

`SCF_ERROR_CONSTRAINT_VIOLATED`	The snaplevel is derived from a service.

The `scf_snapshot_get_base_snaplevel()` function will fail if:

SCF_ERROR_CONNECTION_BROKEN	The connection to the repository was lost.
SCF_ERROR_DELETED	The snapshot has been deleted.
SCF_ERROR_HANDLE_MISMATCH	The snapshot and snaplevel are not derived from the same handle.
SCF_ERROR_NO_RESOURCES	The server does not have the resources to complete the request.
SCF_ERROR_NOT_BOUND	The handle is not bound.
SCF_ERROR_NOT_FOUND	There are no snaplevels in this snapshot.
SCF_ERROR_NOT_SET	The snapshot is not set.

The `scf_snaplevel_get_next_snaplevel()` function will fail if:

SCF_ERROR_DELETED	The snaplevel has been deleted.
SCF_ERROR_NOT_SET	The snaplevel is not set.
SCF_ERROR_HANDLE_MISMATCH	The *in* and *out* arguments are not derived from the same handle.
SCF_ERROR_NOT_BOUND	The handle is not bound.
SCF_ERROR_CONNECTION_BROKEN	The connection to the repository was lost.
SCF_ERROR_NOT_FOUND	There are no more snaplevels in this snapshot.

The `scf_error(3SCF)` function can be used to retrieve the error value.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also

`libscf(3LIB)`, `scf_error(3SCF)`, `scf_iter_snaplevel_pgs(3SCF)`, `scf_iter_snaplevel_pgs_typed(3SCF)`, `scf_snaplevel_get_pg(3SCF)`, `attributes(5)`

Name scf_snapshot_create, scf_snapshot_handle, scf_snapshot_destroy, scf_snapshot_get_parent, scf_snapshot_get_name, scf_snapshot_update, scf_instance_get_snapshot – create and manipulate snapshot handles and snapshots in the Service Configuration Facility

Synopsis

```
cc [ flag... ] file... -lscf [ library... ]
#include <libscf.h>

scf_snapshot_t *scf_snapshot_create(scf_handle_t *handle);

scf_handle_t *scf_snapshot_handle(scf_snapshot_t *snap);

void scf_snapshot_destroy(scf_snapshot_t *snap);

int scf_snapshot_get_parent(const scf_snapshot_t *snap,
     scf_instance_t *inst);

ssize_t scf_snapshot_get_name(const scf_snapshot_t *snap,
     char *buf, size_t size);

int scf_snapshot_update(scf_snapshot_t *snap);

int scf_instance_get_snapshot(const scf_instance_t *inst,
     const char *name, scf_snapshot_t *snap);
```

Description A snapshot is an unchanging picture of the full set of property groups associated with an instance. Snapshots are automatically created and managed by the Solaris Management Facility. See `smf(5)`.

A snapshot consists of a set of snaplevels, each of which holds copies of the property groups associated with an instance or service in the resolution path of the base instance. Typically, there is one snaplevel for the instance and one for the instance's parent service.

The `scf_snapshot_create()` function allocates and initializes a new `scf_snapshot_t` bound to *handle*. The `scf_snapshot_destroy()` function destroys and frees *snap*.

The `scf_snapshot_handle()` function retrieves the handle to which *snap* is bound.

The `scf_snapshot_get_parent()` function sets *inst* to the parent of the snapshot to which *snap* is set.

The `scf_snapshot_get_name()` function retrieves the name of the snapshot to which *snap* is set.

The `scf_snapshot_update()` function reattaches *snap* to the latest version of the snapshot to which *snap* is set.

The `scf_instance_get_snapshot()` function sets *snap* to the snapshot specified by *name* in the instance specified by *inst*. To walk all of the snapshots, see `scf_iter_instance_snapshots(3SCF)`.

To access the snaplevels of a snapshot, see `scf_snapshot_get_base_snaplevel(3SCF)`.

Return Values Upon successful completion, `scf_snapshot_create()` returns a new `scf_snapshot_t`. Otherwise, it returns `NULL`.

Upon successful completion, `scf_snapshot_handle()` returns the handle to which *snap* is bound. Otherwise, it returns `NULL`.

Upon successful completion, `scf_snapshot_get_name()` returns the length of the string written, not including the terminating null byte. Otherwise, it returns `NULL`.

The `scf_snapshot_update()` function returns 1 if the snapshot was updated, 0 if the snapshot had not been updated, and -1 on failure.

Upon successful completion, `scf_snapshot_get_parent()` and `scf_instance_get_snapshot()` return 0. Otherwise, they return -1.

Errors The `scf_snapshot_create()` function will fail if:

`SCF_ERROR_INVALID_ARGUMENT`
The *handle* argument is `NULL`.

`SCF_ERROR_NO_MEMORY`
There is not enough memory to allocate an `scf_snapshot_t`.

`SCF_ERROR_NO_RESOURCES`
The server does not have adequate resources for a new instance handle.

The `scf_snapshot_handle()` function will fail if:

`SCF_ERROR_HANDLE_DESTROYED`
The handle associated with *snap* has been destroyed.

The `scf_snapshot_get_name()` and `scf_snapshot_get_parent()` functions will fail if:

`SCF_ERROR_DELETED`
The snapshot has been deleted.

`SCF_ERROR_NOT_SET`
The snapshot is not set.

`SCF_ERROR_NOT_BOUND`
The handle is not bound.

`SCF_ERROR_CONNECTION_BROKEN`
The connection to the repository was lost.

The `scf_snapshot_update()` function will fail if:

`SCF_ERROR_CONNECTION_BROKEN`
The connection to the repository was lost.

`SCF_ERROR_DELETED`
An ancestor of the snapshot specified by *snap* has been deleted.

`SCF_ERROR_INTERNAL`
An internal error occurred. This can happen if *snap* has been corrupted.

`SCF_ERROR_INVALID_ARGUMENT`
The *snap* argument refers to an invalid `scf_snapshot_t`.

`SCF_ERROR_NOT_BOUND`
The handle is not bound.

`SCF_ERROR_NOT_SET`
The snapshot specified by *snap* is not set.

The `scf_instance_get_snapshot()` function will fail if:

`SCF_ERROR_BACKEND_ACCESS`
The storage mechanism that the repository server (`svc.configd(1M)`) chose for the operation denied access.

`SCF_ERROR_CONNECTION_BROKEN`
The connection to the repository was lost.

`SCF_ERROR_DELETED`
The instance has been deleted.

`SCF_ERROR_HANDLE_MISMATCH`
The instance and snapshot are not derived from the same handle.

`SCF_ERROR_INTERNAL`
An internal error occurred.

`SCF_ERROR_INVALID_ARGUMENT`
The value of the *name* argument is not a valid snapshot name.

`SCF_ERROR_NO_RESOURCES`
The server does not have the resources to complete the request.

`SCF_ERROR_NOT_BOUND`
The handle is not bound.

`SCF_ERROR_NOT_FOUND`
The snapshot specified by *name* was not found.

`SCF_ERROR_NOT_SET`
The instance is not set.

The `scf_error(3SCF)` function can be used to retrieve the error value.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also libscf(3LIB), scf_error(3SCF), scf_iter_instance_snapshots(3SCF), scf_snapshot_get_base_snaplevel(3SCF), attributes(5), smf(5)

Name scf_tmpl_pg_create, scf_tmpl_pg_reset, scf_tmpl_pg_destroy, scf_tmpl_get_by_pg_name, scf_tmpl_get_by_pg, scf_tmpl_iter_pgs – template property group functions

Synopsis

```
cc [ flag... ] file... -lscf [ library... ]
#include <libscf.h>

scf_pg_tmpl_t *scf_tmpl_pg_create(scf_handle_t *handle);

void scf_tmpl_pg_reset(scf_pg_tmpl_t *pg_tmpl);

void scf_tmpl_pg_destroy(scf_pg_tmpl_t *pg_tmpl);

int scf_tmpl_get_by_pg_name(const char *instance_fmri,
     const char *snapshot, const char *pg_name,
     const char *pg_type, scf_pg_tmpl_t *pg_tmpl, int flags);

int scf_tmpl_get_by_pg(scf_propertygroup_t *pg,
     scf_pg_tmpl_t *pg_tmpl, int flags)

int scf_tmpl_iter_pgs(scf_pg_tmpl_t *pg_tmpl, const char *fmri,
     const char *snapshot, const char *pg_type, int flags);
```

Description The template property group functions locate and give access to metadata about SMF configuration for service instances. They are used to directly access property group metadata and explore metadata for properties contained in those property groups.

A property group does not need to be currently defined in order to explore metadata about it, as long as the metadata is defined. Thus, the property group template functions operate on strings rather than `scf_propertygroup_t` entities.

By default, when an instance FMRI is specified, `scf_tmpl_get_by_pg_name()` and `scf_tmpl_iter_pgs()` lookup composed data from the running snapshot of the instance. A different snapshot may be explicitly selected by specifying a valid snapshot name rather than NULL for the snapshot argument. If a service FMRI is specified, the current properties are used.

By default, these functions also explore template data defined by the service or instance itself, the service's restarter, and global template data. See `smf_template(5)` for more information about this composition.

Once retrieved, the `scf_pg_tmpl_t` can be explored using the `scf_tmpl_pg_name(3SCF)` and `scf_tmpl_prop_create(3SCF)` suite of functions.

Before calling `scf_tmpl_get_by_pg()`, `scf_tmpl_get_by_pg_name()`, or `scf_tmpl_iter_pgs()`, the `scf_pg_tmpl_t` must be allocated by `scf_tmpl_pg_create()`. The `scf_pg_tmpl_t` can be reset to contain no template information with `scf_tmpl_pg_reset()`, so that it can be used to start an iteration from scratch. All associated memory can be freed with `scf_tmpl_pg_destroy()`.

The `scf_tmpl_get_by_pg()` function locates the most specific matching template for the property group supplied. The parent of that property group can be either a service or an instance.

The `scf_tmpl_get_by_pg_name()` function locates the most specific matching template for the property group as specified. As described above, when the snapshot argument is `NULL` the default running snapshot is used. If flags includes `SCF_PG_TMPL_FLAG_CURRENT`, the snapshot argument is ignored and the current configuration is used. If flags includes `SCF_PG_TMPL_FLAG_EXACT`, only the exact FMRI is looked up. Either or both of the *pg_name* and *pg_type* arguments may be specified as `NULL`. In this case, *pg_name* and/or *pg_type* is wildcarded and matches any value. The most specific snapshot matching those arguments is returned.

The `scf_tmpl_iter_pgs()` function iterates across all templates defined for the specified FMRI, snapshot, and optional property group type. It also takes an optional flags argument. If flags includes `SCF_PG_TMPL_FLAG_CURRENT`, the snapshot argument is ignored and the "running" snapshot is used. `SCF_PG_TMPL_FLAG_REQUIRED` searches only for required property groups. `SCF_PG_TMPL_FLAG_EXACT` looks only at the exact FMRI provided for templates, and not for templates defined on its restarter or globally.

The iterator state for `scf_tmpl_iter_pgs()` is stored on the template data structure. The data structure should be allocated with `scf_tmpl_pg_create()` and to continue the iteration the previously returned structure should be passed in as an argument.

Return Values The `scf_tmpl_pg_create()` function returns `NULL` on failure and a pointer to an allocated and populated `scf_pg_tmpl_t` on success. The caller is responsible for freeing the memory with `scf_tmpl_pg_destroy()`.

The `scf_tmpl_get_by_pg()` and `scf_tmpl_get_by_pg_name()` functions return 0 on success and -1 on failure.

The `scf_tmpl_iter_pgs()` function returns 1 on successful completion. If the iteration is complete, it returns 0. It returns -1 on error.

Errors The `scf_tmpl_get_by_pg()`, `scf_tmpl_get_by_pg_name()`, and `scf_tmpl_iter_pgs()` functions will fail if:

`SCF_ERROR_BACKEND_ACCESS`	The storage mechanism that the repository server (`svc.configd(1M)`) chose for the operation denied access.
`SCF_ERROR_CONNECTION_BROKEN`	The connection to the repository was lost.
`SCF_ERROR_DELETED`	The instance or its template property group has been deleted.
`SCF_ERROR_HANDLE_DESTROYED`	The handle passed in has been destroyed.
`SCF_ERROR_INTERNAL`	An internal error occurred.

`SCF_ERROR_INVALID_ARGUMENT`	The *handle* argument, *fmri* argument, snapshot name, *pg_name*, or *pg* is invalid.
`SCF_ERROR_NO_MEMORY`	There is not enough memory to populate the `scf_pg_tmpl_t`.
`SCF_ERROR_NO_RESOURCES`	The server does not have adequate resources to complete the request.
`SCF_ERROR_NOT_BOUND`	The handle is not currently bound.
`SCF_ERROR_NOT_FOUND`	The object matching FMRI does not exist in the repository, or the snapshot does not exist.
`SCF_ERROR_PERMISSION_DENIED`	The template could not be read due to access restrictions.

The `scf_tmpl_get_by_pg()` function will fail if:

`SCF_ERROR_NOT_SET`	The property group specified by *pg* is not set.

The `scf_tmpl_pg_create()` function will fail if:

`SCF_ERROR_INVALID_ARGUMENT`	The handle argument is `NULL`.
`SCF_ERROR_NO_MEMORY`	There is no memory available.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also

`svc.configd(1M)`, `scf_tmpl_pg_name(3SCF)`, `scf_tmpl_prop_create(3SCF)`, `attributes(5)`, `smf_template(5)`

Name scf_tmpl_pg_name, scf_tmpl_pg_type, scf_tmpl_pg_target, scf_tmpl_pg_required, scf_tmpl_pg_common_name, scf_tmpl_pg_description – retrieve the metadata about a specific property group

Synopsis

```
cc [ flag... ] file... -lscf [ library... ]
#include <libscf.h>

ssize_t scf_tmpl_pg_name(const scf_pg_tmpl_t *pg_tmpl,
     char **out);

ssize_t scf_tmpl_pg_type(const scf_pg_tmpl_t *pg_tmpl,
     char **out);

ssize_t scf_tmpl_pg_target(const scf_pg_tmpl_t *pg_tmpl,
     char **out);

int scf_tmpl_pg_required(const scf_pg_tmpl_t *pg_tmpl,
     uint8_t *out)

ssize_t scf_tmpl_pg_common_name(const scf_pg_tmpl_t *pg_tmpl,
     char * locale, char **out);

ssize_t scf_tmpl_pg_description(const scf_pg_tmpl_t *pg_tmpl,
     char * locale, char **out);
```

Description These functions retrieve the metadata about a specific property group. They require that the template for the property group has already been located by one of the `scf_tmpl_pg_create(3SCF)` suite of functions.

The `scf_tmpl_pg_name()` function retrieves the name of the property group template and place it in **out*. If the property group name is implicitly wildcarded (see `smf_template(5)`) in the template, this function will return a string containing `SCF_TMPL_WILDCARD` (“*”) in **out*. The caller is responsible for freeing the **out* buffer on success.

The `scf_tmpl_pg_type()` function will retrieve the type of the property group template and place it in **out*. If the property group type is implicitly wildcarded (see `smf_template(5)`) in the template, this function will return a string containing `SCF_TMPL_WILDCARD` (“*”) in **out*. The caller is responsible for freeing the **out* buffer on success.

The `scf_tmpl_pg_target()` function will retrieve the target of the property group template and place it in **out*. The caller is responsible for freeing the **out* buffer on success.

The `scf_tmpl_pg_required()` function will determine whether the property group is required and place the result of that check in **out*. If required is unset, *out* will be the default value of 0. If the property is explicitly set to required, *out* will be 1.

The `scf_tmpl_pg_common_name()` function will retrieve the property group's localized common name as currently templated and place it in **out*. A locale (as described in `setlocale(3C)`) may be specified, or if the supplied locale is NULL, the current locale will be used. If a common_name in the specified locale is not found, the function will also look for a

common_name in the C locale. Some templates will not specify the property group common name. The caller is responsible for freeing the **out* buffer on success.

The `scf_tmpl_pg_description()` function will retrieve the property group's localized description as currently templated and place it in **out*. A locale (as described in `setlocale(3C)`) may be specified, or if the supplied locale is NULL, the current locale will be used. If a description in the specified locale is not found, the function will also look for a description in the C locale. Some templates will not specify the property group description. The caller is responsible for freeing the **out* buffer on success.

Return Values Upon successful completion, `scf_tmpl_pg_name()`, `scf_tmpl_pg_common_name()`, `scf_tmpl_pg_description()`, `scf_tmpl_pg_target()`, and `scf_tmpl_pg_type()` return the length of the string written, not including the terminating null byte. Otherwise, they return -1.

Upon successful completion, `scf_tmpl_pg_required()` returns 0. Otherwise, it returns -1.

Errors The `scf_tmpl_pg_name()`, `scf_tmpl_pg_common_name()`, `scf_tmpl_pg_description()`, `scf_tmpl_pg_required()`, `scf_tmpl_pg_target()`, and `scf_tmpl_pg_type()` functions will fail if:

`SCF_ERROR_BACKEND_ACCESS`	The storage mechanism that the repository server (`svc.configd(1M)`) chose for the operation denied access.
`SCF_ERROR_CONNECTION_BROKEN`	The connection to the repository was lost.
`SCF_ERROR_DELETED`	The template property group has been deleted.
`SCF_ERROR_HANDLE_DESTROYED`	The handle passed in has been destroyed.
`SCF_ERROR_INTERNAL`	An internal error occurred.
`SCF_ERROR_NO_MEMORY`	There is not enough memory to populate the `scf_pg_tmpl_t`.
`SCF_ERROR_NO_RESOURCES`	The server does not have adequate resources to complete the request.
`SCF_ERROR_NOT_BOUND`	The handle is not currently bound.
`SCF_ERROR_PERMISSION_DENIED`	The template could not be read due to access restrictions.
`SCF_ERROR_TEMPLATE_INVALID`	The template data is invalid.

The `scf_tmpl_pg_common_name()` and `scf_tmpl_pg_description()` functions will fail if:

`SCF_ERROR_NOT_FOUND`	The property does not exist or exists and has no value.
`SCF_ERROR_INVALID_ARGUMENT`	The locale string is too long.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also svc.configd(1M), scf_tmpl_pg_create(3SCF), setlocale(3C), attributes(5), smf_template(5)

Name scf_tmpl_prop_create, scf_tmpl_prop_reset, scf_tmpl_prop_destroy, scf_tmpl_get_by_prop, scf_tmpl_iter_props – template property functions

Synopsis

```
cc [ flag... ] file... -lscf [ library... ]
#include <libscf.h>

scf_prop_tmpl_t *scf_tmpl_prop_create(scf_handle_t *handle);

void scf_tmpl_prop_reset(scf_prop_tmpl_t *prop_tmpl);

void scf_tmpl_prop_destroy(scf_prop_tmpl_t *prop_tmpl);

int scf_tmpl_get_by_prop(scf_pg_tmpl_t *pg_tmpl,
     const char *prop, scf_prop_tmpl_t *prop_tmpl, int flags)

int scf_tmpl_iter_props(scf_pg_tmpl_t *pg_tmpl, const char *fmri,
     scf_prop_tmpl_t *prop_tmpl, int flags);
```

Description The template property functions locate and give access to metadata about properties. They require that the template for the property group containing the property has already been located by one of the `scf_tmpl_pg_create(3SCF)` suite of functions.

Once retrieved, the `scf_prop_tmpl_t` can be explored using the `scf_tmpl_prop_name(3SCF)` suite of functions.

Before calling `scf_tmpl_get_by_prop()` or `scf_tmpl_iter_props()`, the `scf_prop_tmpl_t` must be allocated by `scf_tmpl_prop_create`. The `scf_prop_tmpl_t` can be reset to contain no template information with `scf_tmpl_prop_reset()`, so that it can be used to start an iteration from scratch. All associated memory can be freed with `scf_tmpl_prop_destroy()`.

The `scf_tmpl_get_by_prop()` function locates template data about the property name specified.

The `scf_tmpl_iter_props()` function iterates across all property templates defined in the specified property group template.

The iterator state for `scf_tmpl_iter_props()` is stored on the property template data structure. The data structure should be allocated with `scf_tmpl_prop_create()`, and to continue the iteration the previously returned structure should be passed in as an argument.

Return Values The `scf_tmpl_get_by_prop()` function returns -1 on failure and 0 on success.

The `scf_tmpl_iter_props()` function returns 1 on successful completion. If the iteration is complete, it returns 0. It returns -1 on error.

Errors The `scf_tmpl_get_by_prop()` and `scf_tmpl_iter_props()` functions will fail if:

`SCF_ERROR_BACKEND_ACCESS`	The storage mechanism that the repository server (`svc.configd(1M)`) chose for the operation denied access.
`SCF_ERROR_CONNECTION_BROKEN`	The connection to the repository was lost.

SCF_ERROR_DELETED	The instance or its template property group has been deleted.
SCF_ERROR_HANDLE_DESTROYED	The handle passed in has been destroyed.
SCF_ERROR_INTERNAL	An internal error occurred.
SCF_ERROR_INVALID_ARGUMENT	One of the arguments is invalid.
SCF_ERROR_NO_MEMORY	There is not enough memory to populate the scf_prop_tmpl_t.
SCF_ERROR_NO_RESOURCES	The server does not have adequate resources to complete the request.
SCF_ERROR_NOT_BOUND	The handle is not currently bound.

The scf_tmpl_get_by_prop() function will fail if:

SCF_ERROR_NOT_FOUND	Template object matching property doesn't exist in the repository.
SCF_ERROR_TYPE_MISMATCH	Matching template object is the wrong type in the repository.
SCF_ERROR_PERMISSION_DENIED	The template could not be read due to access restrictions.
SCF_ERROR_TEMPLATE_INVALID	The template data is invalid.

The scf_tmpl_prop_create() function will fail if:

SCF_ERROR_INVALID_ARGUMENT	The *handle* argument is NULL.
SCF_ERROR_NO_MEMORY	There is no memory available.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also

svc.configd(1M), scf_tmpl_pg_create(3SCF), scf_tmpl_prop_name(3SCF), attributes(5), smf_template(5)

Name scf_tmpl_prop_name, scf_tmpl_prop_type, scf_tmpl_prop_required, scf_tmpl_prop_common_name, scf_tmpl_prop_description, scf_tmpl_prop_units, scf_tmpl_prop_visibility, scf_tmpl_visibility_to_string, scf_tmpl_prop_cardinality, scf_tmpl_prop_internal_seps, scf_tmpl_value_name_constraints, scf_count_ranges_destroy, scf_int_ranges_destroy, scf_tmpl_value_count_range_constraints, scf_tmpl_value_int_range_constraints, scf_tmpl_value_name_choices, scf_values_destroy, scf_tmpl_value_count_range_choices, scf_tmpl_value_int_range_choices, scf_tmpl_value_common_name, scf_tmpl_value_description, scf_tmpl_value_in_constraint – retrieve the metadata about a specific property

Synopsis

```
cc [ flag... ] file... -lscf [ library... ]
#include <libscf.h>

ssize_t scf_tmpl_prop_name(const scf_prop_tmpl_t *prop_tmpl,
     char **out);

int scf_tmpl_prop_type(const scf_prop_tmpl_t *prop_tmpl,
     scf_type_t *out);

int scf_tmpl_prop_required(const scf_prop_tmpl_t *prop_tmpl,
     uint8_t *out)

ssize_t scf_tmpl_prop_common_name(const scf_prop_tmpl_t *prop_tmpl,
     char *locale, char **out);

ssize_t scf_tmpl_prop_description(const scf_prop_tmpl_t *prop_tmpl,
     char * locale, char **out);

ssize_t scf_tmpl_prop_units(const scf_prop_tmpl_t *prop_tmpl,
     const char *locale, char **out);

int scf_tmpl_prop_visibility(const scf_prop_tmpl_t *prop_tmpl,
     uint8_t *out);

const char *scf_tmpl_visibility_to_string(uint8_t visibility);

int scf_tmpl_prop_cardinality(const scf_prop_tmpl_t *prop_tmpl,
     uint64_t *min, uint64_t *max);

int scf_tmpl_prop_internal_seps(const scf_prop_tmpl_t *prop_tmpl,
     scf_values_t *out);

int scf_tmpl_value_name_constraints(const scf_prop_tmpl_t *prop_tmpl,
     scf_values_t *out);

void scf_count_ranges_destroy(scf_count_ranges_t *ranges);

void scf_int_ranges_destroy(scf_int_ranges_t *ranges);

int scf_tmpl_value_count_range_constraints(
     const scf_prop_tmpl_t *prop_tmpl, scf_count_ranges_t *ranges);

int scf_tmpl_value_int_range_constraints(
     const scf_prop_tmpl_t *prop_tmpl, scf_int_ranges_t *ranges);
```

```
int scf_tmpl_value_name_choices(const scf_prop_tmpl_t *prop_tmpl,
     scf_values_t *vals);

void scf_values_destroy(scf_values_t *vals);

int scf_tmpl_value_count_range_choices(
     const scf_prop_tmpl_t *prop_tmpl, scf_count_ranges_t *ranges);

int scf_tmpl_value_int_range_choices(const scf_prop_tmpl_t *prop_tmpl,
     scf_int_ranges_t *ranges);

ssize_t scf_tmpl_value_common_name(const scf_prop_tmpl_t *prop_tmpl,
     const char *locale, const char *value, char **out);

ssize_t scf_tmpl_value_description(const scf_prop_tmpl_t *prop_tmpl,
     const char *locale, const char *value, char **out);

int scf_tmpl_value_in_constraint(const scf_prop_tmpl_t *prop_tmpl,
     scf_value_t *value, scf_tmpl_errors_t **errs);
```

Description These functions retrieve the metadata about a specific property. They require that the template for the property has already been located by one of the `scf_tmpl_prop_create(3SCF)` suite of functions.

The `scf_tmpl_prop_name()` function will retrieve the property's name as currently templated and place it in **out*. The caller is responsible for freeing the **out* buffer on success.

The `scf_tmpl_prop_type()` function will retrieve the type of the property as templated and place the type in out.

The `scf_tmpl_prop_required()` function will determine whether the property is required in this property group and place the result of that check in out. If required is unset, out will be the default, 0. If the property is explicitly set to required, out will be 1.

The `scf_tmpl_prop_common_name()` function will retrieve the property's localized common name as currently templated and place it in **out*. A locale (as described in `setlocale(3C)`) can be specified, or if the supplied locale is `NULL`, the current locale will be used. If a common name in the specified locale is not found, the function will also look for a common name in the C locale. Some templates will not specify the property common name. The caller is responsible for freeing the **out* buffer on success.

The `scf_tmpl_prop_description()` function will retrieve the property's localized description as currently templated and place it in **out*. A locale (as described in `setlocale(3C)`) can be specified, or if the supplied locale is `NULL`, the current locale will be used. If a description in the specified locale is not found, the function will also look for a description in the C locale. Some templates will not specify the property description. The caller is responsible for freeing the **out* buffer on success.

The `scf_tmpl_prop_visibility()` function will retrieve the visibility of the property as currently templated and place it in out. A property can be `SCF_TMPL_VISIBILITY_HIDDEN`,

SCF_TMPL_VISIBILITY_READONLY, or SCF_TMPL_VISIBILITY_READWRITE. If the visibility is unset, this function will return the default, SCF_TMPL_VISIBILITY_READWRITE.

The scf_tmpl_prop_cardinality() function will retrieve the minimum number of values and maximum number of values allowed for this property and place them in *min* and *max*, respectively. If the values are unset, the defaults of 0 for *min* and UINT64_MAX for *max*.

The scf_values_destroy() function destroys an scf_values_t structure and all memory associated with it.

The scf_values_t structure is populated by a number of functions. Based on the value type, it is populated with an array of the values. It is also always populated with an array of *astring* translations of those values.

```
typedef struct scf_time {
      int64_t         t_seconds;
      int32_t         t_ns;
} scf_time_t;

typedef struct scf_values {
      scf_type_t              value_type;
      char                    *reserved;
      int                     value_count;
      const char              **values_as_astring;
      union {
              uint64_t        *v_count;
              uint8_t         *v_boolean;
              int64_t         *v_integer;
              char            **v_astring;
              char            **v_ustring;
              char            **v_opaque;
              scf_time_t      *v_time;
      } sv_data;
} scf_values_t;
```

The scf_tmpl_prop_internal_seps() function will retrieve the list of internal separators as currently defined in the template. Each separator will be a single string character in a different element of out. Some templates will not specify any internal separators. The caller is responsible for calling scf_values_destroy() on success.

The scf_tmpl_value_name_constraints() function will retrieve the set of property values the property is expected to be part of. Some templates will not specify any constraints. The caller is responsible for calling scf_values_destroy() on success.

The scf_tmpl_value_count_range_constraints() function will retrieve the set of defined lower and upper bounds as defined by the property template and place them in *ranges*. Some templates will not specify any range constraints.

The `scf_count_ranges_t` structure is populated by the `scf_tmpl_value_count_range_constraints()` and `scf_tmpl_value_count_range_choices()` functions. `scf_count_ranges_destroy()` destroys an `scf_count_ranges_t` and all memory associated with it.

```
typedef struct scf_count_ranges {
        int             scr_num_ranges;
        uint64_t        *scr_min;
        uint64_t        *scr_max;
} scf_count_ranges_t;
```

The `scf_tmpl_value_int_range_constraints()` function will retrieve the set of defined lower and upper bounds as defined by the property template and place them in *ranges*. Some templates will not specify any range constraints.

The `scf_int_ranges_t` structure is populated by the `scf_tmpl_value_int_range_constraints()` and `scf_tmpl_value_int_range_choices()` functions. The `scf_int_ranges_destroy()` function destroys an `scf_int_ranges_t` and all memory associated with it.

```
typedef struct scf_int_ranges {
        int             scr_num_ranges;
        int64_t         *scr_min;
        int64_t         *scr_max;
} scf_int_ranges_t;
```

The `scf_tmpl_value_name_choices()` function will retrieve the set of property value choices that should be offered to a user. Some templates will not specify any choices. The caller is responsible for calling `scf_values_destroy()` on success.

The `scf_tmpl_value_count_range_choices()` function will retrieve the set of defined lower and upper bounds as defined by the property template and place them in ranges. Some templates will not specify any range choices.

The `scf_tmpl_value_int_range_constraints()` function will retrieve the set of defined lower and upper bounds as defined by the property template and place them in *ranges*. Some templates will not specify any range constraints.

The `scf_tmpl_value_common_name()` function will retrieve the value's common name as currently templated and place it in **out*. A locale (as described in `setlocale(3C)`) can be specified, or if the supplied locale is `NULL`, the current locale will be used. If a common name in the specified locale is not found, the function will also look for a common name in the C locale. Some templates will not specify the value common name. The caller is responsible for freeing the **out* buffer on success.

The `scf_tmpl_value_description()` function will retrieve the value's description as currently templated and place it in **out*. A locale (as described in `setlocale(3C)`) can be specified, or if the supplied locale is `NULL`, the current locale will be used. If a description in the

specified locale is not found, the function will also look for a description in the C locale. Some templates will not specify the value description. The caller is responsible for freeing the **out* buffer on success.

The `scf_tmpl_value_in_constraint()` function will check that the value provided matches the constraints as defined in the property template provided. This currently means it will determine if the value provided:

- is of the proper type for the property template defined,
- is within a range defined, if it is a numeric type, and
- is within the name constraints, if name constraints are defined.

If the template property does not define a type, ranges will be considered of the same type as the numeric values being checked. Some ranges might consider the value out of constraint when tested as one numeric type but within constraint if tested as other numeric type. Refer to `strtoull(3C)` and `strtoll(3C)` to see the implications when retrieving numeric values from the repository or converting strings to numeric values in `libscf(3LIB)`.

If *errs* is not `NULL`, an `scf_tmpl_error_t` will be created, populated and added to *errs* in case of a constraint violation. The caller is responsible for calling `scf_tmpl_errors_destroy()` to free memory allocated for all `scf_tmpl_error_t` associated to *errs*.

Return Values Upon successful completion, `scf_tmpl_prop_name()`, `scf_tmpl_prop_common_name()`, `scf_tmpl_prop_description()`, `scf_tmpl_prop_units()`, `scf_tmpl_value_common_name()`, and `scf_tmpl_value_description()` return the length of the string written, not including the terminating null byte. Otherwise, they return -1.

Upon successful completion, `scf_tmpl_prop_type()`, `scf_tmpl_prop_required()`, `scf_tmpl_prop_visibility()`, `scf_tmpl_prop_cardinality()`, `scf_tmpl_prop_internal_seps()`, `scf_tmpl_value_name_constraints()`, `scf_tmpl_value_count_range_constraints()`, `scf_tmpl_value_int_range_constraints()`, `scf_tmpl_value_name_choices()`, `scf_tmpl_value_count_range_choices()`, `scf_tmpl_value_int_range_choices()` return 0. Otherwise, they return -1.

The `scf_tmpl_value_in_constraint()` functions returns 0 on success, 1 if the value is not in the constraint, and -1 on failure.

Upon successful completion, `scf_tmpl_visibility_to_string()` returns a string of the visibility supplied.

Errors The `scf_tmpl_prop_name()`, `scf_tmpl_prop_type()`, `scf_tmpl_prop_required()`, `scf_tmpl_prop_common_name()`, `scf_tmpl_prop_description()`, `scf_tmpl_prop_units()`, `scf_tmpl_prop_visibility()`, `scf_tmpl_prop_cardinality()`, `scf_tmpl_prop_internal_seps()`, `scf_tmpl_value_name_constraints()`, `scf_tmpl_value_count_range_constraints()`, `scf_tmpl_value_int_range_constraints()`, `scf_tmpl_value_name_choices()`,

`scf_tmpl_value_count_range_choices()`, `scf_tmpl_value_int_range_choices()`, `scf_tmpl_value_common_name()`, `scf_tmpl_value_description()`, and `scf_tmpl_value_in_constraint()` functions will fail if:

`SCF_ERROR_BACKEND_ACCESS`	The storage mechanism that the repository server (`svc.configd(1M)`) chose for the operation denied access.
`SCF_ERROR_CONNECTION_BROKEN`	The connection to the repository was lost.
`SCF_ERROR_DELETED`	The template property group has been deleted.
`SCF_ERROR_HANDLE_DESTROYED`	The handle passed in has been destroyed.
`SCF_ERROR_INTERNAL`	An internal error occurred.
`SCF_ERROR_NO_MEMORY`	There is not enough memory to populate the `scf_pg_tmpl_t`.
`SCF_ERROR_NO_RESOURCES`	The server does not have adequate resources to complete the request.
`SCF_ERROR_NOT_BOUND`	The handle is not currently bound.
`SCF_ERROR_PERMISSION_DENIED`	The template could not be read due to access restrictions.
`SCF_ERROR_TEMPLATE_INVALID`	The template data is invalid.

The `scf_tmpl_prop_type()`, `scf_tmpl_prop_common_name()`, `scf_tmpl_prop_description()`, `scf_tmpl_prop_units()`, `scf_tmpl_prop_cardinality()`, `scf_tmpl_prop_internal_seps()`, `scf_tmpl_value_name_constraints()`, `scf_tmpl_value_count_range_constraints()`, `scf_tmpl_value_int_range_constraints()`, `scf_tmpl_value_name_choices()`, `scf_tmpl_value_count_range_choices()`, `scf_tmpl_value_int_range_choices()`, `scf_tmpl_value_common_name()`, and `scf_tmpl_value_description()`, functions will fail if:

`SCF_ERROR_NOT_FOUND`	The property does not exist or exists and has no value.

The `scf_tmpl_value_in_constraint()` function will fail if:

`SCF_ERROR_INVALID_ARGUMENT`	Value is not a valid `scf_value_t`.

The `scf_tmpl_prop_common_name()`, `scf_tmpl_prop_description()` and `scf_tmpl_prop_units()` functions will fail if:

`SCF_ERROR_INVALID_ARGUMENT`	The locale string is too long to make a property name.

The `scf_tmpl_value_common_name()` and `scf_tmpl_value_description()` functions will fail if:

`SCF_ERROR_INVALID_ARGUMENT` The value and locale strings are too long to make a property name.

The `scf_tmpl_value_count_range_constraints()` and `scf_tmpl_value_count_range_choices()` functions will fail if:

`SCF_ERROR_CONSTRAINT_VIOLATED` The range has negative values.

The `scf_tmpl_value_int_range_constraints()` and `scf_tmpl_value_int_range_choices()` functions will fail if:

`SCF_ERROR_CONSTRAINT_VIOLATED` The range values don't fit in a `int64_t`.

The `scf_tmpl_value_count_range_constraints()`, `scf_tmpl_value_int_range_constraints()`, `scf_tmpl_value_count_range_choices()` and `scf_tmpl_value_int_range_choices()` functions will fail if:

`SCF_ERROR_CONSTRAINT_VIOLATED` A range with *min* value > *max* value is found.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also

`svc.configd(1M)`, `scf_tmpl_prop_create(3SCF)`, `setlocale(3C)`, `strtoll(3C)`, `strtoull(3C)`, `attributes(5)`, `smf_template(5)`

Name scf_tmpl_validate_fmri, scf_tmpl_errors_destroy, scf_tmpl_next_error, scf_tmpl_reset_errors, scf_tmpl_strerror, scf_tmpl_error_type, scf_tmpl_error_source_fmri, scf_tmpl_error_pg_tmpl, scf_tmpl_error_pg, scf_tmpl_error_prop_tmpl, scf_tmpl_error_prop, scf_tmpl_error_value – template validation functions

Synopsis

```
cc [ flag... ] file... -lscf [ library... ]
#include <libscf.h>

int scf_tmpl_validate_fmri(scf_handle_t *h, const char *fmri,
     const char *snapshot, scf_tmpl_errors_t **errs, int flags);

void scf_tmpl_errors_destroy(scf_tmpl_errors_t *errs);

scf_tmpl_error_t *scf_tmpl_next_error(scf_tmpl_errors_t *errs,
     scf_tmpl_errors_t *err)

void scf_tmpl_reset_errors(scf_tmpl_errors_t *errs);

int scf_tmpl_strerror(scf_tmpl_error_t *err, char *s,
     size_t n, int flags);

int scf_tmpl_error_type(const scf_tmpl_error_t *err,
     scf_tmpl_error_type_t *type);

int scf_tmpl_error_source_fmri(const scf_tmpl_error_t *err,
     char *fmri);

int scf_tmpl_error_pg_tmpl(const scf_tmpl_error_t *err, char *name,
     char *type);

int scf_tmpl_error_pg(const scf_tmpl_error_t *err,
     char **name, char **type);

int scf_tmpl_error_prop_tmpl(const scf_tmpl_error_t *err, char **name,
     char **type);

int scf_tmpl_error_prop(const scf_tmpl_error_t *err, char **name,
     char **type,);

int scf_tmpl_error_value(const scf_tmpl_error_t *err, char**val);
```

Description The template validation functions offer a way to validate the configuration data of an service instance against the appropriate template data. The `scf_tmpl_validate_fmri()` function returns the full set of errors for the specified instance, and those errors can be printed or explored directly.

By default, the validation is performed on the composed data from the running snapshot of an instance. A different snapshot can be explicitly selected by specifying a valid snapshot name rather than `NULL` for the *snapshot* argument. If *flags* includes `SCF_TMPL_VALIDATE_FLAG_CURRENT`, the *snapshot* argument is ignored and the current configuration is used.

By default, these functions also explore template data defined by the service or instance itself, the service's restarter, and global template data. See `smf_template(5)` for more information about this composition.

An instance FMRI is required, and FMRIs that specify other entities (for example, services) are explicitly rejected.

The `scf_tmpl_validate_fmri()` function validates an instance FMRI against the template data in the repository. As described above, when the *snapshot* argument is `NULL`, the default running snapshot is used. If `scf_tmpl_errors_t **` is non-null, the structure is allocated and returned to the caller for further perusal or printing of the errors.

The `scf_tmpl_errors_destroy()` function destroys and frees the `scf_tmpl_errors_t` and all of the `scf_tmpl_error_t` structures to which it refers.

The `scf_tmpl_next_error()` function takes a pointer to a `scf_tmpl_errors_t` structure previously returned by `scf_tmpl_validate_fmri()`. On the first call, it returns a pointer to the first `scf_tmpl_error_t` found during validation. On subsequent calls, the next error is returned. To resume processing from the first error, the caller can use `scf_tmpl_reset_errors()`.

The contents of an `scf_tmpl_error_t` are determined by its type. Types added as additional validation checks are introduced. Based on the error type, a set of fields can be retrieved from the error.

`SCF_TERR_TYPE_INVALID`
  reserved invalid type

`SCF_TERR_MISSING_PG`
  required property group is missing

  template source FMRI
  property group template name and type

`SCF_TERR_WRONG_PG_TYPE`
  property group type is incorrect

  template source FMRI
  property group template name and type
  property group name and type

`SCF_TERR_MISSING_PROP`
  required property is missing

  template source FMRI
  property group template name and type

property template name and type

`SCF_TERR_WRONG_PROP_TYPE`
property type is incorrect

template source FMRI
property group template name and type
property template name and type
property group name and type
property name and type

`SCF_TERR_CARDINALITY_VIOLATION`
number of values violates cardinality

template source FMRI
property group template name and type
property template name and type
property group name and type
property name and type
cardinality and cardinality limits

`SCF_TERR_VALUE_CONSTRAINT_VIOLATED`
constraint violated for value

template source FMRI
property group template name and type
property template name and type
property group name and type
property name and type
value

`SCF_TERR_RANGE_VIOLATION`
value violated specified range

template source FMRI
property group template name and type
property template name and type
property group name and type
property name and type
value

`SCF_TERR_PROP_TYPE_MISMATCH`
value type is different from property type

template source FMRI
property group template name and type
property template name and type

`SCF_TERR_VALUE_OUT_OF_RANGE`
value is out of template defined range

template source FMRI
property group template name and type
property template name and type
value

`SCF_TERR_INVALID_VALUE`
value violates template defined constraints

template source FMRI
property group template name and type
property template name and type
value

The `SCF_TERR_PROP_TYPE_MISMATCH`, `SCF_TERR_VALUE_OUT_OF_RANGE` and `SCF_TERR_INVALID_VALUE` types are only set from calls to `scf_tmpl_value_in_constraint(3SCF)`.

The `scf_tmpl_error_type()` function retrieves the error type.

The `scf_tmpl_error_source_fmri()` function retrieves a string with the FMRI of the source of the template that was violated. This string is freed by `scf_tmpl_errors_destroy()`.

The `scf_tmpl_error_pg_tmpl()` function retrieves strings with the name and type of the property group template that was violated. If the property group name or type was implicitly wildcarded (see `smf_template(5)`) in the template, this function returns a string containing `SCF_TMPL_WILDCARD` ("*"). These strings are freed by `scf_tmpl_errors_destroy()`.

The `scf_tmpl_error_pg()` function retrieves strings with the name and type of the property group that was violated. These strings are freed by `scf_tmpl_errors_destroy()`.

The `scf_tmpl_error_prop_tmpl()` function retrieves strings with the name and type of the property template that was violated. If the property type was implicitly wildcarded (see `smf_template(5)`) in the template, this function returns a string containing `SCF_TMPL_WILDCARD` ("*"). These strings are freed by `scf_tmpl_errors_destroy()`.

The `scf_tmpl_error_prop()` function retrieves strings with the name and type of the property that was violated. These strings are freed by `scf_tmpl_errors_destroy()`.

The `scf_tmpl_error_value()` function retrieves a string with the value containing the error in *val*. This string are freed by `scf_tmpl_errors_destroy()`.

The `scf_tmpl_strerror()` function takes an `scf_tmpl_error_t` previously returned by `scf_tmpl_next_error()` and returns in *s*. If flags includes `SCF_TMPL_STRERROR_HUMAN`, *s* is a human-readable, localized description of the error. Otherwise, *s* is a one-line string suitable for logfile output.

Return Values The `scf_tmpl_validate_fmri()` function returns 0 on successful completion with no validation failures. It returns 1 if there are validation failures. It returns -1 if there is an error validating the instance.

The `scf_tmpl_next_error()` function returns a pointer to the next `scf_tmpl_error_t`. When none remain, it returns `NULL`.

The `scf_tmpl_error_type()`, `scf_tmpl_error_source_fmri()`, `scf_tmpl_error_pg_tmpl()`, `scf_tmpl_error_pg()`, `scf_tmpl_error_prop_tmpl()`, `scf_tmpl_error_prop()`, and `scf_tmpl_error_value()` functions return 0 on success and -1 on failure.

The `scf_tmpl_strerror()` function returns the number of bytes that would have been written to s if n had been sufficiently large.

Errors The `scf_tmpl_validate_fmri()` function will fail if:

`SCF_ERROR_BACKEND_ACCESS`	The storage mechanism that the repository server (`svc.configd(1M)`) chose for the operation denied access.
`SCF_ERROR_CONNECTION_BROKEN`	The connection to the repository was lost.
`SCF_ERROR_DELETED`	The instance or one of its template property group have been deleted.
`SCF_ERROR_HANDLE_DESTROYED`	The handle passed in has been destroyed.
`SCF_ERROR_INTERNAL`	An internal error occurred.
`SCF_ERROR_INVALID_ARGUMENT`	The handle argument, FMRI argument, or snapshot name is invalid
`SCF_ERROR_NO_MEMORY`	There is not enough memory to validate the instance.
`SCF_ERROR_NO_RESOURCES`	The server does not have adequate resources to complete the request.
`SCF_ERROR_NOT_BOUND`	The handle is not currently bound.
`SCF_ERROR_NOT_FOUND`	An object matching FMRI does not exist in the repository, or the snapshot does not exist.
`SCF_ERROR_PERMISSION_DENIED`	The instance or template could not be read due to access restrictions.
`SCF_ERROR_TEMPLATE_INVALID`	The template data is invalid.

The `scf_tmpl_strerror()`, `scf_tmpl_error_type()`, `scf_tmpl_error_source_fmri()`, `scf_tmpl_error_pg_tmpl()`, `scf_tmpl_error_pg()`, `scf_tmpl_error_prop_tmpl()`, `scf_tmpl_error_prop()`, and `scf_tmpl_error_value()` functions will fail if:

`SCF_ERROR_INVALID_ARGUMENT` The `scf_tmpl_errors_t` argument is invalid.

The `scf_tmpl_error_type()`, `scf_tmpl_error_source_fmri()`, `scf_tmpl_error_pg_tmpl()`, `scf_tmpl_error_pg()`, `scf_tmpl_error_prop_tmpl()`, `scf_tmpl_error_prop()`, and `scf_tmpl_error_value()` functions will fail if:

`SCF_ERROR_NOT_FOUND` The data requested is not available for the `scf_tmpl_error_t` argument supplied.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also

`svc.configd(1M)`, `scf_tmpl_value_in_constraint(3SCF)`, `attributes(5)`, `smf_template(5)`

Name scf_transaction_create, scf_transaction_handle, scf_transaction_reset, scf_transaction_reset_all, scf_transaction_destroy, scf_transaction_destroy_children, scf_transaction_start, scf_transaction_property_delete, scf_transaction_property_new, scf_transaction_property_change, scf_transaction_property_change_type, scf_transaction_commit – create and manipulate transaction in the Service Configuration Facility

Synopsis

```
cc [ flag... ] file... -lscf [ library... ]
#include <libscf.h>

scf_transaction_t *scf_transaction_create(scf_handle_t *handle);

scf_handle_t *scf_transaction_handle(scf_transaction_t *tran);

void scf_transaction_reset(scf_transaction_t *tran);

void scf_transaction_reset_all(scf_transaction_t *tran);

void scf_transaction_destroy(scf_transaction_t *tran);

void scf_transaction_destroy_children(scf_transaction_t *tran);

int scf_transaction_start(scf_transaction_t *tran,
     scf_propertygroup_t *pg);

int scf_transaction_property_delete(scf_transaction_t *tran,
     scf_transaction_entry_t *entry, const char *prop_name);

int scf_transaction_property_new(scf_transaction_t *tran,
     scf_transaction_entry_t *entry, const char *prop_name,
     scf_type_t type);

int scf_transaction_property_change(scf_transaction_t *tran,
     scf_transaction_entry_t *entry, const char *prop_name,
     scf_type_t type);

int scf_transaction_property_change_type(
     scf_transaction_t *tran, scf_transaction_entry_t *entry,
     const char *prop_name, scf_type_t type);

int scf_transaction_commit(scf_transaction_t *tran);
```

Description Transactions are the mechanism for changing property groups. They act atomically, whereby either all of the updates occur or none of them do. An `scf_transaction_t` is always in one of the following states:

reset — The initial state. A successful return of `scf_transaction_start()` moves the transaction to the started state.

started — The transaction has started. The `scf_transaction_property_delete()`, `scf_transaction_property_new()`, `scf_transaction_property_change()`, and `scf_transaction_property_change_type()` functions can be used to set

up changes to properties. The `scf_transaction_reset()` and `scf_transaction_reset_all()` functions return the transaction to the reset state.

committed — A call to `scf_transaction_commit()` (whether or not it is successful) moves the transaction to the committed state. Modifying, resetting, or destroying the entries and values associated with a transaction will move it to the invalid state.

invalid — The `scf_transaction_reset()` and `scf_transaction_reset_all()` functions return the transaction to the reset state.

The `scf_transaction_create()` function allocates and initializes an `scf_transaction_t` bound to *handle*. The `scf_transaction_destroy()` function resets, destroys, and frees *tran*. If there are any entries associated with the transaction, `scf_transaction_destroy()` also effects a call to `scf_transaction_reset()`. The `scf_transaction_destroy_children()` function resets, destroys, and frees all entries and values associated the transaction.

The `scf_transaction_handle()` function gets the handle to which *tran* is bound.

The `scf_transaction_start()` function sets up the transaction to modify the property group to which *pg* is set. The time reference used by *pg* becomes the basis of the transaction. The transaction fails if the property group has been modified since the last update of *pg* at the time when `scf_transaction_commit()` is called.

The `scf_transaction_property_delete()`, `scf_transaction_property_new()`, `scf_transaction_property_change()`, and `scf_transaction_property_change_type()` functions add a new transaction entry to the transaction. Each property the transaction affects must have a unique `scf_transaction_entry_t`. Each `scf_transaction_entry_t` can be associated with only a single transaction at a time. These functions all fail if the transaction is not in the started state, *prop_name* is not a valid property name, or *entry* is already associated with a transaction. These functions affect commit and failure as follows:

`scf_transaction_property_delete()`
This function deletes the property *prop_name* in the property group. It fails if *prop_name* does not name a property in the property group.

`scf_transaction_property_new()`
This function adds a new property prop_name to the property group with a value list of type *type*. It fails if *prop_name* names an existing property in the property group.

`scf_transaction_property_change()`
This function changes the value list for an existing property *prop_name* in the property group. It fails if *prop_name* does not name an existing property in the property group or names an existing property with a different type.

`scf_transaction_property_change_type()`
This function changes the value list and type for an existing property *prop_name* in the property group. It fails if *prop_name* does not name an existing property in the property group.

If the function call is successful, *entry* remains active in the transaction until `scf_transaction_destroy()`, `scf_transaction_reset()`, or `scf_transaction_reset_all()` is called. The `scf_entry_add_value(3SCF)` manual page provides information for setting up the value list for entries that are not associated with `scf_transaction_property_delete()`. Resetting or destroying an entry or value active in a transaction will move it into the invalid state.

The `scf_transaction_commit()` function attempts to commit *tran*.

The `scf_transaction_reset()` function returns the transaction to the reset state and releases all of the transaction entries that were added.

The `scf_transaction_reset_all()` function returns the transaction to the reset state, releases all of the transaction entries, and calls `scf_value_reset(3SCF)` on all values associated with the entries.

Return Values Upon successful completion, `scf_transaction_create()` returns a new `scf_transaction_t`. Otherwise, it returns `NULL`.

Upon successful completion, `scf_transaction_handle()` returns the handle associated with the transaction. Otherwise, it returns `NULL`.

Upon successful completion, `scf_transaction_start()`, `scf_transaction_property_delete()`, `scf_transaction_property_new()`, `scf_transaction_property_change()`, and `scf_transaction_property_change_type()` return 0. Otherwise, they return –1.

The `scf_transaction_commit()` function returns 1 upon successful commit, 0 if the property group set in `scf_transaction_start()` is not the most recent, and -1 on failure.

Errors The `scf_transaction_create()` function will fail if:

`SCF_ERROR_INVALID_ARGUMENT`	The value of the *handle* argument is `NULL`.
`SCF_ERROR_NO_MEMORY`	There is not enough memory to allocate an `scf_transaction_t`.
`SCF_ERROR_NO_RESOURCES`	The server does not have adequate resources for a new transaction handle.

The `scf_transaction_handle()` function will fail if:

`SCF_ERROR_HANDLE_DESTROYED`	The handle associated with *tran* has been destroyed.

The `scf_transaction_start()` function will fail if:

SCF_ERROR_BACKEND_ACCESS	The repository backend refused the modification.
SCF_ERROR_BACKEND_READONLY	The repository backend refused modification because it is read-only.
SCF_ERROR_CONNECTION_BROKEN	The connection to the repository was lost.
SCF_ERROR_DELETED	The property group has been deleted.
SCF_ERROR_HANDLE_MISMATCH	The transaction and property group are not derived from the same handle.
SCF_ERROR_IN_USE	The transaction is not in the `reset` state. The `scf_transaction_reset()` and `scf_transaction_reset_all()` functions can be used to return the transaction to the `reset` state.
SCF_ERROR_NO_RESOURCES	The server does not have the resources to complete the request.
SCF_ERROR_NOT_BOUND	The handle was never bound or has been unbound.
SCF_ERROR_NOT_SET	The property group specified by *pg* is not set.
SCF_ERROR_PERMISSION_DENIED	The user does not have sufficient privileges to modify the property group.

The `scf_transaction_property_delete()`, `scf_transaction_property_new()`, `scf_transaction_property_change()`, and `scf_transaction_property_change_type()` functions will fail if:

SCF_ERROR_BACKEND_ACCESS	The storage mechanism that the repository server (`svc.configd(1M)`) chose for the operation denied access.
SCF_ERROR_CONNECTION_BROKEN	The connection to the repository was lost.
SCF_ERROR_DELETED	The property group the transaction is changing has been deleted.
SCF_ERROR_HANDLE_MISMATCH	The transaction and entry are not derived from the same handle.
SCF_ERROR_IN_USE	The property already has an entry in the transaction.
SCF_ERROR_INTERNAL	An internal error occurred.
SCF_ERROR_INVALID_ARGUMENT	The *prop_name* argument is not a valid property name.
SCF_ERROR_NO_RESOURCES	The server does not have the resources to complete the request.
SCF_ERROR_NOT_BOUND	The handle is not bound.

`SCF_ERROR_NOT_SET`	The transaction has not been started.
`SCF_ERROR_TYPE_MISMATCH`	The *tran* argument is not of a type compatible with *type*.

The `scf_transaction_property_delete()`, `scf_transaction_property_change()`, and `scf_transaction_property_change_type()` functions will fail if:

`SCF_ERROR_EXISTS`	The object already exists.
`SCF_ERROR_NOT_FOUND`	The property group does not contain a property named *prop_name*.

The `scf_transaction_property_new()`, `scf_transaction_property_change()`, and `scf_transaction_property_change_type()` functions will fail if:

`SCF_ERROR_INVALID_ARGUMENT`	The *prop_name* argument is not not a valid property name, or the *type* argument is an invalid type.

The `scf_transaction_property_new()` function will fail if:

`SCF_ERROR_EXISTS`	The property group already contains a property named *prop_name*.
`SCF_ERROR_NOT_FOUND`	Nothing of that name was found.

The `scf_transaction_property_change()` function will fail if:

`SCF_ERROR_TYPE_MISMATCH`	The property *prop_name* is not of type *type*.

The `scf_transaction_commit()` function will fail if:

`SCF_ERROR_BACKEND_READONLY`	The repository backend is read-only.
`SCF_ERROR_BACKEND_ACCESS`	The repository backend refused the modification.
`SCF_ERROR_NOT_BOUND`	The handle is not bound.
`SCF_ERROR_CONNECTION_BROKEN`	The connection to the repository was lost.
`SCF_ERROR_INVALID_ARGUMENT`	The transaction is in an invalid state.
`SCF_ERROR_DELETED`	The property group the transaction is acting on has been deleted.
`SCF_ERROR_NOT_SET`	The transaction has not been started.
`SCF_ERROR_PERMISSION_DENIED`	The user does not have sufficient privileges to modify the property group.
`SCF_ERROR_NO_RESOURCES`	The server does not have sufficient resources to commit the transaction.

The `scf_error(3SCF)` function can be used to retrieve the error value.

Examples

EXAMPLE 1 Set an existing boolean value to true.

```
tx = scf_transaction_create(handle);
e1 = scf_entry_create(handle);
v1 = scf_value_create(handle);

do {
     if (scf_pg_update(pg) == -1)
          goto fail;
     if (scf_transaction_start(tx, pg) == -1)
          goto fail;

     /* set up transaction entries */
     if (scf_transaction_property_change(tx, e1, "property",
        SCF_TYPE_BOOLEAN) == -1) {
            scf_transaction_reset(tx);
            goto fail;
     }
     scf_value_set_boolean(v1, 1);
     scf_entry_add_value(e1, v1);


     result = scf_transaction_commit(tx);

     scf_transaction_reset(tx);
} while (result == 0);

if (result < 0)
     goto fail;

/* success */

   cleanup:
scf_transaction_destroy(tx);
scf_entry_destroy(e1);
scf_value_destroy(v1);
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also

libscf(3LIB), scf_value_reset(3SCF), scf_error(3SCF), scf_pg_create(3SCF), attributes(5)

Name scf_value_create, scf_value_handle, scf_value_reset, scf_value_destroy, scf_value_type, scf_value_base_type, scf_value_is_type, scf_type_base_type, scf_value_get_boolean, scf_value_get_count, scf_value_get_integer, scf_value_get_time, scf_value_get_astring, scf_value_get_ustring, scf_value_get_opaque, scf_value_get_as_string, scf_value_get_as_string_typed, scf_value_set_boolean, scf_value_set_count, scf_value_set_integer, scf_value_set_time, scf_value_set_from_string, scf_value_set_astring, scf_value_set_ustring, scf_value_set_opaque – manipulate values in the Service Configuration Facility

Synopsis

```
cc [ flag... ] file... -lscf [ library... ]
#include <libscf.h>

scf_value_t *scf_value_create(scf_handle_t *h);

scf_handle_t *scf_value_handle(scf_value_t *v);

void scf_value_reset(scf_value_t *v);

void scf_value_destroy(scf_value_t *v);

int scf_value_type(scf_value_t *v);

int scf_value_base_type(scf_value_t *v);

int scf_value_is_type(scf_value_t *v, scf_type_t type);

int scf_type_base_type(scf_type_t type, scf_type_t *out);

int scf_value_get_boolean(scf_value_t *v, uint8_t *out);

int scf_value_get_count(scf_value_t *v, uint64_t *out);

int scf_value_get_integer(scf_value_t *v, int64_t *out);

int scf_value_get_time(scf_value_t *v, int64_t *seconds,
     int32_t *ns);

ssize_t scf_value_get_astring(scf_value_t *v, char *buf,
     size_t size);

ssize_t scf_value_get_ustring(scf_value_t *v, char *buf,
     size_t size);

ssize_t scf_value_get_opaque(scf_value_t *v, char *out,
     size_t len);

ssize_t scf_value_get_as_string(scf_value_t *v, char *buf,
     size_t size);

ssize_t scf_value_get_as_string_typed(scf_value_t *v,
     scf_type_t type, char *buf, size_t size);

void scf_value_set_boolean(scf_value_t *v, uint8_t in);

void scf_value_set_count(scf_value_t *v, uint64_t in);

void scf_value_set_integer(scf_value_t *v, int64_t in);
```

```
int scf_value_set_time(scf_value_t *v, int64_t seconds,
     int32_t ns);

int scf_value_set_from_string(scf_value_t *v, scf_type_t type,
     char *in);

int scf_value_set_astring(scf_value_t *v, const char *in);

int scf_value_set_ustring(scf_value_t *v, const char *in);

int scf_value_set_opaque(scf_value_t *v, void *in, size_t sz);
```

Description The `scf_value_create()` function creates a new, reset `scf_value_t` that holds a single typed value. The value can be used only with the handle specified by *h* and objects associated with *h*.

The `scf_value_reset()` function resets the value to the uninitialized state. The `scf_value_destroy()` function deallocates the object.

The `scf_value_type()` function retrieves the type of the contents of *v*. The `scf_value_is_type()` function determines if a value is of a particular type or any of its subtypes. The `scf_type_base_type()` function returns the base type of *type*. The `scf_value_base_type()` function returns the true base type of the value (the highest type reachable from the value's type).

Type Identifier	Base Type	Type Description
`SCF_TYPE_INVALID`		reserved invalid type
`SCF_TYPE_BOOLEAN`		single bit
`SCF_TYPE_COUNT`		unsigned 64-bit quantity
`SCF_TYPE_INTEGER`		signed 64-bit quantity
`SCF_TYPE_TIME`		signed 64-bit seconds, signed 32-bit nanoseconds in the range 0 <= *ns* < 1,000,000,000
`SCF_TYPE_ASTRING`		8-bit NUL-terminated string
`SCF_TYPE_OPAQUE`		opaque 8-bit data
`SCF_TYPE_USTRING`	`ASTRING`	8-bit UTF-8 string
`SCF_TYPE_URI`	`USTRING`	a URI string
`SCF_TYPE_FMRI`	`URI`	a Fault Management Resource Identifier
`SCF_TYPE_HOST`	`USTRING`	either a hostname, IPv4 address, or IPv6 address
`SCF_TYPE_HOSTNAME`	`HOST`	a fully-qualified domain name
`SCF_TYPE_NET_ADDR_V4`	`HOST`	a dotted-quad IPv4 address with optional network portion

Type Identifier	Base Type	Type Description
SCF_TYPE_NET_ADDR_V6	HOST	legal IPv6 address

The scf_value_get_boolean(), scf_value_get_count(), scf_value_get_integer(), scf_value_get_time(), scf_value_get_astring(), scf_value_get_ustring(), and scf_value_get_opaque() functions read a particular type of value from *v*.

The scf_value_get_as_string() and scf_value_get_as_string_typed() functions convert the value to a string form. For scf_value_get_as_string_typed(), the value must be a reachable subtype of *type*.

The scf_value_set_boolean(), scf_value_set_count(), scf_value_set_integer(), scf_value_set_time(), scf_value_set_astring(), scf_value_set_ustring(), and scf_value_set_opaque() functions set *v* to a particular value of a particular type.

The scf_value_set_from_string() function is the inverse of scf_value_get_as_string(). It sets *v* to the value encoded in *buf* of type *type*.

The scf_value_set_*() functions will succeed on scf_value_t objects that have already been set.

Return Values Upon successful completion, scf_value_create() returns a new, reset scf_value_t. Otherwise, it returns NULL.

Upon successful completion, scf_value_handle() returns the handle associated with *v*. Otherwise, it returns NULL.

The scf_value_base_type() function returns the base type of the value, or SCF_TYPE_INVALID on failure.

Upon successful completion, scf_value_type() returns the type of the value. Otherwise, it returns SCF_TYPE_INVALID.

Upon successful completion, scf_value_is_type(), scf_value_get_boolean(), scf_value_get_count(), scf_value_get_integer(), scf_value_get_time(), scf_value_set_time(), scf_value_set_from_string(), scf_value_set_astring(), scf_value_set_ustring(), and scf_value_set_opaque() return 0. Otherwise, they return -1.

Upon successful completion, scf_value_get_astring(), scf_value_get_ustring(), scf_value_get_as_string(), and scf_value_get_as_string_typed() return the length of the source string, not including the terminating null byte. Otherwise, they return -1.

Upon successful completion, scf_value_get_opaque() returns the number of bytes written. Otherwise, it returns -1.

Errors The `scf_value_create()` function will fail if:

`SCF_ERROR_HANDLE_DESTROYED` The handle associated with *h* has been destroyed.

`SCF_ERROR_INVALID_ARGUMENT` The handle is `NULL`.

`SCF_ERROR_NO_MEMORY` There is not enough memory to allocate an `scf_value_t`.

The `scf_value_handle()` function will fail if:

`SCF_ERROR_HANDLE_DESTROYED` The handle associated with *v* has been destroyed.

The `scf_value_set_time()` function will fail if:

`SCF_ERROR_INVALID_ARGUMENT` The nanoseconds field is not in the range 0 <= *ns* < 1,000,000,000.

The `scf_type_base_type()` function will fail if:

`SCF_ERROR_INVALID_ARGUMENT` The *type* argument is not a valid type.

The `scf_value_set_astring()`, `scf_value_set_ustring()`, `scf_value_set_opaque()`, and `scf_value_set_from_string()` functions will fail if:

`SCF_ERROR_INVALID_ARGUMENT` The *in* argument is not a valid value for the specified type or is longer than the maximum supported value length.

The `scf_type_base_type()`, `scf_value_is_type()`, and `scf_value_get_as_string_typed()` functions will fail if:

`SCF_ERROR_INVALID_ARGUMENT` The *type* argument is not a valid type.

The `scf_value_type()`, `scf_value_base_type()`, `scf_value_get_boolean()`, `scf_value_get_count()`, `scf_value_get_integer()`, `scf_value_get_time()`, `scf_value_get_astring()`, `scf_value_get_ustring()`, `scf_value_get_as_string()`, and`scf_value_get_as_string_typed()` functions will fail if:

`SCF_ERROR_NOT_SET` The *v* argument has not been set to a value.

The `scf_value_get_boolean()`, `scf_value_get_count()`, `scf_value_get_integer()`, `scf_value_get_time()`, `scf_value_get_astring()`, `scf_value_get_ustring()`, and `scf_value_get_as_string_typed()` functions will fail if:

`SCF_ERROR_TYPE_MISMATCH` The requested type is not the same as the value's type and is not in the base-type chain.

The `scf_error(3SCF)` function can be used to retrieve the error value.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also libscf(3LIB), scf_entry_add_value(3SCF), scf_error(3SCF), attributes(5)

Name setflabel – move file to zone with corresponding sensitivity label

Synopsis

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

#include <tsol/label.h>

int setflabel(const char *path, const m_label_t *label_p);
```

Description The file that is named by *path* is relabeled by moving it to a new pathname relative to the root directory of the zone corresponding to *label_p*. If the source and destination file systems are loopback mounted from the same underlying file system, the file is renamed. Otherwise, the file is copied and removed from the source directory.

The `setflabel()` function enforces the following policy checks:

- If the sensitivity label of *label_p* equals the existing sensitivity label, then the file is not moved.
- If the corresponding directory does not exist in the destination zone, or if the directory exists, but has a different label than *label_p*, the file is not moved. Also, if the file already exists in the destination directory, the file is not moved.
- If the sensitivity label of the existing file is not equal to the calling process label and the caller is not in the global zone, then the file is not moved. If the caller is in the global zone, the existing file label must be in a labeled zone (not `ADMIN_LOW` or `ADMIN_HIGH`).
- If the calling process does not have write access to both the source and destination directories, then the calling process must have `PRIV_FILE_DAC_WRITE` in its set of effective privileges.
- If the sensitivity label of *label_p* provides read only access to the existing sensitivity label (an upgrade), then the user must have the `solaris.label.file.upgrade` authorization. In addition, if the current zone is a labeled zone, then it must have been assigned the privilege `PRIV_FILE_UPGRADE_SL` when the zone was configured.
- If the sensitivity label of *label_p* does not provide access to the existing sensitivity label (a downgrade), then the calling user must have the `solaris.label.file.downgrade` authorization. In addition, if the current zone is a labeled zone, then it must have been assigned the privilege `PRIV_FILE_DOWNGRADE_SL` when the zone was configured.
- If the calling process is not in the global zone, and the user does not have the `solaris.label.range` authorization, then *label_p* must be within the user's label range and within the system accreditation range.
- If the existing file is in use (not tranquil) it is not moved. This tranquility check does not cover race conditions nor remote file access.

Additional policy constraints can be implemented by customizing the shell script `/etc/security/tsol/relabel`. See the comments in this file.

Return Values Upon successful completion, `setflabel()` returns 0. Otherwise it returns -1 and sets `errno` to indicate the error.

Errors The `setflabel()` function fails and the file is unchanged if:

`EACCES` Search permission is denied for a component of the path prefix of *path*.

The calling process does not have mandatory write access to the final component of path because the sensitivity label of the final component of path does not dominate the sensitivity label of the calling process and the calling process does not have `PRIV_FILE_MAC_WRITE` in its set of effective privileges.

`EBUSY` There is an open file descriptor reference to the final component of *path*.

`ECONNREFUSED` A connection to the label daemon could not be established.

`EEXIST` A file with the same name exists in the destination directory.

`EINVAL` Improper parameters were received by the label daemon.

`EISDIR` The existing file is a directory.

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

`EMLINK` The existing file is hardlinked to another file.

`ENAMETOOLONG` The length of the path argument exceeds `PATH_MAX`.

`ENOENT` The file referred to by *path* does not exist.

`EROFS` The file system is read-only or its label is `ADMIN_LOW` or `ADMIN_HIGH`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also `libtsol(3LIB)`, `attributes(5)`

"Setting a File Sensitivity Label" in *Solaris Trusted Extensions Developer's Guide*

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name smf_enable_instance, smf_disable_instance, smf_refresh_instance, smf_restart_instance, smf_maintain_instance, smf_degrade_instance, smf_restore_instance, smf_get_state – administrative interface to the Service Configuration Facility

Synopsis

```
cc [ flag... ] file... -lscf [ library... ]
#include <libscf.h>

int smf_enable_instance(const char *instance, int flags);

int smf_disable_instance(const char *instance, int flags);

int smf_refresh_instance(const char *instance);

int smf_restart_instance(const char *instance);

int smf_maintain_instance(const char *instance, int flags);

int smf_degrade_instance(const char *instance, int flags);

int smf_restore_instance(const char *instance);

char *smf_get_state(const char *instance);
```

Description These functions provide administrative control over service instances. Using these functions, an administrative tool can make a request to enable, disable, refresh, or restart an instance. All calls are asynchronous. They request an action, but do not wait to see if the action succeeds or fails.

The `smf_enable_instance()` function enables the service instance specified by *instance* FMRI. If *flags* is `SMF_TEMPORARY`, the enabling of the service instance is a temporary change, lasting only for the lifetime of the current system instance. The *flags* argument is set to `0` if no flags are to be use.

The `smf_disable_instance()` function places the service instance specified by *instance* FMRI in the disabled state and triggers the stop method (see `svc.startd(1M)`). If *flags* is `SMF_TEMPORARY`, the disabling of the service instance is a temporary change, lasting only for the lifetime of the current system instance. The *flags* argument is set to `0` if no flags are to be use.

The `smf_refresh_instance()` function causes the service instance specified by *instance* FMRI to re-read its configuration information.

The `smf_restart_instance()` function restarts the service instance specified by *instance* FMRI.

The `smf_maintain_instance()` function moves the service instance specified by *instance* into the maintenance state. If *flags* is `SMF_IMMEDIATE`, the instance is moved into maintenance state immediately, killing any running methods. If *flags* is `SMF_TEMPORARY`, the change to maintenance state is a temporary change, lasting only for the lifetime of the current system instance. The *flags* argument is set to `0` if no flags are to be use.

The `smf_degrade_instance()` function moves an online service instance into the degraded state. This function operates only on instances in the online state. The *flags* argument is set to `0` if no flags are to be use. The only available flag is `SMF_IMMEDIATE`, which causes the instance to be moved into the degraded state immediately.

The `smf_restore_instance()` function brings an instance currently in the maintenance to the uninitialized state, so that it can be brought back online. For a service in the degraded state, `smf_restore_instance()` brings the specified instance back to the online state.

The `smf_get_state()` function returns a pointer to a string containing the name of the instance's current state. The user is responsible for freeing this string. Possible state strings are defined as the following:

```
#define SCF_STATE_STRING_UNINIT         ((const char *)"uninitialized")
#define SCF_STATE_STRING_MAINT          ((const char *)"maintenance")
#define SCF_STATE_STRING_OFFLINE        ((const char *)"offline")
#define SCF_STATE_STRING_DISABLED       ((const char *)"disabled")
#define SCF_STATE_STRING_ONLINE         ((const char *)"online")
#define SCF_STATE_STRING_DEGRADED       ((const char *)"degraded")
```

Return Values Upon successful completion, `smf_enable_instance()`, `smf_disable_instance()`, `smf_refresh_instance()`, `smf_restart_instance()`, `smf_maintain_instance()`, `smf_degrade_instance()`, and `smf_restore_instance()` return `0`. Otherwise, they return –1.

Upon successful completion, smf_get_state returns an allocated string. Otherwise, it returns `NULL`.

Errors These functions will fail if:

`SCF_ERROR_NO_MEMORY`	The memory allocation failed.
`SCF_ERROR_INVALID_ARGUMENT`	The *instance* FMRI or *flags* argument is invalid.
`SCF_ERROR_NOT_FOUND`	The FMRI is valid but there is no matching instance found.
`SCF_ERROR_CONNECTION_BROKEN`	The connection to repository was broken.
`SCF_ERROR_NO_RESOURCES`	The server has insufficient resources.

The `smf_maintain_instance()`, `smf_refresh_instance()`, `smf_restart_instance()`, `smf_degrade_instance()`, and `smf_restore_instance()` functions will fail if:

`SCF_ERROR_PERMISSION_DENIED`	User does not have proper authorizations. See `smf_security(5)`.
`SCF_ERROR_BACKEND_ACCESS`	The repository's backend refused access.
`SCF_ERROR_BACKEND_READONLY`	The repository's backend is read-only.

The `smf_restore_instance()` and `smf_degrade_instance()` functions will fail if:

`SCF_ERROR_CONSTRAINT_VIOLATED` The function is called on an instance in an inappropriate state.

The `scf_error(3SCF)` function can be used to retrieve the error value.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also

`svc.startd(1M)`, `libscf(3LIB)`, `scf_error(3SCF)`, `attributes(5)`, `smf_security(5)`

Name SSAAgentIsAlive, SSAGetTrapPort, SSARegSubtable, SSARegSubagent, SSARegSubtree, SSASendTrap, SSASubagentOpen – Sun Solstice Enterprise Agent registration and communication helper functions

Synopsis

```
cc [ flag ... ] file ... -lssagent  -lssasnmp  [ library .. ]
#include <impl.h>



extern int SSAAgentIsAlive(IPAddress *agent_addr, int *port,
     char *community, struct timeval *timeout);

extern int SSAGetTrapPort();

extern int *SSARegSubagent(Agent* agent);

int SSARegSubtable(SSA_Table *table);

int SSARegSubtree(SSA_Subtree *subtree);

extern void SSASendTrap(char *name);

extern int SSASubagentOpen(int *num_of_retry, char *agent_name);
```

Description The `SSAAgentIsAlive()` function returns `TRUE` if the master agent is alive, otherwise returns `FALSE`. The *agent_addr* parameter is the address of the agent. Specify the security token in the *community* parameter. You can specify the maximum amount of time to wait for a response with the *timeout* parameter.

The `SSAGetTrapPort()` function returns the port number used by the Master Agent to communicate with the subagent.

The `SSARegSubagent()` function enables a subagent to register and unregister with a Master Agent. The *agent* parameter is a pointer to an `Agent` structure containing the following members:

```
int      timeout;         /* optional */
int      agent_id;        /* required */
int      agent_status;    /* required */
char     *personal_file;  /* optional */
char     *config_file;    /* optional */
char     *executable;     /* optional */
char     *version_string; /* optional */
char     *protocol;       /* optional */
int      process_id;      /* optional */
char     *name;           /* optional */
int      system_up_time;  /* optional */
int      watch_dog_time;  /* optional */
Address  address;         /* required */
struct   _Agent;          /* reserved */
struct   _Subtree;        /* reserved */
```

The agent_id member is an integer value returned by the SSASubagentOpen() function. After calling SSASubagentOpen(), you pass the agent_id in the SSARegSubagent() call to register the subagent with the Master Agent.

The following values are supported for agent_status:

```
SSA_OPER_STATUS_ACTIVE
SSA_OPER_STATUS_NOT_IN_SERVICE
SSA_OPER_STATUS_DESTROY
```

You pass SSA_OPER_STATUS_DESTROY as the value in a SSARegSubagent() function call when you want to unregister the agent from the Master Agent.

Address has the same structure as sockaddr_in, that is a common UNIX structure containing the following members:

```
short      sin_family;
ushort_t   sin_port;
struct     in_addr sin_addr;
char       sin_zero[8];
```

The SSARegSubtable() function registers a MIB table with the Master Agent. If this function is successful, an index number is returned, otherwise 0 is returned. The *table* parameter is a pointer to a SSA_Table structure containing the following members:

```
int  regTblIndex;         /* index value */
int  regTblAgentID;       /* current agent ID */
Oid  regTblOID;           /* Object ID of the table */
int  regTblStartColumn;   /* start column index */
int  regTblEndColumn;     /* end column index */
int  regTblStartRow;      /* start row index */
int  regTblEndRow;        /* end row index */
int  regTblStatus;        /* status */
```

The regTblStatus can have one of the following values:

```
SSA_OPER_STATUS_ACTIVE
SSA_OPER_STATUS_NOT_IN_SERVICE
```

The SSARegSubtree() function registers a MIB subtree with the master agent. If successful this function returns an index number, otherwise 0 is returned. The *subtree* parameter is a pointer to a SSA_Subtree structure containing the following members:

```
int  regTreeIndex;      /* index value */
int  regTreeAgentID;    /* current agent ID */
Oid  name;              /* Object ID to register */
int  regtreeStatus;     /* status */
```

The regtreeStatus can have one of the following values:

```
SSA_OPER_STATUS_ACTIVE
SSA_OPER_STATUS_NOT_IN_SERVICE
```

The `SSASendTrap()` function instructs the Master Agent to send a trap notification, based on the keyword passed with *name*. When your subagent MIB is compiled by `mibcodegen`, it creates a lookup table of the trap notifications defined in the MIB. By passing the name of the trap notification type as *name*, the subagent instructs the Master Agent to construct the type of trap defined in the MIB.

The `SSASubagentOpen()` function initializes communication between the subagent and the Master Agent. You must call this function before calling `SSARegSubagent()` to register the subagent with the Master Agent. The `SSASubagentOpen()` function returns a unique agent ID that is passed in the `SSARegSubagent()` call to register the subagent. If `0` is returned as the agent ID, the attempt to initialize communication with the Master Agent was unsuccessful. Since UDP is used to initialize communication with the Master Agent, you may want to set the value of *num_of_retry* to make multiple attempts.

The value for *agent_name* must be unique within the domain for which the Master Agent is responsible.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete
MT-Level	Unsafe

See Also

`attributes(5)`

Name SSAOidCmp, SSAOidCpy, SSAOidDup, SSAOidFree, SSAOidInit, SSAOidNew, SSAOidString, SSAOidStrToOid, SSAOidZero – Sun Solstice Enterprise Agent OID helper functions

Synopsis

```
cc [ flag ... ] file ... -lssasnmp [ library .. ]
#include <impl.h>

int SSAOidCmp(Oid *oid1, Oid *oid2);

int SSAOidCpy(Oid *oid1, Oid *oid2, char *error_label);

Oid *SSAOidDup(Oid *oid, char *error_label);

void SSAOidFree(Oid *oid);

int SSAOidInit(Oid *oid, Subid *subids, int len, char *error_label);

Oid *SSAOidNew();

char *SSAOidString(Oid *oid);

Oid *SSAOidStrToOid(char* name, char *error_label);

void SSAOidZero(Oid *oid);
```

Description The `SSAOidCmp()` function performs a comparison of the given OIDs. This function returns:

`0` if *oid1* is equal to *oid2*

`1` if *oid1* is greater than *oid2*

`−1` if *oid1* is less than *oid2*

The `SSAOidCpy()` function makes a deep copy of *oid2* to *oid1*. This function assumes *oid1* has been processed by the `SSAOidZero()` function. Memory is allocated inside *oid1* and the contents of *oid2*, not just the pointer, is copied to *oid1*. If an error is encountered, an error message is stored in the *error_label* buffer.

The `SSAOidDup()` function returns a clone of *oid*, by using the deep copy. Error information is stored in the *error_label* buffer.

The `SSAOidFree()` function frees the OID instance, with its content.

The `SSAOidNew()` function returns a new OID.

The `SSAOidInit()` function copies the Subid array from *subids* to the OID instance with the specified length *len*. This function assumes that the OID instance has been processed by the `SSAOidZero()` function or no memory is allocated inside the OID instance. If an error is encountered, an error message is stored in the *error_label* buffer.

The `SSAOidString()` function returns a char pointer for the printable form of the given *oid*.

The `SSAOidStrToOid()` function returns a new OID instance from *name*. If an error is encountered, an error message is stored in the *error_label* buffer.

The `SSAOidZero()` function frees the memory used by the OID object for buffers, but not the OID instance itself.

Return Values The `SSAOidNew()` and `SSAOidStrToOid()` functions return `0` if an error is detected.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete
MT-Level	Unsafe

See Also `attributes(5)`

Name SSAStringCpy, SSAStringInit, SSAStringToChar, SSAStringZero – Sun Solstice Enterprise Agent string helper functions

Synopsis

```
cc [ flag ... ] file ... -lssasnmp [ library .. ]
#include <impl.h>

void *SSAStringZero(String *string);

int SSAStringInit(String *string, uchar_t *chars, int len,
     char *error_label);

int SSAStringCpy(String *string1, String *string2, char *error_label);

char *SSAStringToChar(String string);
```

Description The `SSAStringCpy()` function makes a deep copy of *string2* to *string1*. This function assumes that *string1* has been processed by the `SSAStringZero()` function. Memory is allocated inside the *string1* and the contents of *string2*, not just the pointer, is copied to the *string1*. If an error is encountered, an error message is stored in the *error_label* buffer.

The `SSAStringInit()` function copies the char array from *chars* to the string instance with the specified length *len*. This function assumes that the string instance has been processed by the `SSAStringZero()` function or no memory is allocated inside the string instance. If an error is encountered, an error message is stored in the *error_label* buffer.

The `SSAStringToChar()` function returns a temporary char array buffer for printing purposes.

The `SSAStringZero()` function frees the memory inside of the String instance, but not the string object itself.

Return Values The `SSAStringInit()` and `SSAStringCpy()` functions return `0` if successful and `−1` if error.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete
MT-Level	Unsafe

See Also `attributes(5)`

Name stmfAddToHostGroup – add an initiator port to an existing host group

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfAddToHostGroup(stmfGroupName *hostGroupName,
     stmfDevid initiatorName);
```

Parameters

hostGroupName	The name of the host group to which the specified initiatorName is added.
initiatorName	The device identifier of the initiator port to add to the specified host group.

Description The `stmfAddToHostGroup()` function adds an initiator port to an existing host group.

Return Values The following values are returned:

`STMF_ERROR_EXISTS`	The specified *initiatorName* already exists in this *hostGroupName* or in another host group in the system.
`STMF_ERROR_GROUP_NOT_FOUND`	The specified *hostGroupName* was not found in the system.
`STMF_STATUS_SUCCESS`	The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Name stmfAddToTargetGroup – add a target to an existing target group

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfAddToTargetGroup(stmfGroupName *targetGroupName,
     stmfDevid targetName);
```

Parameters

targetGroupName	The name of the target port group to which the specified *targetName* is added.
targetName	The device identifier of the target port to add to the specified target group.

Description The `stmfAddToTargetGroup()` function adds a target to an existing target group.

Return Values The following values are returned:

`STMF_ERROR_EXISTS`	The specified *targetName* already exists in this *tagettGroupName* or in another target group in the system.
`STMF_ERROR_GROUP_NOT_FOUND`	The specified *targetGroupName* was not found in the system.
`STMF_STATUS_SUCCESS`	The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Name stmfAddViewEntry – add a view entry for a given logical unit

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfAddViewEntry(stmfGuid *logicalUnit,
     stmfViewEntry *viewEntry);
```

Parameters

logicalUnit The identifier of the logical unit to which this view entry is being added.

viewEntry The view entry to add to the specified logical unit identifier.

Description The `stmfAddViewEntry()` function adds a view entry for a given logical unit.

Return Values The following values are returned:

`STMF_ERROR_LUN_IN_USE` The specified logical unit number is already in use for this logical unit.

`STMF_ERROR_NOT_FOUND` The ID specified for *logicalUnit* was not found in the system.

`STMF_ERROR_VE_CONFLICT` Adding this view entry is in conflict with one or more existing view entries.

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Notes If *luNbrValid* in the `stmfViewEntry` structure is set to `B_FALSE`, the framework will assign a logical unit number for this view entry. *veIndexValid* must be set to `B_FALSE` when adding a view entry. On successful return, *veIndexValid* will be set to `B_TRUE` and *veIndex* will contain the view entry index assigned to this view entry by the framework.

Name stmfClearProviderData – delete all data for the specified provider

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfClearProviderData(char *providerName, int providerType);
```

Parameters

providerName The name of the provider whose data is being deleted.

providerType The value must be either `STMF_LU_PROVIDER_TYPE` or `STMF_PORT_PROVIDER_TYPE`.

Description The `stmfClearProviderData()` function deletes all data for the specified provider.

Return Values The following values are returned:

`STMF_ERROR_NOT_FOUND` The value specified for *providerName* was not found in the system.

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Name stmfCreateHostGroup – create a new host group

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfCreateHostGroup(stmfGroupName *hostGroupName);
```

Parameters *hostGroupName* The name of the host group to be created.

Description The `stmfCreateHostGroup()` function creates a new host group.

Return Values The following values are returned:

`STMF_ERROR_EXISTS`	The value specified for *hostGroupName* already exists in the system.
`STMF_INVALID_ARGUMENT`	The value specified for *hostGroupName* was not valid.
`STMF_STATUS_SUCCESS`	The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Name stmfCreateLu – create a logical unit

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfCreateLu(luResource hdl, stmfGuid *luGuid);
```

Parameters

hdl The logical unit resource returned from a previous call to `stmfCreateLuResource(3STMF)`.

luGuid If non-null, it must contain a pointer to an `stmfGuid` structure allocated by the caller. On successful return from this API, it will contain the guid of the newly created logical unit. If *luGuid* is `NULL`, this argument is ignored.

Description The stmfCreateLu function creates a logical unit in stmf using the properties of *hdl*. See `stmfSetLuProp(3STMF)` for a complete description of properties and their possible values.

Return Values The following values are returned:

`STMF_STATUS_SUCCESS`
The API call was successful.

`STMF_ERROR_FILE_IN_USE`
The filename specified by the `STMF_LU_PROP_DATA_FILENAME` or `STMF_LU_PROP_META_FILENAME` was in use.

`STMF_ERROR_GUID_IN_USE`
The guid specified by the `STMF_LU_PROP_GUID` property is already being used.

`STMF_ERROR_INVALID_BLKSIZE`
The blocksize specified by `STMF_LU_PROP_BLOCK_SIZE` is invalid.

`STMF_ERROR_WRITE_CACHE_SET`
The requested write cache setting could not be provided.

`STMF_ERROR_SIZE_OUT_OF_RANGE`
The specified logical unit size is not supported.

`STMF_ERROR_META_FILE_NAME`
The specified meta file could not be accessed.

`STMF_ERROR_DATA_FILE_NAME`
The specified data file could not be accessed.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also libstmf(3LIB), stmfCreateLuResource(3STMF), stmfSetLuProp(3STMF), attributes(5)

Name stmfCreateLuResource – create new logical unit resource

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfCreateLuResource(uint16_t dType, luResource *hdl);
```

Parameters

dType The device type of the logical unit resource. Only `STMF_DISK` is currently supported.

hdl The logical unit resource to be created.

Description The `stmfCreateLuResource()` function creates a resource for setting properties of a logical unit for purposes of creating a logical unit in STMF.

Return Values The following values are returned:

`STMF_ERROR_INVALID_ARG` Either type is unrecognized or *hdl* was `NULL`.

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Name stmfCreateTargetGroup – create a new target port group

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfCreateTargetGroup(stmfGroupName *targetGroupName);
```

Parameters *targetGroupName* The name of the target port group to be created.

Description The `stmfCreateTargetGroup()` function creates a new target port group.

Return Values The following values are returned:

`STMF_ERROR_EXISTS` The value specified for *targetGroupName* already exists in the system.

`STMF_INVALID_ARGUMENT` The value specified for *targetGroupName* was not valid.

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Name stmfDeleteHostGroup – delete an existing host group

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfDeleteHostGroup(stmfGroupName *hostGroupName);
```

Parameters *hostGroupName* The name of the host group being deleted.

Description The `stmfDeleteHostGroup()` function deletes an existing host group.

Return Values The following values are returned:

`STMF_ERROR_NOT_FOUND` The specified *hostGroupName* was not found in the system.

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Name stmfDeleteLu – delete a logical unit

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfDeleteLu(stmfGuid *luGuid);
```

Parameters *luGuid* a pointer to an `stmfGuid` structure containing the guid of the logical unit to delete

Description The `stmfDeleteLu()` function deletes the logical unit from the system. Any view entries that may exist for this logical unit will be retained in the system and must be removed using `stmfRemoveViewEntry(3STMF)` if so desired.

Return Values The following values are returned:

`STMF_STATUS_SUCCESS` The API call was successful.

`STMF_ERROR_NOT_FOUND` The guid does not exist.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `stmfRemoveViewEntry(3STMF)`, `attributes(5)`

Name stmfDeleteTargetGroup – delete an existing target port group

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfDeleteTargetGroup(stmfGroupName *targetGroupName);
```

Parameters *targetGroupName* The name of the target port group being deleted.

Description The `stmfDeleteTargetGroup()` function deletes an existing target port group.

Return Values The following values are returned:

`STMF_ERROR_NOT_FOUND` The specified *targetGroupName* was not found in the system.

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Name stmfDestroyProxyDoor – close the door interface

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

void stmfDestroyProxyDoor(int hdl);
```

Parameters *hdl* handle returned from a previous call to `stmfInitProxyDoor(3STMF)`

Description The `stmfDestroyProxyDoor()` function closes the door interface established in the call to `stmfInitProxyDoor()`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `stmfInitProxyDoor(3STMF)`, `attributes(5)`

Name stmfDevidFromIscsiName – convert an iSCSI name to a stmfDevid structure

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfDevidFromIscsiName(char *iscsiName, stmfDevid *devid);
```

Parameters

iscsiName A character string of UTF-8 encoded Unicode characters representing the iSCSI name terminated with the Unicode nul character.

devid A pointer to a `stmfDevid` structure allocated by the caller. On successful return, this will contain the converted device identifier. On error, the value of this parameter is undefined.

Description The `stmfDevidFromIscsiName()` function converts an iSCSI name to a `stmfDevid` structure. It returns the *devid* as a SCSI name string identifier.

Return Values The following values are returned:

`STMF_ERROR_INVALID_ARGUMENT` The value of *iscsiName* was not valid iSCSI name.

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Name stmfDevidFromWwn – convert a WWN to a stmfDevid structure

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfDevidFromWWN(uchar_t wwn[8], stmfDevid *devid);
```

Parameters

wwn The 8-byte WWN identifier.

devid A pointer to a `stmfDevid` structure allocated by the caller. On successful return, this will contain the converted device identifier. On error, the value of this parameter is undefined.

Description The stmfDevidFromWwn function convert a WWN to a stmfDevid structure. It returns the *devid* as a SCSI name string.

Return Values The following values are returned:

`STMF_ERROR_INVALID_ARGUMENT` The value of *wwn* was not valid WWN identifier.

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Name stmfFreeLuResource – free an allocated logical unit resource

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfFreeLuResource(luResource hdl);
```

Parameters

hdl A logical unit resource previously allocated in a call to `stmfCreateLuResource(3STMF)` or `stmfGetLuResource(3STMF)`.

Description The `stmfFreeLuResource()` function frees a logical unit resource that was previously allocated in a call to `stmfCreateLuResource(3STMF)` or `stmfGetLuResource(3STMF)`.

Return Values The following values are returned:

`STMF_STATUS_SUCCESS` The API call was successful.

`STMF_ERROR_INVALID_ARG` The *hdl* argument is not a valid logical unit resource.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `stmfCreateLuResource(3STMF)`, `stmfGetLuResource(3STMF)`, `attributes(5)`

Name stmfFreeMemory – free memory allocated by this library

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

void stmfFreeMemory(void *stmfMemory);
```

Parameters

memory A pointer to memory that was previously allocated by this library. If `stmfMemory()` is equal to `NULL`, the call will return successfully.

Description The `stmfFreeMemory()` function frees memory allocated by this library.

Return Values No values are returned.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Name stmfGetAluaState – return the Asymmetric Logical Unit Access State (ALUA) mode for STMF

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfGetAluaState(boolean_t *alua_enabled, uint32_t *node);
```

Parameters

alua_enabled Set to `B_TRUE` or `B_FALSE` on success.

node Set to 0 or 1 on success.

Description The `stmfGetAluaState()` function returns the Asymmetric Logical Unit Access State (ALUA) mode for STMF along with the node setting.

Return Values The following values are returned:

`STMF_ERROR_INVALID_ARG` Either *alua_enabled* or *node* was incorrectly set.

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Name stmfGetHostGroupList – retrieve the list of host groups

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfGetInitiatorGroupList(stmfGroupList **hostGroupList);
```

Parameters

hostGroupList A pointer to a pointer to an `stmfGroupList` structure. On successful return, this will contain a list of host groups.

Description The `stmfGetInitiatorGroupList()` function retrieves the list of host groups. The caller should call `stmfFreeMemory(3STMF)` when this list is no longer needed.

Return Values The following values are returned:

`STMF_ERROR_NOMEM` The library was unable to allocate sufficient memory for *hostGroupList*.

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `stmfFreeMemory(3STMF)`, `libstmf(3LIB)`, `attributes(5)`

Name stmfGetHostGroupMembers – retrieve the properties of the specified host group

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfGetHostGroupMembers(stmfGroupName *hostGroupName,
     stmfGroupProperties **groupProperties);
```

Parameters

hostGroupName The name of the host group whose member list is being retrieved.

groupProperties A pointer to a pointer to an `stmfGroupProperties` structure. On successful return, this will contain the properties for the specified *hostGroupName*.

Description The `stmfGetHostGroupMembers()` function retrieves the properties of the specified host group. The caller should call `stmfFreeMemory(3STMF)` when this list is no longer needed.

Return Values The following values are returned:

`STMF_ERROR_NOT_FOUND` The specified *hostGroupName* was not found in the system.

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `stmfFreeMemory(3STMF)`, `libstmf(3LIB)`, `attributes(5)`

Name stmfGetLogicalUnitList – retrieve the list of logical units

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfGetLogicalUnitList(stmfGuidList **logicalUnitList);
```

Parameters

logicalUnitList A pointer to a pointer to an `stmfGuidList` structure. On successful return, this will contain a list of logical units in the system.

Description The `stmfGetLogicalUnitList()` function retrieves the list of logical units. The caller should call `stmfFreeMemory(3STMF)` when this list is no longer needed.

Return Values The following values are returned:

`STMF_ERROR_NOMEM` The library was unable to allocate sufficient memory for *logicalUnitList*.

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `stmfFreeMemory(3STMF)`, `libstmf(3LIB)`, `attributes(5)`

Name stmfGetLogicalUnitProperties – retrieve the properties of the specified logical unit

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfGetLogicalUnitProperties(stmfGuid *logicalUnit,
     stmfLogicalUnitProperties *logicalUnitProps);
```

Parameters

logicalUnit The identifier of the logical unit whose properties are being retrieved.

logicalUnitProps A pointer to an `stmfLogicalUnitProperties` structure. On successful return, this will contain the properties for the specified *logicalUnitOid*.

Description The `stmfGetLogicalUnitProperties()` function retrieves the properties of the specified logical unit.

Return Values The following values are returned:

`STMF_ERROR_LOGICAL_UNIT_NOT_REGISTERED` The *logicalUnit* is not a valid registered logical unit in the system.

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Name stmfGetLuResource – get a logical unit resource for a currently registered logical unit

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfGetLuResource(stmfGuid *luGuid, luResource *hdl);
```

Parameters

luGuid The guid of logical unit to retrieve.

hdl The logical unit resource to create.

Description The `stmfGetLuResource()` function retrieves a logical unit resource *hdl* for a given logical unit. The *luGuid* argument must represent a currently registered stmf logical unit. This retrieved resource is a set of device-specific properties for a logical unit device. This allocates an `luResource` *hdl* of device type matching *luGuid*. The `stmfFreeLuResource(3STMF)` function should be used when *hdl* is no longer needed.

Return Values The following values are returned:

`STMF_STATUS_SUCCESS` The API call was successful.

`STMF_ERROR_NOT_FOUND` The guid does not exist.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `stmfFreeLuResource(3STMF)`, `attributes(5)`

Name stmfGetPersistMethod – get the current persistence method for stmf

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfGetPersistMethod(uint8_t *persistType, boolean_t serviceState);
```

Parameters

persistType On success, contains the current persistence setting based on *serviceState*.

serviceState When set to `B_TRUE`, *persistType* will contain the persist method currently set for the service. When set to `B_FALSE`, *persistType* will contain the persist method for the current library open.

Description The `stmfGetPersistMethod()` function retrieves the current persistent method setting for the service or for a given library open. When set to `B_TRUE`, retrieves the setting from the service.

Return Values The following values are returned:

`STMF_STATUS_SUCCESS` The API call was successful.

`STMF_ERROR_PERSIST_TYPE` Unable to retrieve persist type from service.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Name stmfGetProviderData – retrieve the data for the specified provider

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfGetProviderData(char *providerName, nvlist_t **nvl,
     int providerType);
```

Parameters

providerNane	The name of the provider for which data is being retrieved.
nvl	A pointer to a pointer to an `nvlist_t`. On success, this will contain the nvlist retrieved. Caller is responsible for freeing the returned nvlist by calling `nvlist_free(3NVPAIR)`.
providerType	The value for this parameter must be either `STMF_LU_PROVIDER_TYPE` or `STMF_PORT_PROVIDER_TYPE`.

Description The `stmfGetProviderData()` function retrieves the data for the specified provider.

Return Values The following values are returned:

`STMF_ERROR_NOMEM`	The library was unable to allocate sufficient memory to return the data.
`STMF_STATUS_SUCCESS`	The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed (Obsolete)
MT-Level	Safe

See Also `libstmf(3LIB)`, `nvlist_free(3NVPAIR)`, `stmfGetProviderDataProt(3STMF)`, `attributes(5)`

Notes The `stmfGetProviderData()` function is deprecated in favor of `stmfGetProviderDataProt(3STMF)` and may be removed in a future revision of `libstmf(3LIB)`.

Name stmfGetProviderDataProt – retrieve data for the specified provider

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
 #include <libstmf.h>

int stmfGetProviderDataProt(char *providerName, nvlist_t **nvl,
     int providerType, uint64_t *token);
```

Parameters

providerName	The name of the provider for which data is being retrieved.
nvl	A pointer to a pointer to an `nvlist_t`. On success, this will contain the nvlist retrieved. The caller is responsible for freeing the returned nvlist by calling `nvlist_free(3NVPAIR)`.
providerType	The value for this parameter must be either `STMF_LU_PROVIDER_TYPE` or `STMF_PORT_PROVIDER_TYPE`.
token	A pointer to a `uint64_t` allocated by the caller. On success, this will contain a token for the returned data that can be used in a call to `stmfSetProviderDataProt(3STMF)` to ensure that the data returned in this call is not stale. If this value is `NULL`, the token will be ignored.

Description The `stmfGetProviderDataProt()` function retrieves the data for the specified provider.

Return Values The following values are returned:

`STMF_ERROR_NOMEM`	The library was unable to allocate sufficient memory to return the data.
`STMF_STATUS_SUCCESS`	The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `nvlist_free(3NVPAIR)`, `stmfSetProviderDataProt(3STMF)`, `attributes(5)`

Name stmfGetState – retrieve the list of sessions on a target

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfGetState(stmfState *state);
```

Parameters *state* A pointer to an `stmfState` structure allocated by the caller.

Description The `stmfGetState()` function retrieves the list of target port groups.

Return Values The following values are returned:

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Name stmfGetTargetGroupList – retrieve the list of target port groups

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfGetTargetGroupList(stmfGroupList **targetGroupList);
```

Parameters

targetGroupList A pointer to a pointer to an `stmfGroupList` structure. On successful return, this will contain a list of target port group object identifiers.

Description The `stmfGetTargetGroupList()` function retrieves the list of target port groups. The caller should call `stmfFreeMemory(3STMF)` when this list is no longer needed.

Return Values The following values are returned:

`STMF_ERROR_NOMEM` The library was unable to allocate sufficient memory for *targetGroupList*.

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `stmfFreeMemory(3STMF)`, `attributes(5)`

Name stmfGetTargetGroupMembers – retrieve the properties of the specified target port group

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfGetTargetGroupMembers(stmfGroupName *targetGroupName,
     stmfGroupProperties **groupProperties);
```

Parameters

targetGroupName The name of the target port group whose member list is being retrieved.

groupProperties A pointer to a pointer to an `stmfGroupProperties` structure. On successful return, this will contain the properties for the specified *targetGroupName*.

Description The `stmfGetTargetGroupMembers()` function retrieves the properties of the specified target port group. The caller should call `stmfFreeMemory(3STMF)` when this list is no longer needed.

Return Values The following values are returned:

`STMF_ERROR_NOT_FOUND` The specified *targetGroupName* was not found in the system.

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `stmfFreeMemory(3STMF)`, `attributes(5)`

Name stmfGetTargetList – retrieve the list of target ports

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfGetTargetList(stmfDevidList **targetList);
```

Parameters

targetList A pointer to a pointer to an `stmfDevidList` structure. On successful return, this will contain a list of target ports in the system.

Description The `stmfGetTargetList()` function retrieves the list of target ports. The caller should call `stmfFreeMemory(3STMF)` when this list is no longer needed.

Return Values The following values are returned:

`STMF_ERROR_NOMEM` The library was unable to allocate sufficient memory for *targetList*.

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `stmfFreeMemory(3STMF)`, `attributes(5)`

Name stmfGetTargetProperties – retrieve the properties of the specified target port

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfGetTargetProperties(stmfDevid *target,
     stmfTargetProperties *targetProperties);
```

Parameters

target The identifier of the target port whose properties are being retrieved.

targetProperties A pointer to an `stmfTargetProperties` structure allocated by the caller. On successful return, the structure will contain the properties for the specified.

Description The `stmfGetTargetProperties()` function retrieves the properties of the specified target port.

Return Values The following values are returned:

`STMF_ERROR_NOT_FOUND` The specified target was not found in the system.

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Name stmfGetViewEntryList – retrieve the list of view entries for a specified logical unit

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfGetViewEntryList(stmfGuid *logicalUnit,
     stmfViewEntryList **viewEntryList);
```

Parameters

logicalUnit The identifier of the logical unit for which to retrieve the list of view entries.

viewEntryList A pointer to a pointer to an `stmfViewEntryList` structure. On successful return, this will contain a list of view entries for *logicalUnit*.

Description The `stmfGetViewEntryList()` function retrieves the list of view entries for a specified logical unit. The caller should call `stmfFreeMemory(3STMF)` when this list is no longer needed.

Return Values The following values are returned:

`STMF_ERROR_NOMEM` The library was unable to allocate sufficient memory for *viewEntryList*.

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `stmfFreeMemory(3STMF)`, `attributes(5)`

Name stmfImportLu – import a logical unit

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfImportLu(uint16_t dType, char *fname, stmfGuid *luGuid);
```

Parameters

dtype the device type of the logical unit being imported. Only `STMF_DISK` is currently supported.

fname the filename of the logical unit being imported

luGuid pointer to a `stmfGuid` allocated by the caller. On success, this contains the `guid` of the imported logical unit. If *luGuid* is `NULL`, this parameter is ignored.

Description The `stmfImportLu()` function imports a previously created logical unit. The *fname* argument must be set to the filename where the metadata for the logical unit is stored. See `stmfCreateLu(3STMF)`.

Return Values The following values are returned:

`STMF_STATUS_SUCCESS`	The API call was successful.
`STMF_ERROR_INVALID_ARG`	The *dType* or *fname* argument was invalid.
`STMF_ERROR_META_FILE_NAME`	The specified meta file could not be accessed.
`STMF_ERROR_DATA_FILE_NAME`	The data file could not be accessed.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `stmfCreateLu(3STMF)`, `attributes(5)`

Name stmfInitProxyDoor – establish the door server with the STMF proxy service

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfInitProxyDoor(int *hdl, int fd);
```

Parameters

hdl a pointer to an `int` that will contain the handle for the proxy door to be used in calls to `stmfPostProxyMsg(3STMF)` and `stmfDestroyProxyDoor(3STMF)`.

fd the door file descriptor for the established door server

Description The `stmfInitProxyDoor()` function establishes the door server with the STMF proxy service. The STMF proxy service is responsible for sending SCSI commands to the peer node on behalf of a logical unit in the Standby asymmetric logical unit access (ALUA) state. `stmfInitProxyDoor()` should be called once a peer-to-peer communication channel between the two participating ALUA nodes has been established by the caller.

The `door_call(3C)` from the STMF proxy service to the door server will fill in the `door_arg_t` structure as follows:

```
door_arg_t arg;
uint32_t result;

arg.data_ptr = buf;
arg.data_size = size;
arg.desc_ptr = NULL;
arg.desc_num = 0;
arg.rbuf = (char *)&result
arg.rsize = sizeof (result);
```

The tuple <*data_ptr*, *data_size*> is expected to arrive at the peer node STMF proxy service via `stmfPostProxyMsg()`.

The door server is expected to complete the door call with these arguments to `door_return(3C)`:

```
uinit32_t result;

(void) door_return((char *)&result, sizeof(result), NULL, 0);
```

where result is of type `uint32_t` and set to 0 on success, non-zero on failure.

Non-zero values are logged as errors without further action. No file descriptors will be exchanged by the door call or return.

Return Values The following values are returned:

`STMF_ERROR_DOOR_INSTALLED` A previous door has already been established.

STMF_STATUS_SUCCESS	The API call was successful.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also door_call(3C), door_return(3C), libstmf(3LIB), stmfDestroyProxyDoor(3STMF), stmfPostProxyMsg(3STMF), attributes(5)

Name stmfLuStandby – set the access state of a logical unit to standby mode

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfSetAluaState(stmfGuid *luGuid);
```

Parameters

luGuid a pointer to an `stmfGuid` structure containing the `guid` of the logical unit to set to standby

Description The `stmfLuStandby()` function sets the access state of a logical unit to standby mode. When successfully set, a standby logical unit switches its asymmetric logical unit access state to a one of "Transition to Standby" (see `stmfGetLuProp(3STMF)`. Once moved to this state, the backing store for the logical unit will be released by the logical unit provider (`sbd` for disk devices). To move a logical unit out of "Standby" or the "Transition to Standby" state, `stmfImportLu(3STMF)` or the `import-lu` subcommand of `stmfadm(1M)` must be executed on the logical unit. On a successful logical unit import, the access state of the logical unit will move to "Active" in addition to sending a message to its peer that will complete the peer's transition to "Standby". The current access state for the logical unit can be retrieved using `stmfGetLuProp()` where the property type is `STMF_LU_PROP_ACCESS_STATE`.

Return Values The following values are returned:

`STMF_ERROR_NOT_FOUND` The `guid` does not exist.

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `stmfadm(1M)`, `libstmf(3LIB)`, `stmfGetLuProp(3STMF)`, `stmfImportLu(3STMF)`, `attributes(5)`

Name stmfModifyLu, stmfModifyLuByFname – modify a logical unit

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfModifyLu(stmfGuid *luGuid, uint32_t prop, const char *propVal)

int stmfModifyLu(uint16_t dType, const char *fname, uint32_t prop,
     const char *propVal)
```

Parameters

luGuid The `guid` of logical unit to modify.

fname The filename of logical unit to modify.

dType Type of logical unit. See `stmfCreateLuResource(3STMF)`.

prop A property type value. See DESCRIPTION for valid values.

propVal A property value.

Description The `stmfModifyLu()` and `stmfModifyLuByFname()` functions modify the properties of a logical unit device.

Valid properties for modify `STMF_DISK`:

`STMF_LU_PROP_ACCESS_STATE`
Asymmetric access state for the logical unit. Set to one of:

0 Active

1 Transition to Active

2 Standby

3 Transition to Standby

`STMF_LU_PROP_ALIAS`
Up to 255 characters representing a user defined name for the device.

Default: Set to file name of backing store.

`STMF_LU_PROP_SIZE`
Numeric value with optional suffix (for example, 100G, 1T) to specify unit of size.

Default: Size of device specified in the `STMF_LU_PROP_DATA_FILENAME` property value.

`STMF_LU_PROP_WRITE_CACHE_DISABLE`
Write back cache disable. When specified as “true” or “false”, specifies write back cache disable behavior.

Default: Writeback cache setting of the backing store device specified by `STMF_LU_PROP_DATA_FILENAME`.

`STMF_LU_PROP_WRITE_PROTECT`
Write protect bit. When specified as “true” or “false”, specifies whether the device behaves as a write protected device.

Default: “false”

Return Values The following values are returned:

`STMF_STATUS_SUCCESS`	The API call was successful.
`STMF_ERROR_INVALID_ARG`	Either *prop* or *propVal* is unrecognized.
`STMF_ERROR_INVALID_PROPSIZE`	The size of *propVal* is invalid.
`STMF_ERROR_INVALID_PROP`	The value of *prop* is unknown for this resource type.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `stmfCreateLuResource(3STMF)`, `attributes(5)`

Name stmfOfflineLogicalUnit – take offline a logical unit that is currently in the online state

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfOfflineLogicalUnit(stmfGuid *logicalUnit);
```

Parameters *logicalUnit* The identifier of the logical unit to offline.

Description The `stmfOfflineLogicalUnit()` function takes offline a logical unit that is currently in the online state. Once in the offline state, the logical unit will no longer be capable of servicing requests in the system.

This API call can be used to take offline a logical unit for servicing. Once the logical unit is offline, an initiator port that attempts to issue any SCSI commands to the offlined logical unit will receive a check condition. For purposes of the REPORT LUNS command, the logical unit will no longer appear in the logical unit inventory for any initiator ports to which it is currently mapped by one or more view entries.

Return Values The following values are returned:

`STMF_ERROR_BUSY` The device is currently busy.

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Name stmfOfflineTarget – take offline a target port that is currently in the online state

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfOfflineTarget(stmfDevid *target);
```

Parameters *target* The identifier of the target port to offline.

Description The `stmfOfflineTarget()` function takes offline a target port that is currently in the online state. Once in the offline state, the target port will no longer be capable of servicing requests in the system.

This API call can be used to take offline a target port device for servicing. Once the target port is offline, it will no longer be available to any entities outside of the SCSI Target Mode Framework. Any initiator ports that currently have sessions established by the offlined target port will be logged out.

Return Values The following values are returned:

`STMF_ERROR_BUSY` The device is currently busy.

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Name stmfOnlineLogicalUnit – take online of a logical unit that is currently in the offline state

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfOnlineLogicalUnit(stmfGuid *logicalUnit);
```

Parameters *logicalUnit* The identifier of the logical unit to take online.

Description The `stmfOnlineLogicalUnit()` function takes online of a logical unit that is currently in the offline state.

Return Values The following values are returned:

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Name stmfOnlineTarget – take online a target port that is currently in the offline state

Synopsis

```
cc [ flag ... ] file... -lstmf [ library ... ]
#include <libstmf.h>

int stmfOnlineTarget(stmfDevid *target);
```

Parameters *target* The identifier of the target port to online.

Description The `stmfOnlineTarget()` function takes online a target port that is currently in the offline state.

Return Values The following values are returned:

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Name stmfPostProxyMsg – post proxy message

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfPostProxyMsg(int hdl, void *buf, uint32_t buflen);
```

Parameters

hdl	handle returned in a previous successful call to `stmfInitProxyDoor(3STMF)`
buf	pointer to a buffer to received from peer node
buflen	length of *buf*

Description The `stmfPostProxyMsg()` function passes down to the STMF proxy service the message received from the peer node's STMF proxy service door upcall.

Return Values The following values are returned:

`STMF_ERROR_INVALID_ARG`	The *buf* argument is `NULL`.
`STMF_POST_MSG_FAILED`	The attempt to post the message failed.
`STMF_STATUS_SUCCESS`	The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `stmfInitProxyDoor(3STMF)`, `libstmf(3LIB)`, `attributes(5)`

Name stmfRemoveFromHostGroup – remove an initiator port from an host group

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfRemoveFromHostGroup(stmfGroupName *hostGroupName
     stmfDevid *initiatorPortName);
```

Parameters

hostGroupName	The name of the host group from which the specified *hostGroupName* is being removed.
initiatorPortName	The device identifier of the initiator port to remove from the specified host group.

Description The `stmfRemoveFromHostGroup()` function removes an initiator port from an host group.

Return Values The following values are returned:

`STMF_ERROR_GROUP_NOT_FOUND`	The specified *hostGroupName* was not found in the system.
`STMF_ERROR_MEMBER_NOT_FOUND`	The specified *initiatorPortName* was not found in the system.
`STMF_STATUS_SUCCESS`	The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Name stmfRemoveFromTargetGroup – remove a target port from an target port group

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfRemoveFromTargetGroup(stmfGroupName *targetGroupName
     stmfDevid *targetName);
```

Parameters

targetGroupName	The name of the target port group from which the specified *targetGroupName* is being removed.
targetName	The device identifier of the target port to remove from the specified target port group.

Description The `stmfRemoveFromTargetGroup()` function removes a target port from an target port group.

Return Values The following values are returned:

`STMF_ERROR_GROUP_NOT_FOUND`	The specified *targetGroupName* was not found in the system.
`STMF_ERROR_MEMBER_NOT_FOUND`	The specified *targetName* was not found in the system.
`STMF_ERROR_TG_ONLINE`	The specified *targetName* must be offline.
`STMF_STATUS_SUCCESS`	The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Name stmfRemoveViewEntry – remove a view entry from the system

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfRemoveViewEntry(stmfGuid *logicalUnit,
     uint32_t viewEntry);
```

Parameters

logicalUnit The identifier of the logical unit for the view entry being removed.

viewEntry The numeric value of the view entry to be removed.

Description The `stmfRemoveViewEntry()` function removes a view entry from the system.

Return Values The following values are returned:

`STMF_ERROR_NOT_FOUND` The specified *logicalUnit* or *viewEntryName* was not found in the system.

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Name stmfSetAluaState – set the Asymmetric Logical Unit Access State (ALUA) mode for STMF

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfSetAluaState(boolean_t alua_enabled, uint32_t node);
```

Parameters

alua_enabled `B_TRUE` when enabling ALUA mode; `B_FALSE` when disabling ALUA mode.

node Must be the value 0 or 1.

Description The `stmfSetAluaState()` function sets the Asymmetric Logical Unit Access State (ALUA) mode for STMF. When *alua_enabled* is set to `B_FALSE`, *node* is ignored; otherwise, *node* must be set to 0 or 1. The *node* setting must be different for each node in a paired configuration. This function should be called only after the STMF proxy door service has been initialized (see `stmfInitProxyDoor(3STMF)`). When the ALUA state is enabled, all STMF logical units will be registered on the peer node as standby logical units. The standby logical units can then be exported to any SCSI initiator using the existing mechanisms in STMF, `stmfAddViewEntry(3STMF)` or the add-view subcommand of `stmfadm(1M)`. If ALUA mode is already enabled, it is valid to call this interface again with enabled set to `B_TRUE`. This action would result in a re-initialization of the ALUA mode and can be used during recovery of a failed peer node.

Return Values The following values are returned:

`STMF_ERROR_INVALID_ARG` Either *alua_enabled* or *node* was incorrectly set.

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `stmfadm(1M)`, `libstmf(3LIB)`, `stmfAddViewEntry(3STMF)`, `stmfInitProxyDoor(3STMF)`, `attributes(5)`

Name stmfSetLuProp, stmfGetLuProp – set or get a logical unit property

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfSetLuProp(luResource hdl, uint32_t prop, const char *propVal);

int stmfGetLuProp(luResource hdl, uint32_t prop, char *propVal,
     size_t *propLen);
```

Parameters

hdl A logical unit resource previously allocated by `stmfCreateLuResource(3STMF)` or `stmfGetLuResource(3STMF)`.

prop A property type value. See DESCRIPTION for valid values.

propVal A property value.

propLen The length of the specified property value. If *propLen* was of an insufficient size to hold the returned property value, *propLen* will contain the required size of the buffer and `STMF_ERROR_INVALID_ARG` will be returned.

Description These functions set or get property values. All property values are expressed in human-readable form. Boolean properties are expressed in case insensitive form of "true" or "false". Properties that are represented by ASCII hexadecimal contain no leading characters to indicate a base hexadecimal representation (that is, no leading "0x"). The *prop* argument can be one of the following values:

`STMF_LU_PROP_ACCESS_STATE`
Asymmetric access state for the logical unit. Set to one of:

0 Active

1 Transition to Standby

2 Standby

3 Transition to Active

`STMF_LU_PROP_ALIAS`
Up to 255 characters representing a user defined name for the device.

Default: Set to file name of backing store.

`STMF_LU_PROP_BLOCK_SIZE`
Numeric value for block size in bytes in 2^n.

Default: 512

`STMF_LU_PROP_COMPANY_ID`
Organizational Unique Identifier. 6 hexadecimal ASCII characters representing the IEEE OUI company id assignment. This will be used to generate the device identifier (GUID).

Default: 00144F

`STMF_LU_PROP_DATA_FILENAME`
Character value representing the file name of the backing store device.

Default: None

`STMF_LU_PROP_GUID`
ASCII hexadecimal string of 32 characters representing the unique identifier for the device. This must be of valid 32 hexadecimal ASCII characters representing a valid NAA Registered Extended Identifier.

Default: Set by framework to a generated value.

`STMF_LU_PROP_HOST_ID`
8 hexadecimal ASCII characters representing the host ID assignment. This will be used to generate the globally unique identifier (GUID) for the logical unit.

Default: identifer returned by `hostid(1)`.

`STMF_LU_PROP_META_FILENAME`
Metadata file name. When specified, will be used to hold the SCSI metadata for the logical unit.

Default: None. If this value is not specified, the value specified in `STMF_LU_PROP_DATA_FILENAME` will be used.

`STMF_LU_PROP_MGMT_URL`
Up to 1024 characters representing Management Network Address URLs. More than one URL can be passed using space delimited URLs.

`STMF_LU_PROP_PID`
Up to 16 characters of product identification that will be reflected in the Standard INQUIRY data returned for the device.

Default: sSet to COMSTAR.

`STMF_LU_PROP_SERIAL_NUM`
Serial Number. Specifies the SCSI Vital Product Data Serial Number (page 80h). It is a character value up to 252 bytes in length.

Default: None

`STMF_LU_PROP_SIZE`
Numeric value w/optional suffix, e.g. 100G, 1T, to specify unit of size.

Default: Size of the device specified in the `STMF_LU_PROP_DATA_FILENAME` property value.

`STMF_LU_PROP_VID`
8 characters of vendor identification per SCSI SPC-3 and will be reflected in the Standard INQUIRY data returned for the device.

Default: Set to SUN.

`STMF_LU_PROP_WRITE_CACHE_DISABLE`
Write back cache disable. When specified as "true" or "false", specifies write back cache disable behavior.

Default: Writeback cache setting of the backing store device specified by `STMF_LU_PROP_DATA_FILENAME`.

`STMF_LU_PROP_WRITE_PROTECT`
Write protect bit. When specified as "true" or "false", specifies whether the device behaves as a write protected device.

Default: "false"

Return Values The following values are returned:

`STMF_STATUS_SUCCESS`	The API call was successful.
`STMF_ERROR_INVALID_ARG`	Either *prop* or *propVal* is unrecognized.
`STMF_ERROR_INVALID_PROPSIZE`	The size of *propVal* is invalid.
`STMF_ERROR_INVALID_PROP`	The value of *prop* is unknown for this resource type.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `hostid(1)`, `libstmf(3LIB)`, `stmfCreateLuResource(3STMF)`, `stmfGetLuResource(3STMF)`, `attributes(5)`

Name stmfSetPersistMethod – set persistence method for the stmf service

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfSetPersistMethod(uint8_t persistType, boolean_t serviceSet);
```

Parameters

persistType The requested persistence setting. Can be either `STMF_PERSIST_SMF` or `STMF_PERSIST_NONE`.

serviceSet Set to indicate whether the setting should persist on the stmf service. When set to `B_FALSE`, this setting is only applicable for the duration of the current library open or until a subsequent call is made to change the setting.

Description The `stmfSetPersistMethod()` function sets the persistence method for stmf.

Return Values The following values are returned:

`STMF_STATUS_SUCCESS` The API call was successful.

`STMF_ERROR_INVALID_ARG` The *persistType* argument is invalid.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `attributes(5)`

Name stmfSetProviderData – set the data for the specified provider

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
#include <libstmf.h>

int stmfSetProviderData(char *providerName, nvlist_t *nvl,
     int providerType);
```

Parameters

providerName	The name of the provider for which data is being set.
nvl	A pointer to an `nvlist_t` containing the nvlist to be set.
providerType	The value must be either `STMF_LU_PROVIDER_TYPE` or `STMF_PORT_PROVIDER_TYPE`.

Description The `stmfSetProviderData()` function sets the data for the specified provider.

Return Values The following values are returned:

`STMF_ERROR_NOMEM`	The library was unable to allocate sufficient memory to return the data.
`STMF_STATUS_SUCCESS`	The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed (Obsolete)
MT-Level	Safe

See Also `libstmf(3LIB)`, `stmfSetProviderDataProt(3STMF)`, `attributes(5)`

Notes The `stmfSetProviderData()` function is deprecated in favor of `stmfSetProviderDataProt(3STMF)` and may be removed in a future revision of `libstmf(3LIB)`.

Name stmfSetProviderDataProt – retrieve data for the specified provider

Synopsis

```
cc [ flag... ] file... -lstmf [ library... ]
 #include <libstmf.h>

int stmfSetProviderDataProt(char *providerName, nvlist_t **nvl,
     int providerType, uint64_t *token);
```

Parameters

providerName	The name of the provider for which data is being set.
nvl	A pointer to a pointer to an `nvlist_t` containing the nvlist to be set.
providerType	The value for this parameter must be either `STMF_LU_PROVIDER_TYPE` or `STMF_PORT_PROVIDER_TYPE`.
token	A pointer to a `uint64_t` that contains the value returned from a successful call to `stmfGetProviderDataProt(3STMF)`. If this argument is `NULL`, the token is ignored. Otherwise, the token will be verified against the current data. If the token represents stale data, the call fails. On success, *token* will contain the new token for the data being set and can be used in subsequent calls to `stmfSetProviderData(3STMF)`. On failure the contents are undefined.

Description The `stmfSetProviderDataProt()` function sets the data for the specified provider.

Return Values The following values are returned:

`STMF_ERROR_PROV_DATA_STALE`	The token value represents stale data.
`STMF_STATUS_SUCCESS`	The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `nvlist_free(3NVPAIR)`, `stmfGetProviderDataProt(3STMF)`, `stmfSetProviderData(3STMF)`, `attributes(5)`

Name stmfValidateView – remove a target port from an target port group

Synopsis

```
cc [ flag... ] file... lstmf [ library... ]
#include <libstmf.h>

int stmfValidateView(stmfViewEntry *view);
```

Parameters *view* The view entry to validate or get the logical number.

Description The `stmfValidateView()` function validates the logical unit number. This is done by setting *view*->`luNbrValid` to `B_TRUE` and setting *view*->`luNbr` to the logical unit number. A valid logical unit number is in the range of 0-16383.

The `stmfValidateView()` function finds the next available logical unit numbere by setting *view*->`luNbrValid` to `B_FALSE`. On success, the available logical unit number is returned in *view*->`luNbr`. A logical unit number is considered to be available if it is not currently consumed by an existing view entry where the target group and host group matches the view entry passed into this function. Until the logical unit number is no longer available, any calls to this function will get the same logical unit number in *view*->`luNbr`.

Return Values The following values are returned:

`STMF_ERROR_LUN_IN_USE` The specified logical unit number is already in use for this logical unit.

`STMF_STATUS_SUCCESS` The API call was successful.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `libstmf(3LIB)`, `stmfAddViewEntry(3STMF)`, `attributes(5)`

Name stobl, stobsl, stobclear – translate character-coded labels to binary labels

Synopsis

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

#include <tsol/label.h>

int stobsl(const char *string, m_label_t *label, const int flags,
     int *error);

int stobclear(const char *string, m_label_t *clearance,
     const int flags, int *error);
```

Description The `stobsl()` and `stobclear()` functions translate character-coded labels into binary labels. They also modify an existing binary label by incrementing or decrementing it to produce a new binary label relative to its existing value.

The calling process must have `PRIV_SYS_TRANS_LABEL` in its set of effective privileges to perform label translation on character-coded labels that dominate the process's sensitivity label.

The generic form of an input character-coded label string is:

`[ + ]` *classification name* `] [ [ + | – ]` *word* `...`

Leading and trailing white space is ignored. Fields are separated by white space, a '/' (slash), or a ',' (comma). Case is irrelevant. If *string* starts with + or –, *string* is interpreted a modification to an existing label. If *string* starts with a classification name followed by a + or –, the new classification is used and the rest of the old label is retained and modified as specified by *string*. + modifies an existing label by adding words. – modifies an existing label by removing words. To the maximum extent possible, errors in *string* are corrected in the resulting binary label *label*.

The `stobsl()` and `stobclear()` functions also translate hexadecimal label representations into binary labels (see `hextob`(3TSOL)) when the string starts with `0x` and either `NEW_LABEL` or `NO_CORRECTION` is specified in *flags*.

The *flags* argument can take the following values:

`NEW_LABEL`	*label* contents is not used, is formatted as a label of the relevant type, and is assumed to be `ADMIN_LOW` for modification changes. If `NEW_LABEL` is not present, *label* is validated as a defined label of the correct type dominated by the process's sensitivity label.
`NO_CORRECTION`	No corrections are made if there are errors in the character-coded label *string*. *string* must be complete and contain all the label components that are required by the `label_encodings` file. The `NO_CORRECTION` flag implies the `NEW_LABEL` flag.
`0 (zero)`	The default action is taken.

The *error* argument is a return parameter that is set only if the function is unsuccessful.

The `stobsl()` function translates the character-coded sensitivity label string into a binary sensitivity label and places the result in the return parameter *label*.

The *flags* argument can be either `NEW_LABEL`, `NO_CORRECTION`, or 0 (zero). Unless `NO_CORRECTION` is specified, this translation forces the label to dominate the minimum classification, and initial compartments set that is specified in the `label_encodings` file and corrects the label to include other label components required by the `label_encodings` file, but not present in *string*.

The `stobclear()` function translates the character-coded clearance string into a binary clearance and places the result in the return parameter *clearance*.

The *flags* argument can be either `NEW_LABEL`, `NO_CORRECTION`, or `0` (zero). Unless `NO_CORRECTION` is specified, this translation forces the label to dominate the minimum classification, and initial compartments set that is specified in the `label_encodings` file and corrects the label to include other label components that are required by the `label_encodings` file, but not present in *string*. The translation of a clearance might not be the same as the translation of a sensitivity label. These functions use different tables of the `label_encodings` file that might contain different words and constraints.

Return Values These functions return 1 if the translation was successful and a valid binary label was returned. Otherwise they return 0 and the value of the *error* argument indicates the error.

Errors When these functions return zero, *error* contains one of the following values:

`−1` Unable to access the `label_encodings` file.

`0` The label *label* is not valid for this translation and the `NEW_LABEL` or `NO_CORRECTION` flag was not specified, or the label *label* is not dominated by the process's *sensitivity label* and the process does not have `PRIV_SYS_TRANS_LABEL` in its set of effective privileges.

`>0` The character-coded label *string* is in error. `error` is a one-based index into *string* indicating where the translation error occurred.

Files `/etc/security/tsol/label_encodings`
The label encodings file contains the classification names, words, constraints, and values for the defined labels of this system.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete
MT-Level	MT-Safe

The `stobsl()` and `stobclear()` functions are obsolete. Use the `str_to_label(3TSOL)` function instead.

See Also `blcompare(3TSOL)`, `hextob(3TSOL)`, `libtsol(3LIB)`, `str_to_label(3TSOL)`, `attributes(5)`

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

In addition to the `ADMIN_LOW` name and `ADMIN_HIGH` name strings defined in the `label_encodings` file, the strings "`ADMIN_LOW`" and "`ADMIN_HIGH`" are always accepted as character-coded labels to be translated to the appropriate `ADMIN_LOW` and `ADMIN_HIGH` label, respectively.

Modifying an existing `ADMIN_LOW` label acts as the specification of a `NEW_LABEL` and forces the label to start at the minimum label that is specified in the `label_encodings` file.

Modifying an existing `ADMIN_HIGH` label is treated as an attempt to change a label that represents the highest defined classification and all the defined compartments that are specified in the `label_encodings` file.

The `NO_CORRECTION` flag is used when the character-coded label must be complete and accurate so that translation to and from the binary form results in an equivalent character-coded label.

Name str_to_label – parse human readable strings to label

Synopsis

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

#include <tsol/label.h>

int str_to_label(const char *string, m_label_t **label,
    const m_label_type_t label_type, uint_t flags, int *error);
```

Description The `str_to_label()` function is a simple function to parse human readable strings into labels of the requested type.

The *string* argument is the string to parse. If *string* is the result of a `label_to_str()` conversion of type `M_INTERNAL`, *flags* are ignored, and any previously parsed label is replaced.

If **label* is `NULL`, `str_to_label()` allocates resources for *label* and initializes the label to the *label_type* that was requested before parsing *string*.

If **label* is not `NULL`, the label is a pointer to a mandatory label that is the result of a previously parsed label and *label_type* is ignored. The type that is used for parsing is derived from *label* for any type-sensitive operations.

If *flags* is `L_MODIFY_EXISTING`, the parsed string can be used to modify this label.

If *flags* is `L_NO_CORRECTION`, the previously parsed label is replaced and the parsing algorithm does not attempt to infer missing elements from string to compose a valid label.

If *flags* is `L_DEFAULT`, the previously parsed label is replaced and the parsing algorithm makes a best effort to imply a valid label from the elements of *string*.

If *flags* contains `L_CHECK_AR` logically OR-ed with another value, the resulting label will be checked to ensure that it is within the "Accreditation Range" of the DIA encodings schema. This flag is interpreted only for `MAC_LABEL` label types.

The caller is responsible for freeing the allocated resources by calling the `m_label_free()` function. *label_type* defines the type for a newly allocated label. The label type can be:

`MAC_LABEL` The string should be translated as a Mandatory Access Control (MAC) label.

`USER_CLEAR` The string should be translated as a label that represents the least upper bound of the labels that the user is allowed to access.

If *error* is `NULL`, do not return additional error information for `EINVAL`. The calling process must have mandatory read access to *label* and human readable *string*. Or the calling process must have the `sys_trans_label` privilege.

The manifest constants `ADMIN_HIGH` and `ADMIN_LOW` are the human readable strings that correspond to the Trusted Extensions policy `admin_high` and `admin_low` label values. See `labels(5)`.

Return Values Upon successful completion, the `str_to_label()` function returns 0. Otherwise, `-1` is returned, `errno` is set to indicate the error, and *error* provides additional information for `EINVAL`. Otherwise, *error* is a zero-based index to the string parse failure point.

Errors The `str_to_label()` function will fail if:

`EINVAL` Invalid parameter. `M_BAD_STRING` indicates that *string* could not be parsed. `M_BAD_LABEL` indicates that the label passed in was in error. `M_OUTSIDE_AR` indicates that the resulting label is not within the "Accreditation Range" specified in the DIA encodings schema.

`ENOTSUP` The system does not support label translations.

`ENOMEM` The physical limits of the system are exceeded by size bytes of memory which cannot be allocated.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe
Standard	See below.

Parsing types that are relative to Defense Intelligence Agency (DIA) encodings schema are Standard. Standard is specified in `label_encodings(4)`.

See Also `label_to_str(3TSOL)`, `libtsol(3LIB)`, `m_label(3TSOL)`, `label_encodings(4)`, `attributes(5)`, `labels(5)`

"Validating the Label Request Against the Printer's Label Range" in *Solaris Trusted Extensions Developer's Guide*

Warnings A number of the parsing rules rely on the DIA label encodings schema. The rules might not be valid for other label schemata.

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name sysevent_bind_handle, sysevent_unbind_handle – bind or unbind subscriber handle

Synopsis

```
cc [flag...] file ... -lsysevent [library ...]
#include <libsysevent.h>

sysevent_handle_t *sysevent_bind_handle(void (*event_handler)
     (sysevent_t *ev));

void sysevent_unbind_handle(sysevent_handle_t *sysevent_hdl);
```

Parameters

ev	pointer to `sysevent` buffer handle
event_handler	pointer to an event handling function
sysevent_hdl	pointer to a `sysevent` subscriber handle

Description The `sysevent_bind_handle()` function allocates memory associated with a subscription handle and binds it to the caller's *event_handler*. The *event_handler* is invoked during subsequent system event notifications once a subscription has been made with `sysevent_subscribe_event(3SYSEVENT)`.

The system event is represented by the argument *ev* and is passed as an argument to the invoked event delivery function, *event_handler*.

Additional threads are created to service communication between `syseventd(1M)` and the calling process and to run the event handler routine, *event_handler*.

The `sysevent_unbind_handle()` function deallocates memory and other resources associated with a subscription handle and deactivates all system event notifications for the calling process. All event notifications are guaranteed to stop upon return from `sysevent_unbind_handle()`.

Return Values The `sysevent_bind_handle()` function returns a valid `sysevent` subscriber handle if the handle is successfully allocated. Otherwise, `NULL` is returned and `errno` is set to indicate the error.

The `sysevent_unbind_handle()` function returns no value.

Errors The `sysevent_bind_handle()` function will fail if:

`EACCES`	The calling process has an ID other than the privileged user.
`EBUSY`	There are no resources available.
`EINVAL`	The pointer to the function *event_handler* is `NULL`.
`EMFILE`	The process has too many open descriptors.
`ENOMEM`	There are insufficient resources to allocate the handle.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also sysseventd(1M), sysevent_subscribe_event(3SYSEVENT), attributes(5)

Notes Event notifications are revoked by sysseventd when the bound process dies. Event notification is suspended if a signal is caught and handled by the *event_handler* thread. Event notification is also suspended when the calling process attempts to use fork(2) or fork1(2). Event notifications might be lost during suspension periods.

The libsysevent interfaces do not work at all in non-global zones.

Name sysevent_free – free memory for sysevent handle

Synopsis

```
cc [flag ...]  file... -lsysevent [library ...]
#include <libsysevent.h>

void sysevent_free(sysevent_t *ev);
```

Parameters *ev* handle to event an event buffer

Description The `sysevent_free()` function deallocates memory associated with an event buffer.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also `attributes(5)`

Notes The `libsysevent` interfaces do not work at all in non-global zones.

Name sysevent_get_attr_list – get attribute list pointer

Synopsis

```
cc [flag ...] file... -lsysevent -lnvpair [library ...]
#include <libsysevent.h>
#include <libnvpair.h>

int sysevent_get_attr_list(sysevent_t *ev, nvlist_t **attr_list);
```

Parameters

ev handle to a system event

attr_list address of a pointer to attribute list (`nvlist_t`)

Description The `sysevent_get_attr_list()` function updates *attr_list* to point to a searchable name-value pair list associated with the `sysevent` event, *ev*. The interface manages the allocation of the attribute list, but it is up to the caller to free the list when it is no longer needed with a call to `nvlist_free()`. See `nvlist_alloc(3NVPAIR)`.

Return Values The `sysevent_get_attr_list()` function returns `0` if the attribute list for *ev* is found to be valid. Otherwise it returns `−1` and sets `errno` to indicate the error.

Errors The `sysevent_get_attr_list()` function will fail if:

`ENOMEM` Insufficient memory available to allocate an `nvlist`.

`EINVAL` Invalid `sysevent` event attribute list.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also `sysevent(1M)`, `nvlist_alloc(3NVPAIR)`, `nvlist_lookup_boolean(3NVPAIR)`, `attributes(5)`

Notes The `libsysevent` interfaces do not work at all in non-global zones.

Name sysevent_get_class_name, sysevent_get_subclass_name, sysevent_get_size, sysevent_get_seq, sysevent_get_time – get class name, subclass name, ID or buffer size of event

Synopsis

```
cc [flag...] file... -lsysevent [library...]
#include <libsysevent.h>

char *sysevent_get_class_name(sysevent_t *ev);

char *sysevent_get_subclass_name(sysevent_t *ev);

int sysevent_get_size(sysevent_t *ev);

uint64_t sysevent_get_seq(sysevent_t *ev);

void sysevent_get_time(sysevent_t *ev, hrtime_t *etimep);
```

Parameters

ev handle to event

etimep pointer to high resolution event time variable

Description The `sysevent_get_class_name()` and `sysevent_get_subclass_name()` functions return, respectively, the class and subclass names for the provided event *ev*.

The `sysevent_get_size()` function returns the size of the event buffer, *ev*.

The `sysevent_get_seq()` function returns a unique event sequence number of event *ev*. The sequence number is reset on every system boot.

The `sysevent_get_time()` function writes the time the event was published into the variable pointed to by *etimep*. The event time is added to the event just before it is put into the kernel internal event queue.

Examples **EXAMPLE 1** Parse sysevent header information.

The following example parses sysevent header information from an application's event handler.

```
hrtime_t last_ev_time;
unit64_t last_ev_seq;

void
event_handler(sysevent_t *ev)
{
         sysevent_t *new_ev;
         int ev_sz;
         hrtime_t ev_time;
         uint64_t ev_seq;


         /* Filter on class and subclass */
         if (strcmp(EC_PRIV, sysevent_get_class_name(ev)) != 0) {
```

EXAMPLE 1 Parse sysevent header information. *(Continued)*

```
                return;
        } else if (strcmp("ESC_MYSUBCLASS,
            sysevent_get_subclass_name(ev)) != 0) {
                return;
        }

        /*
         * Check for replayed sysevent, time must
         * be greater than previously recorded.
         */
        sysevent_get_event_time(ev, &ev_time);
        ev_seq = sysevent_get_seq(ev);
        if (ev_time < last_ev_time ||
            (ev_time == last_ev_time && ev_seq <=
            last_ev_seq)) {
                return;
        }

        last_ev_time = ev_time;
        last_ev_seq = ev_seq;

        /* Store event for later processing */
        ev_sz = sysevent_get_size(ev):
        new_ev (sysevent_t *)malloc(ev_sz);
        bcopy(ev, new_ev, ev_sz);
        queue_event(new_ev);
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also

`attributes(5)`

Notes

The `libsysevent` interfaces do not work at all in non-global zones.

Name sysevent_get_vendor_name, sysevent_get_pub_name, sysevent_get_pid – get vendor name, publisher name or processor ID of event

Synopsis

```
cc [flag ...] file... -lsysevent [library ...]
#include <libsysevent.h>

char *sysevent_get_vendor_name(sysevent_t *ev);

char *sysevent_get_pub_name(sysevent_t *ev);

pid_t sysevent_get_pid(sysevent_t *ev);
```

Parameters *ev* handle to a system event object

Description The `sysevent_get_pub_name()` function returns the publisher name for the sysevent handle, *ev*. The publisher name identifies the name of the publishing application or kernel subsystem of the sysevent.

The `sysevent_get_pid()` function returns the process ID for the publishing application or `SE_KERN_PID` for sysevents originating in the kernel. The publisher name and PID are useful for implementing event acknowledgement.

The `sysevent_get_vendor_name()` function returns the vendor string for the publishing application or kernel subsystem. A vendor string is the company's stock symbol that provided the application or kernel subsystem that generated the system event. This information is useful for filtering sysevents for one or more vendors.

The interface manages the allocation of the vendor and publisher name strings, but it is the caller's responsibility to free the strings when they are no longer needed by calling `free(3MALLOC)`. If the new vendor and publisher name strings cannot be created, `sysevent_get_vendor_name()` and `sysevent_get_pub_name()` return a null pointer and may set errno to `ENOMEM` to indicate that the storage space available is insufficient.

Examples **EXAMPLE 1** Parse sysevent header information.

The following example parses sysevent header information from an application's event handler.

```
char *vendor;
char *pub;

void
event_handler(sysevent_t *ev)
{
        if (strcmp(EC_PRIV, sysevent_get_class_name(ev)) != 0) {
                return;
        }

        vendor = sysevent_get_vendor_name(ev);
```

EXAMPLE 1 Parse sysevent header information. *(Continued)*

```
        if (strcmp("SUNW", vendor) != 0) {
                free(vendor);
                return;
        }
        pub = sysevent_get_pub_name(ev);
        if (strcmp("test_daemon", pub) != 0) {
                free(vendor);
                free(pub);
                return;
        }
        (void) kill(sysevent_get_pid(ev), SIGUSR1);
        free(vendor);
        free(pub);
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also

`malloc(3MALLOC)`, `attributes(5)`

Notes

The `libsysevent` interfaces do not work at all in non-global zones.

Name sysevent_post_event – post system event for applications

Synopsis

```
cc [ flag... ] file... -lsysevent -lnvpair [ library... ]
#include <libsysevent.h>
#include <libnvpair.h>

int sysevent_post_event(char *class, char *subclass, char *vendor,
     char *publisher, nvlist_t *attr_list, sysevent_id_t *eid);
```

Parameters

attr_list pointer to an `nvlist_t`, listing the name-value attributes associated with the event, or `NULL` if there are no such attributes for this event

class pointer to a string defining the event class

eid pointer to a system unique identifier

publisher pointer to a string defining the event's publisher nam

subclass pointer to a string defining the event subclass

vendor pointer to a string defining the vendor

Description The `sysevent_post_event()` function causes a system event of the specified class, subclass, vendor, and publisher to be generated on behalf of the caller and queued for delivery to the sysevent daemon `sysevent`d(1M).

The vendor should be the company stock symbol (or similarly enduring identifier) of the event posting application. The publisher should be the name of the application generating the event.

For example, all events posted by Sun applications begin with the company's stock symbol, “SUNW”. The publisher is usually the name of the application generating the system event. A system event generated by `devfsadm`(1M) has a publisher string of `devfsadm`.

The publisher information is used by sysevent consumers to filter unwanted event publishers.

Upon successful queuing of the system event, a unique identifier is assigned to *eid*.

Return Values The `sysevent_post_event()` function returns `0` if the system event has been queued successfully for delivery. Otherwise it returns −1 and sets `errno` to indicate the error.

Errors The `sysevent_post_event()` function will fail if:

`ENOMEM` Insufficient resources to queue the system event.

`EIO` The `sysevent`d daemon is not responding and events cannot be queued or delivered at this time.

`EINVAL` Invalid argument.

`EPERM` Permission denied.

EFAULT A copy error occurred.

Examples

EXAMPLE 1 Post a system event event with no attributes.

The following example posts a system event event with no attributes.

```
if (sysevent_post_event(EC_PRIV, "ESC_MYSUBCLASS", "SUNW", argv[0],
    NULL), &eid == -1) {
        fprintf(stderr, "error logging system event\n");
}
```

EXAMPLE 2 Post a system event with two name-value pair attributes.

The following example posts a system event event with two name-value pair attributes, an integer value and a string.

```
nvlist_t        *attr_list;
uint32_t        uint32_val = 0XFFFFFFFF;
char            *string_val = "string value data";

if (nvlist_alloc(&attr_list, 0, 0) == 0) {
        err = nvlist_add_uint32(attr_list, "uint32 data", uint32_val);
        if (err == 0)
                err = nvlist_add_string(attr_list, "str data",
                    string_val);
        if (err == 0)
                err = sysevent_post_event(EC_PRIV, "ESC_MYSUBCLASS",
                    "SUNW", argv[0], attr_list, &eid);
        if (err != 0)
                fprintf(stderr, "error logging system event\n");
        nvlist_free(attr_list);
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also

devfsadm(1M), syseventd(1M), nvlist_add_boolean(3NVPAIR), nvlist_alloc(3NVPAIR), attributes(5)

Notes

The libsysevent interfaces do not work at all in non-global zones.

Name sysevent_subscribe_event, sysevent_unsubscribe_event – register or unregister interest in event receipt

Synopsis

```
cc  [ flag... ] file... -lsysevent [ library... ]
#include <libsysevent.h>

int sysevent_subscribe_event(sysevent_handle_t *sysevent_hdl,
     char *event_class, char **event_subclass_list,
     int num_subclasses);

void sysevent_unsubscribe_event(sysevent_handle_t *sysevent_hdl,
     char *event_class);
```

Parameters

event_class	system event class string
event_subclass_list	array of subclass strings
num_subclasses	number of subclass strings
sysevent_hdl	`sysevent` subscriber handle

Description The `sysevent_subscribe_event()` function registers the caller's interest in event notifications belonging to the class *event_class* and the subclasses contained in *event_subclass_list*. The subscriber handle *sysevent_hdl* is updated with the new subscription and the calling process receives event notifications from the event handler specified in *sysevent_bind_handle*.

System events matching *event_class* and a subclass contained in *event_subclass_list* published after the caller returns from `sysevent_subscribe_event()` are guaranteed to be delivered to the calling process. Matching system events published and queued prior to a call to `sysevent_subscribe_event()` may be delivered to the process's event handler.

The *num_subclasses* argument provides the number of subclass string elements in *event_subclass_list*.

A caller can use the event class `EC_ALL` to subscribe to all event classes and subclasses. The event class `EC_SUB_ALL` can be used to subscribe to all subclasses within a given event class.

Subsequent calls to `sysevent_subscribe_event()` are allowed to add additional classes or subclasses. To remove an existing subscription, `sysevent_unsubscribe_event()` must be used to remove the subscription.

The `sysevent_unsubscribe_event()` function removes the subscription described by *event_class* for *sysevent_hdl*. Event notifications matching event_class will not be delivered to the calling process upon return.

A caller can use the event class `EC_ALL` to remove all subscriptions for *sysevent_hdl*.

The library manages all subscription resources.

Return Values The `sysevent_subscribe_event()` function returns 0 if the subscription is successful. Otherwise, −1 is returned and `errno` is set to indicate the error.

The `sysevent_unsubscribe_event()` function returns no value.

Errors The `sysevent_subscribe_event()` function will fail if:

`EACCES` The calling process has an ID other than the privileged user.

`EINVAL` The *sysevent_hdl* argument is an invalid `sysevent` handle.

`ENOMEM` There is insufficient memory available to allocate subscription resources.

Examples **EXAMPLE 1** Subscribing for environmental events

```
#include <libsysevent.h>
#include <sys/nvpair.h>

static int32_t attr_int32;

#define    CLASS1        "class1"
#define    CLASS2        "class2"
#define    SUBCLASS_1    "subclass_1"
#define    SUBCLASS_2    "subclass_2"
#define    SUBCLASS_3    "subclass_3"
#define    MAX_SUBCLASS  3

static void
event_handler(sysevent_t *ev)
{
    nvlist_t *nvlist;

    /*
     * Special processing  for events (CLASS1, SUBCLASS_1) and
     * (CLASS2, SUBCLASS_3)
     */
    if ((strcmp(CLASS1, sysevent_get_class_name(ev)) == 0 &&
        strcmp(SUBCLASS_1, sysevent_get_subclass_name(ev)) == 0) ||
        (strcmp(CLASS2, sysevent_get_subclass_name(ev) == 0) &&
        strcmp(SUBCLASS_3, sysevent_get_subclass(ev)) == 0)) {
        if (sysevent_get_attr_list(ev, &nvlist) != 0)
            return;
        if (nvlist_lookup_int32(nvlist, "my_int32_attr", &attr_int32)
            != 0)
            return;

        /* Event Processing */
```

EXAMPLE 1 Subscribing for environmental events *(Continued)*

```
    } else {
        /* Event Processing */
    }

}


int
main(int argc, char **argv)
{
    sysevent_handle_t *shp;
    const char *subclass_list[MAX_SUBCLASS];

    /* Bind event handler and create subscriber handle */
    shp = sysevent_bind_handle(event_handler);
    if (shp == NULL)
        exit(1);

    /* Subscribe to all CLASS1 event notifications */
    subclass_list[0] = EC_SUB_ALL;
    if (sysevent_subscribe_event(shp, CLASS1, subclass_list, 1) != 0) {
        sysevent_unbind_handle(shp);
        exit(1);
    }

    /* Subscribe to CLASS2 events for subclasses: SUBCLASS_1,
     * SUBCLASS_2 and SUBCLASS_3
     */
    subclass_list[0] = SUBCLASS_1;
    subclass_list[1] = SUBCLASS_2;
    subclass_list[2] = SUBCLASS_3;
    if (sysevent_subscribe_event(shp, CLASS2, subclass_list,
        MAX_SUBCLASS) != 0) {
        sysevent_unbind_handle(shp);
        exit(1);
    }

    for (;;) {
        (void) pause();
    }
}
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also `sysevent_bind_handle(3SYSEVENT)`, `sysevent_get_attr_list(3SYSEVENT)`, `sysevent_get_class_name(3SYSEVENT)`, `sysevent_get_vendor_name(3SYSEVENT)`, `attributes(5)`

Notes The `libsysevent` interfaces do not work at all in non-global zones.

Name tnfctl_buffer_alloc, tnfctl_buffer_dealloc – allocate or deallocate a buffer for trace data

Synopsis

```
cc [ flag ... ] file ... -ltnfctl [ library ... ]
#include <tnf/tnfctl.h>

tnfctl_errcode_t tnfctl_buffer_alloc(tnfctl_handle_t *hndl,
     const char *trace_file_name, size_t trace_buffer_size);

tnfctl_buffer_dealloc(tnfctl_handle_t *hndl);
```

Description `tnfctl_buffer_alloc()` allocates a buffer to which trace events are logged. When tracing a process using a `tnfctl` handle returned by `tnfctl_pid_open(3TNF)`, `tnfctl_exec_open(3TNF)`, `tnfctl_indirect_open(3TNF)`, and `tnfctl_internal_open(3TNF)`), *trace_file_name* is the name of the trace file to which trace events should be logged. It can be an absolute path specification or a relative path specification. If it is relative, the current working directory of the process that is calling `tnfctl_buffer_alloc()` is prefixed to *trace_file_name*. If the named trace file already exists, it is overwritten. For kernel tracing, that is, for a tnfctl handle returned by `tnfctl_kernel_open(3TNF)`, trace events are logged to a trace buffer in memory; therefore, *trace_file_name* is ignored. Use `tnfxtract(1)` to extract a kernel buffer into a file.

trace_buffer_size is the size in bytes of the trace buffer that should be allocated. An error is returned if an attempt is made to allocate a buffer when one already exists. `tnfctl_buffer_alloc()` affects the trace attributes; use `tnfctl_trace_attrs_get(3TNF)` to get the latest trace attributes after a buffer is allocated.

`tnfctl_buffer_dealloc()` is used to deallocate a kernel trace buffer that is no longer needed. *hndl* must be a kernel handle, returned by `tnfctl_kernel_open(3TNF)`. A process's trace file cannot be deallocated using `tnfctl_buffer_dealloc()`. Instead, once the trace file is no longer needed for analysis and after the process being traced exits, use `rm(1)` to remove the trace file. Do not remove the trace file while the process being traced is still alive. `tnfctl_buffer_dealloc()` affects the trace attributes; use `tnfctl_trace_attrs_get(3TNF)` to get the latest trace attributes after a buffer is deallocated.

For a complete discussion of `tnf tracing`, see `tracing(3TNF)`.

Return Values `tnfctl_buffer_alloc()` and `tnfctl_buffer_dealloc()` return `TNFCTL_ERR_NONE` upon success.

Errors The following error codes apply to `tnfctl_buffer_alloc()`:

`TNFCTL_ERR_BUFEXISTS`	A buffer already exists.
`TNFCTL_ERR_ACCES`	Permission denied; could not create a trace file.
`TNFCTL_ERR_SIZETOOSMALL`	The *trace_buffer_size* requested is smaller than the minimum trace buffer size needed. Use `trace_min_size` of trace attributes in `tnfctl_trace_attrs_get(3TNF)` to determine the minimum size of the buffer.

TNFCTL_ERR_SIZETOOBIG	The requested trace file size is too big.
TNFCTL_ERR_BADARG	*trace_file_name* is NULL or the absolute path name is longer than MAXPATHLEN.
TNFCTL_ERR_ALLOCFAIL	A memory allocation failure occurred.
TNFCTL_ERR_INTERNAL	An internal error occurred.

The following error codes apply to `tnfctl_buffer_dealloc()`:

TNFCTL_ERR_BADARG	*hndl* is not a kernel handle.
TNFCTL_ERR_NOBUF	No buffer exists to deallocate.
TNFCTL_ERR_BADDEALLOC	Cannot deallocate a trace buffer unless tracing is stopped. Use tnfctl_trace_state_set(3TNF) to stop tracing.
TNFCTL_ERR_INTERNAL	An internal error occurred.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWtnfc
MT Level	MT-Safe

See Also

prex(1), rm(1), tnfxtract(1), TNF_PROBE(3TNF), libtnfctl(3TNF), tnfctl_exec_open(3TNF), tnfctl_indirect_open(3TNF), tnfctl_internal_open(3TNF), tnfctl_kernel_open(3TNF), tnfctl_pid_open(3TNF), tnfctl_trace_attrs_get(3TNF), tracing(3TNF), attributes(5)

Name tnfctl_close – close a tnfctl handle

Synopsis

```
cc [ flag ... ] file ... -ltnfctl [ library ... ]
#include <tnf/tnfctl.h>


tnfctl_errcode_t tnfctl_close(tnfctl_handle_t *hndl,
     tnfctl_targ_op_t action);
```

Description `tnfctl_close()` is used to close a tnfctl handle and to free up the memory associated with the handle. When the handle is closed, the tracing state and the states of the probes are not changed. `tnfctl_close()` can be used to close handles in any mode, that is, whether they were created by `tnfctl_internal_open(3TNF)`, `tnfctl_pid_open(3TNF)`, `tnfctl_exec_open(3TNF)`, `tnfctl_indirect_open(3TNF)`, or `tnfctl_kernel_open(3TNF)`.

The *action* argument is only used in direct mode, that is, if *hndl* was created by `tnfctl_exec_open(3TNF)` or `tnfctl_pid_open(3TNF)`. In direct mode, *action* specifies whether the process will proceed, be killed, or remain suspended. *action* may have the following values:

`TNFCTL_TARG_DEFAULT` Kills the target process if *hndl* was created with `tnfctl_exec_open(3TNF)`, but lets it continue if it was created with `tnfctl_pid_open(3TNF)`.

`TNFCTL_TARG_KILL` Kills the target process.

`TNFCTL_TARG_RESUME` Allows the target process to continue.

`TNFCTL_TARG_SUSPEND` Leaves the target process suspended. This is not a job control suspend. It is possible to attach to the process again with a debugger or with the `tnfctl_pid_open(3TNF)` interface. The target process can also be continued with `prun(1)`.

Return Values `tnfctl_close()` returns `TNFCTL_ERR_NONE` upon success.

Errors The following error codes apply to `tnfctl_close()`:

`TNFCTL_ERR_BADARG` A bad argument was sent in *action*.

`TNFCTL_ERR_INTERNAL` An internal error occurred.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWtnfc
MT Level	MT-Safe

See Also prex(1), prun(1), TNF_PROBE(3TNF), libtnfctl(3TNF), tnfctl_exec_open(3TNF), tnfctl_indirect_open(3TNF), tnfctl_kernel_open(3TNF), tnfctl_pid_open(3TNF), tracing(3TNF), attributes(5)

Name tnfctl_indirect_open, tnfctl_check_libs – control probes of another process where caller provides /proc functionality

Synopsis

```
cc [ flag ... ] file ... -ltnfctl [ library ... ]
#include <tnf/tnfctl.h>

tnfctl_errcode_t tnfctl_indirect_open(void *prochandle,
     tnfctl_ind_config_t *config, tnfctl_handle_t **ret_val);

tnfctl_errcode_t tnfctl_check_libs(tnfctl_handle_t *hndl);
```

Description The interfaces `tnfctl_indirect_open()` and `tnfctl_check_libs()` are used to control probes in another process where the `libtnfctl(3TNF)` client has already opened `proc(4)` on the target process. An example of this is when the client is a debugger. Since these clients already use `/proc` on the target, `libtnfctl(3TNF)` cannot use `/proc` directly. Therefore, these clients must provide callback functions that can be used to inspect and to update the target process. The target process must load `libtnfprobe.so.1` (defined in `<tnf/tnfctl.h>` as macro `TNFCTL_LIBTNFPROBE`).

The first argument *prochandle* is a pointer to an opaque structure that is used in the callback functions that inspect and update the target process. This structure should encapsulate the state that the caller needs to use `/proc` on the target process (the `/proc` file descriptor). The second argument, *config*, is a pointer to

```
typedef
struct tnfctl_ind_config {
    int (*p_read)(void *prochandle, paddr_t addr, char *buf,
                  size_t size);
    int (*p_write)(void *prochandle, paddr_t addr, char *buf,
                  size_t size);
    pid_t (*p_getpid)(void *prochandle);
    int (*p_obj_iter)(void *prochandle, tnfctl_ind_obj_f *func,
                  void *client_data);
} tnfctl_ind_config_t;
```

The first field *p_read* is the address of a function that can read `size` bytes at address *addr* in the target image into the buffer *buf*. The function should return `0` upon success.. The second field *p_write* is the address of a function that can write `size` bytes at address *addr* in the target image from the buffer *buf*. The function should return `0` upon success. The third field *p_getpid* is the address of a function that should return the process id of the target process (*prochandle*). The fourth field *p_obj_iter* is the address of a function that iterates over all load objects and the executable by calling the callback function *func* with *client_data*. If *func* returns `0`, *p_obj_iter* should continue processing link objects. If *func* returns any other value, *p_obj_iter* should stop calling the callback function and return that value. *p_obj_iter* should return `0` if it iterates over all load objects.

If a failure is returned by any of the functions in *config*, the error is propagated back as `PREX_ERR_INTERNAL` by the `libtnfctl` interface that called it.

The definition of `tnfctl_ind_obj_f` is:

```
typedef int
tnfctl_ind_obj_f(void *prochandle,
    const struct tnfctl_ind_obj_info *obj
    void *client_data);
typedef struct tnfctl_ind_obj_info {
    int     objfd;            /* -1 indicates fd not available */
    paddr_t text_base;        /* virtual addr of text segment */
    paddr_t data_base;        /* virtual addr of data segment */
    const char *objname;        /* null-term. pathname to loadobj */
} tnfctl_ind_obj_info_t;
```

objfd should be the file descriptor of the load object or executable. If it is −1, then *objname* should be an absolute pathname to the load object or executable. If *objfd* is not closed by `libtnfctl`, it should be closed by the load object iterator function. *text_base* and *data_base* are the addresses where the text and data segments of the load object are mapped in the target process.

Whenever the target process opens or closes a dynamic object, the set of available probes may change. See `dlopen(3C)` and `dlclose(3C)`. In indirect mode, call `tnfctl_check_libs()` when such events occur to make `libtnfctl` aware of any changes. In other modes this is unnecessary but harmless. It is also harmless to call `tnfctl_check_libs()` when no such events have occurred.

Return Values `tnfctl_indirect_open()` and `tnfctl_check_libs()` return `TNFCTL_ERR_NONE` upon success.

Errors The following error codes apply to `tnfctl_indirect_open()`:

`TNFCTL_ERR_ALLOCFAIL`	A memory allocation failure occurred.
`TNFCTL_ERR_BUSY`	Internal tracing is being used.
`TNFCTL_ERR_NOLIBTNFPROBE`	`libtnfprobe.so.1` is not loaded in the target process.
`TNFCTL_ERR_INTERNAL`	An internal error occurred.

The following error codes apply to `tnfctl_check_libs()`:

`TNFCTL_ERR_ALLOCFAIL`	A memory allocation failure occurred.
`TNFCTL_ERR_INTERNAL`	An internal error occurred.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWtnfc

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	MT-Safe

See Also prex(1), TNF_PROBE(3TNF), dlclose(3C), dlopen(3C), libtnfctl(3TNF), tnfctl_probe_enable(3TNF), tnfctl_probe_trace(3TNF), tracing(3TNF), proc(4), attributes(5)

Linker and Libraries Guide

Notes `tnfctl_indirect_open()` should only be called after the dynamic linker has mapped in all the libraries (rtld sync point) and called only after the process is stopped. Indirect process probe control assumes the target process is stopped whenever any `libtnfctl` interface is used on it. For example, when used for indirect process probe control, tnfctl_probe_enable(3TNF) and tnfctl_probe_trace(3TNF) should be called only for a process that is stopped.

Name tnfctl_internal_open – create handle for internal process probe control

Synopsis

```
cc [ flag ... ] file ... -ltnfctl [ library ... ]
#include <tnf/tnfctl.h>

tnfctl_errcode_t tnfctl_internal_open(tnfctl_handle_t **ret_val);
```

Description `tnfctl_internal_open()` returns in *ret_val* a pointer to an opaque handle that can be used to control probes in the same process as the caller (internal process probe control). The process must have `libtnfprobe.so.1` loaded. Probes in libraries that are brought in by `dlopen(3C)` will be visible after the library has been opened. Probes in libraries closed by a `dlclose(3C)` will not be visible after the library has been disassociated. See the `NOTES` section for more details.

Return Values `tnfctl_internal_open()` returns `TNFCTL_ERR_NONE` upon success.

Errors

`TNFCTL_ERR_ALLOCFAIL`	A memory allocation failure occurred.
`TNFCTL_ERR_BUSY`	Another client is already tracing this program (internally or externally).
`TNFCTL_ERR_NOLIBTNFPROBE`	`libtnfprobe.so.1` is not linked in the target process.
`TNFCTL_ERR_INTERNAL`	An internal error occurred.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWtnfc
MT Level	MT-Safe

See Also `ld(1)`, `prex(1)`, `TNF_PROBE(3TNF)`, `dlopen(3C)`, `dlclose(3C)`, `libtnfctl(3TNF)`, `tracing(3TNF)`, `attributes(5)`

Linker and Libraries Guide

Notes `libtnfctl` interposes on `dlopen(3C)` and `dlclose(3C)` in order to be notified of libraries being dynamically opened and closed. This interposition is necessary for internal process probe control to update its list of probes. In these interposition functions, a lock is acquired to synchronize on traversal of the library list maintained by the runtime linker. To avoid deadlocking on this lock, `tnfctl_internal_open()` should not be called from within the init section of a library that can be opened by `dlopen(3C)`.

Since interposition does not work as expected when a library is opened dynamically, `tnfctl_internal_open()` should not be used if the client opened `libtnfctl` through

dlopen(3C). In this case, the client program should be built with a static dependency on `libtnfctl`. Also, if the client program is explicitly linking in `-ldl`, it should link `-ltnfctl` before `-ldl`.

Probes in filtered libraries (see ld(1)) will not be seen because the filtee (backing library) is loaded lazily on the first symbol reference and not at process startup or dlopen(3C) time. A workaround is to call tnfctl_check_libs(3TNF) once the caller is sure that the filtee has been loaded.

Name tnfctl_kernel_open – create handle for kernel probe control

Synopsis

```
cc [ flag ... ] file ... -ltnfctl [ library ... ]
#include <tnf/tnfctl.h>

tnfctl_errcode_t tnfctl_kernel_open(tnfctl_handle_t **ret_val);
```

Description `tnfctl_kernel_open()` starts a kernel tracing session and returns in `ret_val` an opaque handle that can be used to control tracing and probes in the kernel. Only one kernel tracing session is possible at a time on a given machine. An error code of `TNFCTL_ERR_BUSY` is returned if there is another process using kernel tracing. Use the command

```
fuser -f /dev/tnfctl
```

to print the process id of the process currently using kernel tracing. Only a superuser may use `tnfctl_kernel_open()`. An error code of `TNFCTL_ERR_ACCES` is returned if the caller does not have the necessary privileges.

Return Values `tnfctl_kernel_open` returns `TNFCTL_ERR_NONE` upon success.

Errors

`TNFCTL_ERR_ACCES`	Permission denied. Superuser privileges are needed for kernel tracing.
`TNFCTL_ERR_BUSY`	Another client is currently using kernel tracing.
`TNFCTL_ERR_ALLOCFAIL`	Memory allocation failed.
`TNFCTL_ERR_FILENOTFOUND`	`/dev/tnfctl` not found.
`TNFCTL_ERR_INTERNAL`	Some other failure occurred.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWtnfc
MT Level	MT-Safe

See Also `prex(1)`, `fuser(1M)`, `TNF_PROBE(3TNF)`, `libtnfctl(3TNF)`, `tracing(3TNF)`, `tnf_kernel_probes(4)`, `attributes(5)`

Name tnfctl_pid_open, tnfctl_exec_open, tnfctl_continue – interfaces for direct probe and process control for another process

Synopsis

```
cc [ flag ... ] file ... -ltnfctl [ library ... ]
#include <tnf/tnfctl.h>

tnfctl_errcode_t tnfctl_pid_open(pid_t pid, tnfctl_handle_t **ret_val);

tnfctl_errcode_t tnfctl_exec_open(const char *pgm_name,
     char * const *argv, char * const *envp,
     const char *libnfprobe_path, const char *ld_preload,
     tnfctl_handle_t **ret_val);

tnfctl_errcode_t tnfctl_continue(tnfctl_handle_t *hndl,
     tnfctl_event_t *evt, tnfctl_handle_t **child_hndl);
```

Description The `tnfctl_pid_open()`, `tnfctl_exec_open()`, and `tnfctl_continue()` functions create handles to control probes in another process (direct process probe control). Either `tnfctl_pid_open()` or `tnfctl_exec_open()` will return a handle in *ret_val* that can be used for probe control. On return of these calls, the process is stopped. `tnfctl_continue()` allows the process specified by *hndl* to continue execution.

The `tnfctl_pid_open()` function attaches to a running process with process id of *pid*. The process is stopped on return of this call. The `tnfctl_pid_open()` function returns an error message if *pid* is the same as the calling process. See `tnfctl_internal_open(3TNF)` for information on internal process probe control. A pointer to an opaque handle is returned in *ret_val*, which can be used to control the process and the probes in the process. The target process must have `libtnfprobe.so.1` (defined in <tnf/tnfctl.h> as macro `TNFCTL_LIBTNFPROBE`) linked in for probe control to work.

The `tnfctl_exec_open()` function is used to `exec(2)` a program and obtain a probe control handle. For probe control to work, the process image to be `exec`'d must load `libtnfprobe.so.1`. The `tnfctl_exec_open()` function makes it simple for the library to be loaded at process start up time. The *pgm_name* argument is the command to `exec`. If *pgm_name* is not an absolute path, then the `$PATH` environment variable is used to find the *pgm_name*. *argv* is a null-terminated argument pointer, that is, it is a null-terminated array of pointers to null-terminated strings. These strings constitute the argument list available to the new process image. The *argv* argument must have at least one member, and it should point to a string that is the same as *pgm_name*. See `execve(2)`. The *libnfprobe_path* argument is an optional argument, and if set, it should be the path to the directory that contains `libtnfprobe.so.1`. There is no need for a trailing "/" in this argument. This argument is useful if `libtnfprobe.so.1` is not installed in `/usr/lib`. *ld_preload* is a space-separated list of libraries to preload into the target program. This string should follow the syntax guidelines of the `LD_PRELOAD` environment variable. See `ld.so.1(1)`. The following illustrates how strings are concatenated to form the `LD_PRELOAD` environment variable in the new process image:

```
<current value of $LD_PRELOAD> + <space> +
libtnfprobe_path + "/libtnfprobe.so.1" +<space> +
ld_preload
```

This option is useful for preloading interposition libraries that have probes in them.

envp is an optional argument, and if set, it is used for the environment of the target program. It is a null-terminated array of pointers to null-terminated strings. These strings constitute the environment of the new process image. See execve(2). If *envp* is set, it overrides *ld_preload*. In this case, it is the caller's responsibility to ensure that `libtnfprobe.so.1` is loaded into the target program. If *envp* is not set, the new process image inherits the environment of the calling process, except for `LD_PRELOAD`.

The *ret_val* argument is the handle that can be used to control the process and the probes within the process. Upon return, the process is stopped before any user code, including `.init` sections, has been executed.

The `tnfctl_continue()` function is a blocking call and lets the target process referenced by *hndl* continue running. It can only be used on handles returned by `tnfctl_pid_open()` and `tnfctl_exec_open()` (direct process probe control). It returns when the target stops; the reason that the process stopped is returned in *evt*. This call is interruptible by signals. If it is interrupted, the process is stopped, and `TNFCTL_EVENT_EINTR` is returned in *evt*. The client of this library will have to decide which signal implies a stop to the target and catch that signal. Since a signal interrupts `tnfctl_continue()`, it will return, and the caller can decide whether or not to call `tnfctl_continue()` again.

`tnfctl_continue()` returns with an event of `TNFCTL_EVENT_DLOPEN`, `TNFCTL_EVENT_DLCLOSE`, `TNFCTL_EVENT_EXEC`, `TNFCTL_EVENT_FORK`, `TNFCTL_EVENT_EXIT`, or `TNFCTL_EVENT_TARGGONE`, respectively, when the target program calls dlopen(3C), dlclose(3C), any flavor of exec(2), fork(2) (or fork1(2)), exit(2), or terminates unexpectedly. If the target program called exec(2), the client then needs to call tnfctl_close(3TNF) on the current handle leaving the target resumed, suspended, or killed (second argument to tnfctl_close(3TNF)). No other `libtnfctl` interface call can be used on the existing handle. If the client wants to control the exec'ed image, it should leave the old handle suspended, and use `tnfctl_pid_open()` to reattach to the same process. This new handle can then be used to control the exec'ed image. See `EXAMPLES` below for sample code. If the target process did a fork(2) or fork1(2), and if control of the child process is not needed, then *child_hndl* should be `NULL`. If control of the child process is needed, then *child_hndl* should be set. If it is set, a pointer to a handle that can be used to control the child process is returned in *child_hndl*. The child process is stopped at the end of the `fork()` system call. See `EXAMPLES` for an example of this event.

Return Values The `tnfctl_pid_open()`, `tnfctl_exec_open()`, and `tnfctl_continue()` functions return `TNFCTL_ERR_NONE` upon success.

Errors The following error codes apply to `tnfctl_pid_open()`:

`TNFCTL_ERR_BADARG`	The *pid* specified is the same process. Use tnfctl_internal_open(3TNF) instead.
`TNFCTL_ERR_ACCES`	Permission denied. No privilege to connect to a setuid process.

TNFCTL_ERR_ALLOCFAIL	A memory allocation failure occurred.
TNFCTL_ERR_BUSY	Another client is already using /proc to control this process or internal tracing is being used.
TNFCTL_ERR_NOTDYNAMIC	The process is not a dynamic executable.
TNFCTL_ERR_NOPROCESS	No such target process exists.
TNFCTL_ERR_NOLIBTNFPROBE	libtnfprobe.so.1 is not linked in the target process.
TNFCTL_ERR_INTERNAL	An internal error occurred.

The following error codes apply to tnfctl_exec_open():

TNFCTL_ERR_ACCES	Permission denied.
TNFCTL_ERR_ALLOCFAIL	A memory allocation failure occurred.
TNFCTL_ERR_NOTDYNAMIC	The target is not a dynamic executable.
TNFCTL_ERR_NOLIBTNFPROBE	libtnfprobe.so.1 is not linked in the target process.
TNFCTL_ERR_FILENOTFOUND	The program is not found.
TNFCTL_ERR_INTERNAL	An internal error occurred.

The following error codes apply to tnfctl_continue():

TNFCTL_ERR_BADARG	Bad input argument. *hndl* is not a direct process probe control handle.
TNFCTL_ERR_INTERNAL	An internal error occurred.
TNFCTL_ERR_NOPROCESS	No such target process exists.

Examples

EXAMPLE 1 Using tnfctl_pid_open()

These examples do not include any error-handling code. Only the initial example includes the declaration of the variables that are used in all of the examples.

The following example shows how to preload libtnfprobe.so.1 from the normal location and inherit the parent's environment.

```
const char        *pgm;
char * const      *argv;
tnfctl_handle_t   *hndl, *new_hndl, *child_hndl;
tnfctl_errcode_t  err;
char * const      *envptr;
extern char       **environ;
tnfctl_event_t    evt;
int               pid;
```

EXAMPLE 1 Using `tnfctl_pid_open()` *(Continued)*

```
/* assuming argv has been allocated */
argv[0] = pgm;
/* set up rest of argument vector here */
err = tnfctl_exec_open(pgm, argv, NULL, NULL, NULL, &hndl);
```

This example shows how to preload two user-supplied libraries `libc_probe.so.1` and `libthread_probe.so.1`. They interpose on the corresponding `libc.so` and `libthread.so` interfaces and have probes for function entry and exit. `libtnfprobe.so.1` is preloaded from the normal location and the parent's environment is inherited.

```
/* assuming argv has been allocated */
argv[0] = pgm;
/* set up rest of argument vector here */
err = tnfctl_exec_open(pgm, argv, NULL, NULL,
      "libc_probe.so.1 libthread_probe.so.1", &hndl);
```

This example preloads an interposition library `libc_probe.so.1`, and specifies a different location from which to preload `libtnfprobe.so.1`.

```
/* assuming argv has been allocated */
argv[0] = pgm;
/* set up rest of argument vector here */
err = tnfctl_exec_open(pgm, argv, NULL, "/opt/SUNWXXX/lib",
      "libc_probe.so.1", &hndl);
```

To set up the environment explicitly for probe control to work, the target process must link `libtnfprobe.so.1`. If using *envp*, it is the caller's responsibility to do so.

```
/* assuming argv has been allocated */
argv[0] = pgm;
/* set up rest of argument vector here */
/* envptr set up to caller's needs */
err = tnfctl_exec_open(pgm, argv, envptr, NULL, NULL, &hndl);
```

Use this example to resume a process that does an exec(2) without controlling it.

```
err = tnfctl_continue(hndl, &evt, NULL);
switch (evt) {
case TNFCTL_EVENT_EXEC:
     /* let target process continue without control */
     err = tnfctl_close(hndl, TNFCTL_TARG_RESUME);
     ...
     break;
}
```

Alternatively, use the next example to control a process that does an exec(2).

EXAMPLE 1 Using `tnfctl_pid_open()` *(Continued)*

```
/*
 * assume the pid variable has been set by calling
 * tnfctl_trace_attrs_get()
*/
err = tnfctl_continue(hndl, &evt, NULL);
switch (evt) {
case TNFCTL_EVENT_EXEC:
     /* suspend the target process */
     err = tnfctl_close(hndl, TNFCTL_TARG_SUSPEND);
     /* re-open the exec'ed image */
     err = tnfctl_pid_open(pid, &new_hndl);
     /* new_hndl now controls the exec'ed image */
     ...
     break;
}
```

To let `fork`'ed children continue without control, use `NULL` as the last argument to `tnfctl_continue( )`.

```
err = tnfctl_continue(hndl, &evt, NULL);
```

The next example is how to control child processes that `fork(2)` or `fork1(2)` create.

```
err = tnfctl_continue(hndl, &evt, &child_hndl);
switch (evt) {
case TNFCTL_EVENT_FORK:
     /* spawn a new thread or process to control child_hndl */
     ...
     break;
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWtnfc
MT Level	MT-Safe

See Also

`ld(1)`, `prex(1)`, `proc(1)`, `exec(2)`, `execve(2)`, `exit(2)`, `fork(2)`, `TNF_PROBE(3TNF)`, `dlclose(3C)`, `dlopen(3C)`, `libtnfctl(3TNF)`, `tnfctl_close(3TNF)`, `tnfctl_internal_open(3TNF)`, `tracing(3TNF)` `attributes(5)`

Linker and Libraries Guide

Notes After a call to `tnfctl_continue()` returns, a client should use `tnfctl_trace_attrs_get(3TNF)` to check the `trace_buf_state` member of the trace attributes and make sure that there is no internal error in the target.

Name tnfctl_probe_apply, tnfctl_probe_apply_ids – iterate over probes

Synopsis

```
cc [ flag ... ] file ... -ltnfctl [ library ... ]
#include <tnf/tnfctl.h>

tnfctl_errcode_t tnfctl_probe_apply(tnfctl_handle_t *hndl,
     tnfctl_probe_op_t probe_op, void *clientdata);

tnfctl_errcode_t tnfctl_probe_apply_ids(tnfctl_handle_t *hndl,
     ulong_t probe_count, ulong_t *probe_ids,
     tnfctl_probe_op_t probe_op, void *clientdata);
```

Description `tnfctl_probe_apply()` is used to iterate over the probes controlled by *hndl*. For every probe, the *probe_op* function is called:

```
typedef tnfctl_errcode_t (*tnfctl_probe_op_t)(
    tnfctl_handle_t *hndl,
    tnfctl_probe_t *probe_hndl,
    void *clientdata);
```

Several predefined functions are available for use as *probe_op*. These functions are described in `tnfctl_probe_state_get`(3TNF).

The *clientdata* supplied in `tnfctl_probe_apply()` is passed in as the last argument of *probe_op*. The *probe_hndl* in the probe operation function can be used to query or change the state of the probe. See `tnfctl_probe_state_get`(3TNF). The *probe_op* function should return `TNFCTL_ERR_NONE` upon success. It can also return an error code, which will cause `tnfctl_probe_apply()` to stop processing the rest of the probes and return with the same error code. Note that there are five (5) error codes reserved that the client can use for its own semantics. See `ERRORS`.

The lifetime of *probe_hndl* is the same as the lifetime of *hndl*. It is good until *hndl* is closed by `tnfctl_close`(3TNF). Do not confuse a *probe_hndl* with *hndl*. The *probe_hndl* refers to a particular probe, while *hndl* refers to a process or the kernel. If *probe_hndl* is used in another `libtnfctl`(3TNF) interface, and it references a probe in a library that has been dynamically closed (see `dlclose`(3C)), then the error code `TNFCTL_ERR_INVALIDPROBE` will be returned by that interface.

`tnfctl_probe_apply_ids()` is very similar to `tnfctl_probe_apply()`. The difference is that *probe_op* is called only for probes that match a probe id specified in the array of integers referenced by *probe_ids*. The number of probe ids in the array should be specified in *probe_count*. Use `tnfctl_probe_state_get()` to get the *probe_id* that corresponds to the *probe_handl*.

Return Values `tnfctl_probe_apply()` and `tnfctl_probe_apply_ids()` return `TNFCTL_ERR_NONE` upon success.

Errors The following errors apply to both `tnfctl_probe_apply()` and `tnfctl_probe_apply_ids()`:

`TNFCTL_ERR_INTERNAL`	An internal error occurred.
`TNFCTL_ERR_USR1`	Error code reserved for user.
`TNFCTL_ERR_USR2`	Error code reserved for user.
`TNFCTL_ERR_USR3`	Error code reserved for user.
`TNFCTL_ERR_USR4`	Error code reserved for user.
`TNFCTL_ERR_USR5`	Error code reserved for user.

`tnfctl_probe_apply()` and `tnfctl_probe_apply_ids()` also return any error returned by the callback function *probe_op*.

The following errors apply only to `tnfctl_probe_apply_ids()`:

`TNFCTL_ERR_INVALIDPROBE`	The probe handle is no longer valid. For example, the probe is in a library that has been closed by `dlclose(3C)`.

Examples **EXAMPLE 1** Enabling Probes

To enable all probes:

```
tnfctl_probe_apply(hndl, tnfctl_probe_enable, NULL);
```

EXAMPLE 2 Disabling Probes

To disable the probes that match a certain pattern in the probe attribute string:

```
/* To disable all probes that contain the string "vm" */
tnfctl_probe_apply(hndl, select_disable, "vm");
static tnfctl_errcode_t
select_disable(tnfctl_handle_t *hndl, tnfctl_probe_t *probe_hndl,
void *client_data)
{
     char *pattern = client_data;
     tnfctl_probe_state_t probe_state;
     tnfctl_probe_state_get(hndl, probe_hndl, &probe_state);
     if (strstr(probe_state.attr_string, pattern)) {
          tnfctl_probe_disable(hndl, probe_hndl, NULL);
     }
}
```

Note that these examples do not have any error handling code.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWtnfc
MT-Level	MT-Safe

See Also prex(1), TNF_PROBE(3TNF), dlclose(3C), dlopen(3C), libtnfctl(3TNF), tnfctl_close(3TNF), tnfctl_probe_state_get(3TNF), tracing(3TNF), tnf_kernel_probes(4), attributes(5)

Linker and Libraries Guide

Name tnfctl_probe_state_get, tnfctl_probe_enable, tnfctl_probe_disable, tnfctl_probe_trace, tnfctl_probe_untrace, tnfctl_probe_connect, tnfctl_probe_disconnect_all – interfaces to query and to change the state of a probe

Synopsis

```
cc [ flag ... ] file ... -ltnfctl [ library ... ]
#include <tnf/tnfctl.h>

tnfctl_errcode_t tnfctl_probe_state_get(tnfctl_handle_t *hndl,
     tnfctl_probe_t *probe_hndl, tnfctl_probe_state_t *state);

tnfctl_errcode_t tnfctl_probe_enable(tnfctl_handle_t *hndl,
     tnfctl_probe_t *probe_hndl, void *ignored);

tnfctl_errcode_t tnfctl_probe_disable(tnfctl_handle_t *hndl,
     tnfctl_probe_t *probe_hndl, void *ignored);

tnfctl_errcode_t tnfctl_probe_trace(tnfctl_handle_t *hndl,
     tnfctl_probe_t *probe_hndl, void *ignored);

tnfctl_errcode_t tnfctl_probe_untrace(tnfctl_handle_t *hndl,
     tnfctl_probe_t *probe_hndl, void *ignored);

tnfctl_errcode_t tnfctl_probe_disconnect_all(tnfctl_handle_t *hndl,
     tnfctl_probe_t *probe_hndl, void *ignored);

tnfctl_errcode_t tnfctl_probe_connect(tnfctl_handle_t *hndl,
     tnfctl_probe_t *probe_hndl, const char *lib_base_name,
     const char *func_name);
```

Description `tnfctl_probe_state_get()` returns the state of the probe specified by *probe_hndl* in the process or kernel specified by *hndl*. The user will pass these in to an apply iterator. The caller must also allocate *state* and pass in a pointer to it. The semantics of the individual members of *state* are:

`id` The unique integer assigned to this probe. This number does not change over the lifetime of this probe. A *probe_hndl* can be obtained by using the calls `tnfctl_apply()`, `tanfctl_apply_ids()`, or `tnfctl_register_funcs()`.

`attr_string` A string that consists of *attribute value* pairs separated by semicolons. For the syntax of this string, see the syntax of the `detail` argument of the `TNF_PROBE`(3TNF) macro. The attributes *name*, *slots*, *keys*, `file`, and `line` are defined for every probe. Additional user-defined attributes can be added by using the *detail* argument of the `TNF_PROBE`(3TNF) macro. An example of *attr_string* follows:

```
"name pageout;slots vnode pages_pageout ;
keys vm pageio io;file vm.c;line 25;"
```

`enabled`	`B_TRUE` if the probe is enabled, or `B_FALSE` if the probe is disabled. Probes are disabled by default. Use `tnfctl_probe_enable()` or `tnfctl_probe_disable()` to change this state.
`traced`	`B_TRUE` if the probe is traced, or `B_FALSE` if the probe is not traced. Probes in user processes are traced by default. Kernel probes are untraced by default. Use `tnfctl_probe_trace()` or `tnfctl_probe_untrace()` to change this state.
`new_probe`	`B_TRUE` if this is a new probe brought in since the last change in libraries. See `dlopen`(3C) or `dlclose`(3C). Otherwise, the value of `new_probe` will be `B_FALSE`. This field is not meaningful for kernel probe control.
`obj_name`	The name of the shared object or executable in which the probe is located. This string can be freed, so the client should make a copy of the string if it needs to be saved for use by other `libtnfctl` interfaces. In kernel mode, this string is always `NULL`.
`func_names`	A null-terminated array of pointers to strings that contain the names of functions connected to this probe. Whenever an enabled probe is encountered at runtime, these functions are executed. This array also will be freed by the library when the state of the probe changes. Use `tnfctl_probe_connect()` or `tnfctl_probe_disconnect_all()` to change this state.
`func_addrs`	A null-terminated array of pointers to addresses of functions in the target image connected to this probe. This array also will be freed by the library when the state of the probe changes.
`client_registered_data`	Data that was registered by the client for this probe by the creator function in `tnfctl_register_funcs`(3TNF).

`tnfctl_probe_enable( )`, `tnfctl_probe_disable( )`, `tnfctl_probe_trace( )`, `tnfctl_probe_untrace( )`, and `tnfctl_probe_disconnect_all()` ignore the last argument. This convenient feature permits these functions to be used in the *probe_op* field of `tnfctl_probe_apply`(3TNF) and `tnfctl_probe_apply_ids`(3TNF).
`tnfctl_probe_enable()` enables the probe specified by *probe_hndl*. This is the master switch on a probe. A probe does not perform any action until it is enabled.

`tnfctl_probe_disable()` disables the probe specified by *probe_hndl*.

`tnfctl_probe_trace()` turns on tracing for the probe specified by *probe_hndl*. Probes emit a trace record only if the probe is traced.

`tnfctl_probe_untrace()` turns off tracing for the probe specified by *probe_hndl*. This is useful if you want to connect probe functions to a probe without tracing it.

`tnfctl_probe_connect()` connects the function *func_name* which exists in the library *lib_base_name*, to the probe specified by *probe_hndl*. `tnfctl_probe_connect()` returns an error code if used on a kernel tnfctl handle. *lib_base_name* is the base name (not a path) of the library. If it is `NULL`, and multiple functions in the target process match *func_name*, one of the matching functions is chosen arbitrarily. A probe function is a function that is in the target's address space and is written to a certain specification. The specification is not currently published.

`tnf_probe_debug()` is one function exported by `libtnfprobe.so.1` and is the debug function that `prex(1)` uses. When the debug function is executed, it prints out the probe arguments and the value of the `sunw%debug` attribute of the probe to `stderr`.

`tnfctl_probe_disconnect_all()` disconnects all probe functions from the probe specified by *probe_hndl*.

Note that no `libtnfctl` call returns a probe handle (`tnfctl_probe_t`), yet each of the routines described here takes a *probe_hndl* as an argument. These routines may be used by passing them to one of the `tnfctl_probe_apply(3TNF)` iterators as the "op" argument. Alternatively, probe handles may be obtained and saved by a user's "op" function, and they can be passed later as the *probe_hndl* argument when using any of the functions described here.

Return Values `tnfctl_probe_state_get( )`, `tnfctl_probe_enable( )`, `tnfctl_probe_disable( )`, `tnfctl_probe_trace( )`, `tnfctl_probe_untrace( )`, `tnfctl_probe_disconnect_all()` and `tnfctl_probe_connect()` return `TNFCTL_ERR_NONE` upon success.

Errors The following error codes apply to `tnfctl_probe_state_get()`:

`TNFCTL_ERR_INVALIDPROBE`	*probe_hndl* is no longer valid. The library that the probe was in could have been dynamically closed by `dlclose(3C)`.

The following error codes apply to `tnfctl_probe_enable()`, `tnfctl_probe_disable()`, `tnfctl_probe_trace()`, `tnfctl_probe_untrace()`, and `tnfctl_probe_disconnect_all()`

`TNFCTL_ERR_INVALIDPROBE`	*probe_hndl* is no longer valid. The library that the probe was in could have been dynamically closed by `dlclose(3C)`.
`TNFCTL_ERR_BUFBROKEN`	Cannot do probe operations because tracing is broken in the target.
`TNFCTL_ERR_NOBUF`	Cannot do probe operations until a buffer is allocated. See `tnfctl_buffer_alloc(3TNF)`. This error code does not apply to kernel probe control.

The following error codes apply to `tnfctl_probe_connect()`:

`TNFCTL_ERR_INVALIDPROBE`	*probe_hndl* is no longer valid. The library that the probe was in could have been dynamically closed by `dlclose(3C)`.
`TNFCTL_ERR_BADARG`	The handle is a kernel handle, or *func_name* could not be found.
`TNFCTL_ERR_BUFBROKEN`	Cannot do probe operations because tracing is broken in the target.
`TNFCTL_ERR_NOBUF`	Cannot do probe operations until a buffer is allocated. See `tnfctl_buffer_alloc(3TNF)`.

Attributes

See `attributes(5)` for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWtnfc
MT Level	MT-Safe

See Also

`prex(1)`, `TNF_PROBE(3TNF)`, `libtnfctl(3TNF)`, `tnfctl_check_libs(3TNF)`, `tnfctl_continue(3TNF)`, `tnfctl_probe_apply(3TNF)`, `tnfctl_probe_apply_ids(3TNF)`, `tracing(3TNF)`, `tnf_kernel_probes(4)`, `attributes(5)`

Name tnfctl_register_funcs – register callbacks for probe creation and destruction

Synopsis

```
cc [ flag ... ] file ... -ltnfctl [ library ... ]
#include <tnf/tnfctl.h>

tnfctl_errcode_t tnfctl_register_funcs(tnfctl_handle_t *hndl, void * (*create_func)
     (tnfctl_handle_t *, tnfctl_probe_t *), void (*destroy_func)(void *));
```

Description The function `tnfctl_register_funcs()` is used to store client-specific data on a per-probe basis. It registers a creator and a destructor function with *hndl*, either of which can be NULL. The creator function is called for every probe that currently exists in *hndl.* Every time a new probe is discovered, that is brought in by `dlopen(3C)`, *create_func* is called.

The return value of the creator function is stored as part of the probe state and can be retrieved by `tnfctl_probe_state_get(3TNF)` in the member field *client_registered_data*.

destroy_func is called for every probe handle that is freed. This does not necessarily happen at the time `dlclose(3C)` frees the shared object. The probe handles are freed only when *hndl* is closed by `tnfctl_close(3TNF)`. If `tnfctl_register_funcs()` is called a second time for the same *hndl,* then the previously registered destructor function is called first for all of the probes.

Return Values `tnfctl_register_funcs()` returns `TNFCTL_ERR_NONE` upon success.

Errors `TNFCTL_ERR_INTERNAL` An internal error occurred.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWtnfc
MT Level	MT-Safe

See Also `prex(1)`, `TNF_PROBE(3TNF)`, `dlclose(3C)`, `dlopen(3C)`, `libtnfctl(3TNF)`, `tnfctl_close(3TNF)`, `tnfctl_probe_state_get(3TNF)`, `tracing(3TNF)`, `tnf_kernel_probes(4)`, `attributes(5)`

Linker and Libraries Guide

Name tnfctl_strerror – map a tnfctl error code to a string

Synopsis

```
cc [ flag ... ] file ... -ltnfctl [ library ... ]
#include <tnf/tnfctl.h>

const char * tnfctl_strerror(tnfctl_errcode_t errcode);
```

Description `tnfctl_strerror()` maps the error number in *errcode* to an error message string, and it returns a pointer to that string. The returned string should not be overwritten or freed.

Errors `tnfctl_strerror()` returns the string "unknown libtnfctl.so error code" if the error number is not within the legal range.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWtnfc
MT Level	MT-Safe

See Also `prex(1)`, `TNF_PROBE(3TNF)`, `libtnfctl(3TNF)`, `tracing(3TNF)`, `attributes(5)`

Name tnfctl_trace_attrs_get – get the trace attributes from a tnfctl handle

Synopsis

```
cc [ flag... ] file... -ltnfctl [ library... ]
#include <tnf/tnfctl.h>

tnfctl_errcode_t tnfctl_trace_attrs_get(tnfctl_handle_t *hndl,
     tnfctl_trace_attrs_t *attrs);
```

Description The `tnfctl_trace_attrs_get()` function returns the trace attributes associated with *hndl* in *attrs*. The trace attributes can be changed by some of the other interfaces in `libtnfctl(3TNF)`. It is the client's responsibility to use `tnfctl_trace_attrs_get()` to get the new trace attributes after use of interfaces that change them. Typically, a client will use `tnfctl_trace_attrs_get()` after a call to `tnfctl_continue(3TNF)` in order to make sure that tracing is still working. See the discussion of `trace_buf_state` that follows.

Trace attributes are represented by the `struct tnfctl_trace_attrs` structure defined in `<tnf/tnfctl.h>`:

```
struct tnfctl_trace_attrs {
pid_t              targ_pid;          /* not kernel mode */
const char         *trace_file_name;  /* not kernel mode */
size_t             trace_buf_size;
size_t             trace_min_size;
tnfctl_bufstate_t  trace_buf_state;
boolean_t          trace_state;
boolean_t          filter_state;      /* kernel mode only */
long               pad;
};
```

The semantics of the individual members of *attrs* are:

`targ_pid` The process id of the target process. This is not valid for kernel tracing.

`trace_file_name` The name of the trace file to which the target writes. `trace_file_name` will be NULL if no trace file exists or if kernel tracing is implemented. This pointer should not be used after calling other `libtnfctl` interfaces. The client should copy this string if it should be saved for the use of other `libtnfctl` interfaces.

`trace_buf_size` The size of the trace buffer or file in bytes.

`trace_min_size` The minimum size in bytes of the trace buffer that can be allocated by using the `tnfctl_buffer_alloc(3TNF)` interface.

`trace_buf_state` The state of the trace buffer. `TNFCTL_BUF_OK` indicates that a trace buffer has been allocated. `TNFCTL_BUF_NONE` indicates that no buffer has been allocated. `TNFCTL_BUF_BROKEN` indicates that there is an internal error in the target for tracing. The target will continue to run correctly, but no trace records will be written. To fix tracing, restart the

process. For kernel tracing, deallocate the existing buffer with `tnfctl_buffer_dealloc(3TNF)` and allocate a new one with `tnfctl_buffer_alloc(3TNF)`.

`trace_state` The global tracing state of the target. Probes that are enabled will not write out data unless this state is on. This state is off by default for the kernel and can be changed by `tnfctl_trace_state_set(3TNF)`. For a process, this state is on by default and can only be changed by `tnf_process_disable(3TNF)` and `tnf_process_enable(3TNF)`.

`filter_state` The state of process filtering. For kernel probe control, it is possible to select a set of processes for which probes are enabled. See `tnfctl_filter_list_get(3TNF)`, `tnfctl_filter_list_add(3TNF)`, and `tnfctl_filter_list_delete(3TNF)`. No trace output will be written when other processes traverse these probe points. By default process filtering is off, and all processes cause the generation of trace records when they hit an enabled probe. Use `tnfctl_filter_state_set(3TNF)` to change the filter state.

Return Values The `tnfctl_trace_attrs_get()` function returns `TNFCTL_ERR_NONE` upon success.

Errors The `tnfctl_trace_attrs_get()` function will fail if:

`TNFCTL_ERR_INTERNAL` An internal error occurred.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWtnfc
MT Level	MT-Safe

See Also `prex(1)`, `TNF_PROBE(3TNF)`, `libtnfctl(3TNF)`, `tnfctl_buffer_alloc(3TNF)`, `tnfctl_continue(3TNF)`, `tnfctl_filter_list_get(3TNF)`, `tnf_process_disable(3TNF)`, `tracing(3TNF)`, `attributes(5)`

Name tnfctl_trace_state_set, tnfctl_filter_state_set, tnfctl_filter_list_get, tnfctl_filter_list_add, tnfctl_filter_list_delete – control kernel tracing and process filtering

Synopsis

```
cc [ flag ... ] file ... -ltnfctl [ library ... ]
#include <tnf/tnfctl.h>

tnfctl_errcode_t tnfctl_trace_state_set(tnfctl_handle_t *hndl,
     boolean_t trace_state);

tnfctl_errcode_t tnfctl_filter_state_set(tnfctl_handle_t *hndl,
     boolean_t filter_state);

tnfctl_errcode_t tnfctl_filter_list_get(tnfctl_handle_t *hndl,
     pid_t **pid_list, int *pid_count);

tnfctl_errcode_t tnfctl_filter_list_add(tnfctl_handle_t *hndl,
     pid_t pid_to_add);

tnfctl_errcode_t tnfctl_filter_list_delete(tnfctl_handle_t *hndl,
     pid_t pid_to_delete);
```

Description The interfaces to control kernel tracing and process filtering are used only with kernel handles, handles created by `tnfctl_kernel_open(3TNF)`. These interfaces are used to change the tracing and filter states for kernel tracing.

`tnfctl_trace_state_set()` sets the kernel global tracing state to "on" if *trace_state* is `B_TRUE`, or to "off" if *trace_state* is `B_FALSE`. For the kernel, *trace_state* is off by default.Probes that are enabled will not write out data unless this state is on. Use `tnfctl_trace_attrs_get(3TNF)` to retrieve the current tracing state.

`tnfctl_filter_state_set()` sets the kernel process filtering state to "on" if *filter_state* is `B_TRUE`, or to "off" if *filter_state* is `B_FALSE`. *filter_state* is off by default. If it is on, only probe points encountered by processes in the process filter set by `tnfctl_filter_list_add()` will generate trace points. Use `tnfctl_trace_attrs_get(3TNF)` to retrieve the current process filtering state.

`tnfctl_filter_list_get()` returns the process filter list as an array in *pid_list*. The count of elements in the process filter list is returned in *pid_count*. The caller should use `free(3C)` to free memory allocated for the array *pid_list*.

`tnfctl_filter_list_add()` adds *pid_to_add* to the process filter list. The process filter list is maintained even when the process filtering state is off, but it has no effect unless the process filtering state is on.

`tnfctl_filter_list_delete()` deletes *pid_to_delete* from the process filter list. It returns an error if the process does not exist or is not in the filter list.

Return Values The interfaces `tnfctl_trace_state_set()`, `tnfctl_filter_state_set()`, `tnfctl_filter_list_add()`, `tnfctl_filter_list_delete()`, and `tnfctl_filter_list_get()` return `TNFCTL_ERR_NONE` upon success.

Errors The following error codes apply to `tnfctl_trace_state_set`:

`TNFCTL_ERR_BADARG` The handle is not a kernel handle.

`TNFCTL_ERR_NOBUF` Cannot turn on tracing without a buffer being allocated.

`TNFCTL_ERR_BUFBROKEN` Tracing is broken in the target.

`TNFCTL_ERR_INTERNAL` An internal error occurred.

The following error codes apply to `tnfctl_filter_state_set`:

`TNFCTL_ERR_BADARG` The handle is not a kernel handle.

`TNFCTL_ERR_INTERNAL` An internal error occurred.

The following error codes apply to `tnfctl_filter_list_add`:

`TNFCTL_ERR_BADARG` The handle is not a kernel handle.

`TNFCTL_ERR_NOPROCESS` No such process exists.

`TNFCTL_ERR_ALLOCFAIL` A memory allocation failure occurred.

`TNFCTL_ERR_INTERNAL` An internal error occurred.

The following error codes apply to `tnfctl_filter_list_delete`:

`TNFCTL_ERR_BADARG` The handle is not a kernel handle.

`TNFCTL_ERR_NOPROCESS` No such process exists.

`TNFCTL_ERR_INTERNAL` An internal error occurred.

The following error codes apply to `tnfctl_filter_list_get`:

`TNFCTL_ERR_BADARG` The handle is not a kernel handle.

`TNFCTL_ERR_ALLOCFAIL` A memory allocation failure occurred.

`TNFCTL_ERR_INTERNAL` An internal error occurred.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWtnfc
MT Level	MT-Safe

See Also prex(1), TNF_PROBE(3TNF), free(3C), libtnfctl(3TNF), tnfctl_kernel_open(3TNF), tnfctl_trace_attrs_get(3TNF), tracing(3TNF), tnf_kernel_probes(4), attributes(5)

Name TNF_DECLARE_RECORD, TNF_DEFINE_RECORD_1, TNF_DEFINE_RECORD_2, TNF_DEFINE_RECORD_3, TNF_DEFINE_RECORD_4, TNF_DEFINE_RECORD_5 – TNF type extension interface for probes

Synopsis

```
cc [ flag ... ] file ...[ -ltnfprobe ] [ library ... ]
#include <tnf/probe.h>
```

```
TNF_DECLARE_RECORD(c_type, tnf_type);

TNF_DEFINE_RECORD_1(c_type, tnf_type, tnf_member_type_1, c_member_name_1);

TNF_DEFINE_RECORD_2(c_type, tnf_type, tnf_member_type_1, c_member_name_1,
     tnf_member_type_2, c_member_name_2);

TNF_DEFINE_RECORD_3(c_type, tnf_type, tnf_member_type_1, c_member_name_1,
     tnf_member_type_2, c_member_name_2, tnf_member_type_3,
     c_member_name_3);

TNF_DEFINE_RECORD_4(c_type, tnf_type, tnf_member_type_1, c_member_name_1,
     tnf_member_type_2, c_member_name_2, tnf_member_type_3,
     c_member_name_3, tnf_member_type_4, c_member_name_4);

TNF_DEFINE_RECORD_5(c_type, tnf_type, tnf_member_type_1, c_member_name_1,
     tnf_member_type_2, c_member_name_2, tnf_member_type_3,
     c_member_name_3,tnf_member_type_4, c_member_name_4,
     tnf_member_type_5, c_member_name_5);
```

Description This macro interface is used to extend the TNF (Trace Normal Form) types that can be used in TNF_PROBE(3TNF).

There should be only one TNF_DECLARE_RECORD and one TNF_DEFINE_RECORD per new type being defined. The TNF_DECLARE_RECORD should precede the TNF_DEFINE_RECORD. It can be in a header file that multiple source files share if those source files need to use the *tnf_type* being defined. The TNF_DEFINE_RECORD should only appear in one of the source files.

The TNF_DEFINE_RECORD macro interface defines a function as well as a couple of data structures. Hence, this interface has to be used in a source file (.c or .cc file) at file scope and not inside a function.

Note that there is no semicolon after the TNF_DEFINE_RECORD interface. Having one will generate a compiler warning.

Compiling with the preprocessor option -DNPROBE or with the preprocessor control statement #define NPROBE ahead of the #include <tnf/probe.h> statement, will stop the TNF type extension code from being compiled into the program.

The *c_type* argument must be a C struct type. It is the template from which the new *tnf_type* is being created. Not all elements of the C struct need be provided in the TNF type being defined.

The *tnf_type* argument is the name being given to the newly created type. Use of this interface uses the name space prefixed by *tnf_type*. If a new type called "xxx_type" is defined by a library, then the library should not use "xxx_type" as a prefix in any other symbols it defines. The policy on managing the type name space is the same as managing any other name space in a library; that is, prefix any new TNF types by the unique prefix that the rest of the symbols in the library use. This would prevent name space collisions when linking multiple libraries that define new TNF types. For example, if a library `libpalloc.so` uses the prefix "pal" for all symbols it defines, then it should also use the prefix "pal" for all new TNF types being defined.

The *tnf_member_type_n* argument is the TNF type of the *n*th provided member of the C structure.

The *tnf_member_name_n* argument is the name of the *n*th provided member of the C structure.

Examples

EXAMPLE 1 Defining and using a TNF type.

The following example demonstrates how a new TNF type is defined and used in a probe. This code is assumed to be part of a fictitious library called "libpalloc.so" which uses the prefix "pal" for all it's symbols.

```
#include <tnf/probe.h>
typedef struct pal_header {
        long    size;
        char *  descriptor;
        struct pal_header *next;
} pal_header_t;
TNF_DECLARE_RECORD(pal_header_t, pal_tnf_header);
TNF_DEFINE_RECORD_2(pal_header_t, pal_tnf_header,
                        tnf_long,   size,
                        tnf_string, descriptor)
/*
 * Note: name space prefixed by pal_tnf_header should not
 *       be used by this client anymore.
 */
void
pal_free(pal_header_t *header_p)
{
        int state;
        TNF_PROBE_2(pal_free_start, "palloc pal_free",
                "sunw%debug entering pal_free",
                tnf_long,       state_var,  state,
                pal_tnf_header, header_var, header_p);
        . . .
}
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWtnfd
MT-Level	MT-Safe

See Also prex(1), tnfdump(1), TNF_PROBE(3TNF), tnf_process_disable(3TNF), attributes(5)

Notes It is possible to make a *tnf_type* definition be recursive or mutually recursive e.g. a structure that uses the "next" field to point to itself (a linked list). If such a structure is sent in to a TNF_PROBE(3TNF), then the entire linked list will be logged to the trace file (until the "next" field is NULL). But, if the list is circular, it will result in an infinite loop. To break the recursion, either do not include the "next" field in the *tnf_type*, or define the type of the "next" member as tnf_opaque.

Name TNF_PROBE, TNF_PROBE_0, TNF_PROBE_1, TNF_PROBE_2, TNF_PROBE_3, TNF_PROBE_4, TNF_PROBE_5, TNF_PROBE_0_DEBUG, TNF_PROBE_1_DEBUG, TNF_PROBE_2_DEBUG, TNF_PROBE_3_DEBUG, TNF_PROBE_4_DEBUG, TNF_PROBE_5_DEBUG, TNF_DEBUG – probe insertion interface

Synopsis

```
cc [ flag ... ] [ -DTNF_DEBUG ] file ... [ -ltnfprobe ] [ library ... ]
#include <tnf/probe.h>

TNF_PROBE_0(name, keys, detail);

TNF_PROBE_1(name, keys, detail, arg_type_1, arg_name_1, arg_value_1);

TNF_PROBE_2(name, keys, detail, arg_type_1, arg_name_1, arg_value_1,
     arg_type_2, arg_name_2, arg_value_2);

TNF_PROBE_3(name, keys, detail, arg_type_1, arg_name_1,arg_value_1,
     arg_type_2, arg_name_2, arg_value_2,
     arg_type_3, arg_name_3, arg_value_3);

TNF_PROBE_4(name, keys, detail, arg_type_1, arg_name_1, arg_value_1,
     arg_type_2, arg_name_2, arg_value_2,
     arg_type_3, arg_name_3, arg_value_3,
     arg_type_4, arg_name_4, arg_value_4);

TNF_PROBE_5(name, keys, detail, arg_type_1, arg_name_1, arg_value_1,
     arg_type_2, arg_name_2, arg_value_2,
     arg_type_3, arg_name_3, arg_value_3,
     arg_type_4, arg_name_4, arg_value_4,
     arg_type_5, arg_name_5, arg_value_5);

TNF_PROBE_0_DEBUG(name, keys, detail);

TNF_PROBE_1_DEBUG(name, keys, detail, arg_type_1, arg_name_1, arg_value_1);

TNF_PROBE_2_DEBUG(name, keys, detail, arg_type_1, arg_name_1, arg_value_1,
     arg_type_2, arg_name_2, arg_value_2);

TNF_PROBE_3_DEBUG(name, keys, detail, arg_type_1, arg_name_1, arg_value_1,
     arg_type_2, arg_name_2, arg_value_2,
     arg_type_3, arg_name_3, arg_value_3);

TNF_PROBE_4_DEBUG(name, keys, detail, arg_type_1, arg_name_1, arg_value_1,
     arg_type_2, arg_name_2, arg_value_2,
     arg_type_3, arg_name_3, arg_value_3,
     arg_type_4, arg_name_4, arg_value_4);

TNF_PROBE_5_DEBUG(name, keys, detail, arg_type_1, arg_name_1, arg_value_1,
     arg_type_2, arg_name_2, arg_value_2,
     arg_type_3, arg_name_3, arg_value_3,
     arg_type_4, arg_name_4, arg_value_4,
     arg_type_5, arg_name_5, arg_value_5);
```

Description This macro interface is used to insert probes into C or C++ code for tracing. See `tracing(3TNF)` for a discussion of the Solaris tracing architecture, including example source code that uses it.

You can place probes anywhere in C and C++ programs including .init sections, .fini sections, multi-threaded code, shared objects, and shared objects opened by `dlopen(3C)`. Use probes to generate trace data for performance analysis or to write debugging output to stderr. Probes are controlled at runtime by `prex(1)`.

The trace data is logged to a trace file in Trace Normal Form (TNF). The interface for the user to specify the name and size of the trace file is described in `prex(1)`. Think of the trace file as the least recently used circular buffer. Once the file has been filled, newer events will overwrite the older ones.

Use `TNF_PROBE_0` through `TNF_PROBE_5` to create production probes. These probes are compiled in by default. Developers are encouraged to embed such probes strategically, and to leave them compiled within production software. Such probes facilitate on-site analysis of the software.

Use `TNF_PROBE_0_DEBUG` through `TNF_PROBE_5_DEBUG` to create debug probes. These probes are compiled out by default. If you compile the program with the preprocessor option `-DTNF_DEBUG` or with the preprocessor control statement `#define TNF_DEBUG` ahead of the `#include <tnf/probe.h>` statement, the debug probes will be compiled into the program. When compiled in, debug probes differ in only one way from the equivalent production probes. They contain an additional “debug” attribute which may be used to distinguish them from production probes at runtime, for example, when using `prex()`. Developers are encouraged to embed any number of probes for debugging purposes. Disabled probes have such a small runtime overhead that even large numbers of them do not make a significant impact.

If you compile with the preprocessor option `-DNPROBE` or place the preprocessor control statement `#define NPROBE` ahead of the `#include <tnf/probe.h>` statement, no probes will be compiled into the program.

name The *name* of the probe should follow the syntax guidelines for identifiers in ANSI C. The use of *name* declares it, hence no separate declaration is necessary. This is a block scope declaration, so it does not affect the name space of the program.

keys *keys* is a string of space-separated keywords that specify the groups that the probe belongs to. Semicolons, single quotation marks, and the equal character (=) are not allowed in this string. If any of the groups are enabled, the probe is enabled. *keys* cannot be a variable. It must be a string constant.

detail *detail* is a string that consists of <attribute> <value> pairs that are each separated by a semicolon. The first word (up to the space) is considered to be the attribute and the rest of the string (up to the semicolon) is considered the value. Single quotation marks are used to denote

a string value. Besides quotation marks, spaces separate multiple values. The value is optional. Although semicolons or single quotation marks generally are not allowed within either the attribute or the value, when text with embedded spaces is meant to denote a single value, use single quotes surrounding this text.

Use *detail* for one of two reasons. First, use *detail* to supply an attribute that a user can type into `prex(1)` to select probes. For example, if a user defines an attribute called color, then `prex(1)` can select probes based on the value of color. Second, use *detail* to annotate a probe with a string that is written out to a trace file only once. `prex(1)` uses spaces to tokenize the value when searching for a match. Spaces around the semicolon delimiter are allowed. *detail* cannot be a variable; it must be a string constant. For example, the *detail* string:

```
"XYZ%debug 'entering function A'; XYZ%exception 'no file';
XYZ%func_entry; XYZ%color red blue"
```

consists of 4 units:

Attribute	Value	Values that prex matches on
XYZ%debug	'entering function A'	'entering function A'
XYZ%exception	'no file'	'no file'
XYZ%func_entry	/.*/	(regular expression)
XYZ%color	red blue	red <or> blue

Attribute names must be prefixed by the vendor stock symbol followed by the '%' character. This avoids conflicts in the attribute name space. All attributes that do not have a '%' character are reserved. The following attributes are predefined:

Attribute	Semantics
name	name of probe
keys	keys of the probe (value is space– separated tokens)
file	file name of the probe
line	line number of the probe
slots	slot names of the probe event (*arg_name_n*)
object	the executable or shared object that this probe is in.
debug	distinguishes debug probes from production probes

arg_type_n This is the type of the *n*th argument. The following are predefined TNF types:

tnf Type	Associated C type (and semantics)
tnf_int	int
tnf_uint	unsigned int
tnf_long	long
tnf_ulong	unsigned long
tnf_longlong	long long (if implemented in compilation system)
tnf_ulonglong	unsigned long long (if implemented in compilation system)
tnf_float	float
tnf_double	double
tnf_string	char *
tnf_opaque	void *

To define new TNF types that are records consisting of the predefined TNF types or references to other user defined types, use the interface specified in TNF_DECLARE_RECORD(3TNF).

arg_name_n *arg_name_n* is the name that the user associates with the *n*th argument. Do not place quotation marks around *arg_name_n*. Follow the syntax guidelines for identifiers in ANSI C. The string version of *arg_name_n* is stored for every probe and can be accessed as the attribute "slots".

arg_value_n *arg_value_n* is evaluated to yield a value to be included in the trace file. A read access is done on any variables that are in mentioned in *arg_value_n*. In a multithreaded program, it is the user's responsibility to place locks around the TNF_PROBE macro if *arg_value_n* contains a variable that should be read protected.

Examples

EXAMPLE 1 tracing(3TNF)

See tracing(3TNF) for complete examples showing debug and production probes in source code.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWtnfd

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	MT-Safe

See Also ld(1), prex(1), tnfdump(1), dlopen(3C), libtnfctl(3TNF), TNF_DECLARE_RECORD(3TNF), threads(5), tnf_process_disable(3TNF), tracing(3TNF), attributes(5)

Notes If attaching to a running program with prex(1) to control the probes, compile the program with `-ltnfprobe` or start the program with the environment variable `LD_PRELOAD` set to `libtnfprobe.so.1`. See ld(1). If `libtnfprobe` is explicitly linked into the program, it must be listed before `libdoor`, which in turn must be listed before `libthread` on the link line.

Name tnf_process_disable, tnf_process_enable, tnf_thread_disable, tnf_thread_enable – probe control internal interface

Synopsis

```
cc [ flag ... ] file ... -ltnfprobe [ library ... ]
#include <tnf/probe.h>

void tnf_process_disable(void);

void tnf_process_enable(void);

void tnf_thread_disable(void);

void tnf_thread_enable(void);
```

Description There are three levels of granularity for controlling tracing and probe functions (called probing from here on): probing for the entire process, a particular thread, and the probe itself can be disabled or enabled. The first two (process and thread) are controlled by this interface. The probe is controlled with the `prex(1)` utility.

The `tnf_process_disable()` function turns off probing for the process. The default process state is to have probing enabled. The `tnf_process_enable()` function turns on probing for the process.

The `tnf_thread_disable()` function turns off probing for the currently running thread. Threads are "born" or created with this state enabled. The `tnf_thread_enable()` function turns on probing for the currently running thread. If the program is a non-threaded program, these two thread interfaces disable or enable probing for the process.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWtnfd
Interface Stability	Unstable
MT-Level	MT-Safe

See Also `prex(1)`, `tnfdump(1)`, `TNF_DECLARE_RECORD(3TNF)`, `TNF_PROBE(3TNF)`, `attributes(5)`

Notes A probe is considered enabled only if:

- `prex(1)` has enabled the probe AND
- the process has probing enabled, which is the default or could be set with `tnf_process_enable()` AND
- the thread that hits the probe has probing enabled, which is every thread's default or could be set with `tnf_thread_enable()`.

There is a run time cost associated with determining that the probe is disabled. To reduce the performance effect of probes, this cost should be minimized. The quickest way that a probe can be determined to be disabled is by the enable control that `prex(1)` uses. Therefore, to disable all the probes in a process use the `disable` command in `prex(1)` rather than `tnf_process_disable()`.

The `tnf_process_disable()` and `tnf_process_enable()` functions should only be used to toggle probing based on some internal program condition. The `tnf_thread_disable()` function should be used to turn off probing for threads that are uninteresting.

Name tracing – overview of tnf tracing system

Description `tnf tracing` is a set of programs and API's that can be used to present a high-level view of the performance of an executable, a library, or part of the kernel. `tracing` is used to analyze a program's performance and identify the conditions that produced a bug.

The core elements of `tracing` are:

`TNF_PROBE_*( )`	The `TNF_PROBE_*( )` macros define "probes" to be placed in code which, when enabled and executed, cause information to be added to a trace file. See `TNF_PROBE(3TNF)`. If there are insufficient `TNF_PROBE_*` macros to store all the data of interest for a probe, data may be grouped into records. See `TNF_DECLARE_RECORD(3TNF)`.
`prex`	Displays and controls probes in running software. See prex(1).
`kernel probes`	A set of probes built into the Solaris kernel which capture information about system calls, multithreading, page faults, swapping, memory management, and I/O. You can use these probes to obtain detailed traces of kernel activity under your application workloads. See `tnf_kernel_probes(4)`.
`tnfxtract`	A program that extracts the trace data from the kernel's in-memory buffer into a file. See `tnfxtract(1)`.
`tnfdump`	A program that displays the information from a trace file. See `tnfdump(1)`.
`libtnfctl`	A library of interfaces that controls probes in a process. See `libtnfctl(3TNF)`. `prex(1)` also utilizes this library. Other tools and processes use the `libtnfctl` interfaces to exercise fine control over their own probes.
`tnf_process_enable()`	A routine called by a process to turn on tracing and probe functions for the current process. See `tnf_process_enable(3TNF)`.
`tnf_process_disable()`	A routine called by a process to turn off tracing and probe functions for the current process. See `tnf_process_disable(3TNF)`.
`tnf_thread_enable()`	A routine called by a process to turn on tracing and probe functions for the currently running thread. See `tnf_thread_enable(3TNF)`.
`tnf_thread_disable()`	A routine called by a process to turn off tracing and probe functions for the currently running thread. See `tnf_thread_disable(3TNF)`.

Examples

EXAMPLE 1 Tracing a Process

The following function in some daemon process accepts job requests of various types, queueing them for later execution. There are two "debug probes" and one "production probe." Note that probes which are intended for debugging will not be compiled into the final version of the code; however, production probes are compiled into the final product.

```
 /*
  * To compile in all probes (for development):
  *    cc -DTNF_DEBUG ...
  *
  * To compile in only production probes (for release):
  *    cc ...
  *
  * To compile in no probes at all:
  *    cc -DNPROBE ...
  */
#include <tnf/probe.h>
void work(long, char *);
enum work_request_type { READ, WRITE, ERASE, UPDATE };
static char *work_request_name[] = {"read", "write", "erase", "update"};
main( )
{
  long i;
  for (i = READ;  i <= UPDATE; i++)
    work(i, work_request_name[i]);
}
void work(long request_type, char *request_name)
{
    static long q_length;
    TNF_PROBE_2_DEBUG(work_start, "work",
        "XYZ%debug 'in function work'",
        tnf_long, request_type_arg, request_type,
        tnf_string, request_name_arg, request_name);
    /* assume work request is queued for later processing */
    q_length++;
    TNF_PROBE_1(work_queue, "work queue",
        "XYZ%work_load heavy",
        tnf_long, queue_length, q_length);
    TNF_PROBE_0_DEBUG(work_end, "work", "");
}
```

The production probe "work_queue," which remains compiled in the code, will, when enabled, log the length of the work queue each time a request is received.

The debug probes "work_start" and "work_end, " which are compiled only during the development phase, track entry to and exit from the `work()` function and measure how much time is spent executing it. Additionally, the debug probe "work_start" logs the value of the two

EXAMPLE 1 Tracing a Process *(Continued)*

incoming arguments `request_type` and `request_name`. The runtime overhead for disabled probes is low enough that one can liberally embed them in the code with little impact on performance.

For debugging, the developer would compile with `-DTNF_DEBUG`, run the program under control of `prex(1)`, enable the probes of interest (in this case, all probes), continue the program until exit, and dump the trace file:

```
% cc
-DTNF_DEBUG -o daemon daemon.c  # compile in all probes
% prex daemon                  # run program under prex control
Target process stopped
Type "continue" to resume the target, "help" for help ...
prex> list probes $all         # list all probes in program
<probe list output here>
prex> enable $all              # enable all probes
prex> continue                 # let target process execute
<program output here>
prex: target process finished
% ls /tmp/trace-*              # trace output is in trace-<pid>
/tmp/trace-4194
% tnfdump /tmp/trace-4194      # get ascii output of trace file
<trace records output here>
```

For the production version of the system, the developer simply compiles without –`DTNF_DEBUG`.

EXAMPLE 2 Tracing the Kernel

Kernel tracing is similar to tracing a process; however, there are some differences. For instance, to trace the kernel, you need superuser privileges. The following example uses prex(1) and traces the probes in the kernel that capture system call information.

```
Allocate kernel
trace buffer and capture trace data:
root# prex -k
Type "help" for help ...
prex> buffer alloc 2m          # allocate kernel trace buffer
Buffer of size 2097152 bytes allocated
prex> list probes $all         # list all kernel probes
<probe list output here>
prex> list probes syscall      # list syscall probes
                               # (keys=syscall)
<syscall probes list output here>
prex> enable syscall           # enable only syscall probes
```

EXAMPLE 2 Tracing the Kernel *(Continued)*

```
prex> ktrace on                 # turn on kernel tracing
<Run your application in another window at this point>
prex> ktrace off                # turn off kernel tracing
prex> quit                      # exit prex
Extract the kernel's trace buffer into a file:
root# tnfxtract /tmp/ktrace     # extract kernel trace buffer
Reset kernel tracing:
root# prex -k
prex> disable $all              # disable all probes
prex> untrace $all              # untrace all probes
prex> buffer dealloc            # deallocate kernel trace buffer
prex> quit
```

CAUTION: Do not deallocate the trace buffer until you have extracted it into a trace file. Otherwise, you will lose the trace data that you collected from your experiment!

Examine the kernel trace file:

```
root# tnfdump /tmp/ktrace       # get ascii dump of trace file
<trace records output here>
```

`prex` can also attach to a running process, list probes, and perform a variety of other tasks.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWtnfd
MT Level	MT-Safe

See Also

`prex(1)`, `tnfdump(1)`, `tnfxtract(1)`, `TNF_DECLARE_RECORD(3TNF)`, `TNF_PROBE(3TNF)`, `libtnfctl(3TNF)`, `tnf_process_disable(3TNF)`, `tnf_kernel_probes(4)`, `attributes(5)`

Name tsol_getrhtype – get trusted network host type

Synopsis

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

#include <libtsnet.h>

tsol_host_type_t tsol_getrhtype(char *hostname);
```

Description The `tsol_getrhtype()` function queries the kernel-level network information to determine the host type that is associated with the specified *hostname*. The *hostname* can be a regular hostname, an IP address, or a network wildcard address.

Return Values The returned value will be one of the enumerated types that is defined in the `tsol_host_type_t` typedef. Currently these types are `UNLABELED` and `SUN_CIPSO`.

Files `/etc/security/tsol/tnrhdb` Trusted network remote-host database

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also `libtsnet(3LIB)`, `attributes(5)`

"Obtaining the Remote Host Type" in *Solaris Trusted Extensions Developer's Guide*

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name uuid_clear, uuid_compare, uuid_copy, uuid_generate, uuid_generate_random, uuid_generate_time, uuid_is_null, uuid_parse, uuid_time, uuid_unparse – universally unique identifier (UUID) operations

Synopsis

```
cc [ flag ... ] file... -luuid [ library ... ]
#include <uuid/uuid.h>

void uuid_clear(uuid_t uu);

int uuid_compare(uuid_t uu1, uuid_t uu2);

void uuid_copy(uuid_t dst, uuid_t src);

void uuid_generate(uuid_t out);

void uuid_generate_random(uuid_t out);

void uuid_generate_time(uuid_t out);

int uuid_is_null(uuid_t uu);

int uuid_parse(char *in, uuid_t uu);

time_t uuid_time(uuid_t uu, struct timeval *ret_tv);

void uuid_unparse(uuid_t uu, char *out);
```

Description The `uuid_clear()` function sets the value of the specified universally unique identifier (UUID) variable *uu* to the `NULL` value.

The `uuid_compare()` function compares the two specified UUID variables *uu1* and *uu2* to each other. It returns an integer less than, equal to, or greater than zero if *uu1* is found to be, respectively, lexicographically less than, equal, or greater than *uu2*.

The `uuid_copy()` function copies the UUID variable *src* to *dst*.

The `uuid_generate()` function creates a new UUID that is generated based on high-quality randomness from `/dev/urandom`, if available. If `/dev/urandom` is not available, `uuid_generate()` calls `uuid_generate_time()`. Because the use of this algorithm provides information about when and where the UUID was generated, it could cause privacy problems for some applications.

The `uuid_generate_random()` function produces a UUID with a random or pseudo-randomly generated time and Ethernet MAC address that corresponds to a DCE version 4 UUID.

The `uuid_generate_time()` function uses the current time and the local Ethernet MAC address (if available, otherwise a MAC address is fabricated) that corresponds to a DCE version 1 UUID. If the UUID is not guaranteed to be unique, the multicast bit is set (the high-order bit of octet number 10).

The `uuid_is_null()` function compares the value of the specified UUID variable *uu* to the `NULL` value. If the value is equal to the `NULL` UUID, 1 is returned. Otherwise 0 is returned.

The `uuid_parse()` function converts the UUID string specified by *in* to the internal `uuid_t` format. The input UUID is a string of the form `cefa7a9c-1dd2-11b2-8350-880020adbeef`. In `printf(3C)` format, the string is "`%08x-%04x-%04x-%04x-%012x`", 36 bytes plus the trailing null character. If the input string is parsed successfully, `0` is returned and the UUID is stored in the location pointed to by *uu*. Otherwise `-1` is returned.

The `uuid_time()` function extracts the time at which the specified UUID *uu* was created. Since the UUID creation time is encoded within the UUID, this function can reasonably be expected to extract the creation time only for UUIDs created with the `uuid_generate_time()` function. The time at which the UUID was created, in seconds since January 1, 1970 GMT (the epoch), is returned (see `time(2)`). The time at which the UUID was created, in seconds and microseconds since the epoch is also stored in the location pointed to by `ret_tv` (see `gettimeofday(3C)`).

The `uuid_unparse()` function converts the specified UUID *uu* from the internal binary format to a string of the length defined in the `uuid.h` macro, `UUID_PRINTABLE_STRING_LENGTH`, which includes the trailing null character. The resulting value is stored in the character string pointed to by *out*.

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also

`inetd(1M)`, `time(2)`, `gettimeofday(3C)`, `libuuid(3LIB)`, `printf(3C)`, `attributes(5)`

Name volmgt_acquire – reserve removable media device

Synopsis

```
cc [ flag ... ] file ... -lvolmgt [ library ... ]
#include <sys/types.h>

#include <volmgt.h>


int volmgt_acquire(char *dev, char *id, int ovr, char **err, pid_t *pidp);
```

Description This function is obsolete. The management of removable media by the Volume Management feature, including `vold`, has been replaced by software that supports the Hardware Abstraction Layer (HAL). Programmatic support for HAL is through the HAL APIs, which are documented on the HAL web site. See `hal(5)`. The return value of this function is undefined.

The `volmgt_acquire()` routine reserves the removable media device specified as *dev*. `volmgt_acquire()` operates in two different modes, depending on whether or not volume management is running.

If volume management *is* running, `volmgt_acquire()` attempts to reserve the removable media device specified as *dev*. Specify *dev* as *either* a symbolic device name (for example, `floppy0`) or a physical device pathname (for example, `/dsk/unnamed_floppy`).

If volume management *is not* running, `volmgt_acquire()` requires callers to specify a physical device pathname for *dev*. Specifying *dev* as a symbolic device name is *not* acceptable. In this mode, `volmgt_acquire()` relies entirely on the major and minor numbers of the device to determine whether or not the device is reserved.

If *dev* is free, `volmgt_acquire()` updates the internal device reservation database with the caller's process id (*pid*) and the specified `id` string.

If *dev* is reserved by another process, the reservation attempt fails and `volmgt_acquire( ):`

- sets `errno` to `EBUSY`
- fills the caller's `id` value in the array pointed to by *err*
- fills in the *pid* to which the pointer *pidp* points with the *pid* of the process which holds the reservation, if the supplied *pidp* is non-zero

If the override *ovr* is non-zero, the call overrides the device reservation.

Return Values The return from this function is undefined.

Errors The `volmgt_acquire()` routine fails if one or more of the following are true:

`EINVAL` One of the specified arguments is invalid or missing.

`EBUSY` *dev* is already reserved by another process (and *ovr* was not set to a non-zero value)

Examples

EXAMPLE 1 Using `volmgt_acquire()`

In the following example, volume management is running and the first floppy drive is reserved, accessed and released.

```
#include <volmgt.h>
char *errp;
if (!volmgt_acquire("floppy0", "FileMgr", 0, NULL,
    &errp, NULL)) {
        /* handle error case */
        . . .
}
/* floppy acquired - now access it */
if (!volmgt_release("floppy0")) {
        /* handle error case */
         . . .
}
```

EXAMPLE 2 Using `volmgt_acquire()` To Override A Lock On Another Process

The following example shows how callers can override a lock on another process using `volmgt_acquire()`.

```
char *errp, buf[20];
int override = 0;
pid_t pid;
if (!volmgt_acquire("floppy0", "FileMgr", 0, &errp,
    &pid)) {
      if (errno == EBUSY) {
             (void) printf("override %s (pid=%ld)?\n",
                errp, pid); {
             (void) fgets(buf, 20, stdin);
             if (buf[0] == 'y') {
                   override++;
             }
       } else {
             /* handle other errors */
             . . .
       }
}
if (override) {
      if (!volmgt_acquire("floppy0", "FileMgr", 1,
          &errp, NULL)) {
             /* really give up this time! */
              . . .
      }
}
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe
Interface Stability	Obsolete

See Also free(3C), malloc(3C), volmgt_release(3VOLMGT), attributes(5), hal(5)

Notes When returning a string through *err*, `volmgt_acquire()` allocates a memory area using malloc(3C). Use free(3C) to release the memory area when no longer needed.

The *ovr* argument is intended to allow callers to override the current device reservation. It is assumed that the calling application has determined that the current reservation can safely be cleared. See `EXAMPLES`.

Name volmgt_check – have Volume Management check for media

Synopsis

```
cc [ flag... ] file... -lvolmgt [ library ... ]
#include <volmgt.h>

int volmgt_check(char *pathname);
```

Description This function is obsolete. The management of removable media by the Volume Management feature, including `vold`, has been replaced by software that supports the Hardware Abstraction Layer (HAL). Programmatic support for HAL is through the HAL APIs, which are documented on the HAL web site. See `hal(5)`. The return value of this function is undefined.

This routine asks volume Management to check the specified *pathname* and determine if new media has been inserted in that drive.

If a null pointer is passed in, then Volume Management will check each device it is managing that can be checked.

If new media is found, `volmgt_check()` tells volume management to initiate appropriate actions.

Return Values The return from this function is undefined.

Errors This routine can fail, returning `0`, if a `stat(2)` or `open(2)` of the supplied *pathname* fails, or if any of the following is true:

`ENXIO` volume management is not running.

`EINTR` An interrupt signal was detected while checking for media.

Examples **EXAMPLE 1** Checking If Any New Media Is Inserted

To check if any drive managed by volume management has any new media inserted in it:

```
if (volmgt_check(NULL)) {
        (void) printf("Volume management found media\n");
}
```

This would also request volume management to take whatever action was appropriate for the new media.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `volcheck(1)`, `open(2)`, `stat(2)`, `volmgt_inuse(3VOLMGT)`, `volmgt_running(3VOLMGT)`, `attributes(5)`, `hal(5)`

Notes Since `volmgt_check()` returns `0` for two different cases (both when no media is found, and when an error occurs), it is up to the user to check *errno* to differentiate the two, and to ensure that volume management is running.

Name volmgt_feature_enabled – check whether specific Volume Management features are enabled

Synopsis

```
cc [ flag ... ] file ... -l volmgt [ library ... ]
#include <volmgt.h>

int volmgt_feature_enabled(char *feat_str);
```

Description This function is obsolete. The management of removable media by the Volume Management feature, including `vold`, has been replaced by software that supports the Hardware Abstraction Layer (HAL). Programmatic support for HAL is through the HAL APIs, which are documented on the HAL web site. See `hal(5)`. The return value of this function is undefined.

The `volmgt_feature_enabled()` routine checks whether specific volume management features are enabled. `volmgt_feature_enabled()` checks for the volume management features passed in to it by the *feat_str* parameter.

Currently, the only supported feature string that `volmgt_feature_enabled()` checks for is `floppy-summit-interfaces`. The `floppy-summit-interfaces` feature string checks for the presence of the `libvolmgt` routines `volmgt_acquire()` and `volmgt_release()`.

Return Values The return from this function is undefined.

Examples **EXAMPLE 1** A sample of the `volmgt_feature_enabled()` function.

In the following example, `volmgt_feature_enabled()` checks whether the `floppy-summit-interfaces` feature is enabled.

```
if (volmgt_feature_enabled("floppy-summit-interfaces")) {
        (void) printf("Media Sharing Routines ARE present\n");
} else {
        (void) printf("Media Sharing Routines are NOT present\n");
}
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe
Interface Stability	Obsolete

See Also `volmgt_acquire(3VOLMGT)`, `volmgt_release(3VOLMGT)`, `attributes(5)`, `hal(5)`

Name volmgt_inuse – check whether or not volume management is managing a pathname

Synopsis

```
cc [ flag... ] file... -lvolmgt [ library ... ]
#include <volmgt.h>

int volmgt_inuse(char *pathname);
```

Description This function is obsolete. The management of removable media by the Volume Management feature, including `vold`, has been replaced by software that supports the Hardware Abstraction Layer (HAL). Programmatic support for HAL is through the HAL APIs, which are documented on the HAL web site. See `hal(5)`. The return value of this function is undefined.

`volmgt_inuse()` checks whether volume management is managing the specified *pathname*.

Return Values The return from this function is undefined.

Errors This routine can fail, returning `0`, if a `stat(2)` of the supplied *pathname* or an `open(2)` of `/dev/volctl` fails, or if any of the following is true:

`ENXIO` Volume management is not running.

`EINTR` An interrupt signal was detected while checking for the supplied *pathname* for use.

Examples **EXAMPLE 1** Using `volmgt_inuse()`

To see if volume management is managing the first floppy disk:

```
if (volmgt_inuse("/dev/rdiskette0") != 0) {
        (void) printf("volmgt is managing diskette 0\n");
} else {
        (void) printf("volmgt is NOT managing diskette 0\n");
}
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe
Interface Stability	Obsolete

See Also `open(2)`, `stat(2)`, `errno(3C)`, `volmgt_check(3VOLMGT)`, `volmgt_running(3VOLMGT)`, `attributes(5)`, `hal(5)`

Notes This routine requires volume management to be running.

Since `volmgt_inuse()` returns `0` for two different cases (both when a volume is not in use, and when an error occurs), it is up to the user to to check `errno` to differentiate the two, and to ensure that volume management is running.

Name volmgt_ownspath – check volume management name space for path

Synopsis

```
cc [flag]... file... –lvolgmt [library]...
#include <volmgt.h>

int volmgt_ownspath(char *path);
```

Parameters *path* A string containing the path.

Description This function is obsolete. The management of removable media by the Volume Management feature, including `vold`, has been replaced by software that supports the Hardware Abstraction Layer (HAL). Programmatic support for HAL is through the HAL APIs, which are documented on the HAL web site. See `hal(5)`. The return value of this function is undefined.

The `volmgt_ownspath()` function checks to see if a given *path* is contained in the volume management name space. This is achieved by comparing the beginning of the supplied path name with the output from `volmgt_root(3VOLMGT)`

Return Values The return from this function is undefined.

Examples **EXAMPLE 1** Using `volmgt_ownspath()`

The following example first checks if volume management is running, then checks the volume management name space for *path*, and then returns the *id* for the piece of media.

```
char *path;

...

if (volmgt_running()) {
     if (volmgt_ownspath(path)) {
          (void) printf("id of %s is %lld\n",
               path, media_getid(path));
     }
}
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe
Interface Stability	Obsolete

See Also `volmgt_root(3VOLMGT)`, `volmgt_running(3VOLMGT)`, `attributes(5)`, `hal(5)`

Name volmgt_release – release removable media device reservation

Synopsis

```
cc [ flag ... ] file ... -lvolmgt [ library ... ]
#include <volmgt.h>

int volmgt_release(char *dev);
```

Description This function is obsolete. The management of removable media by the Volume Management feature, including `vold`, has been replaced by software that supports the Hardware Abstraction Layer (HAL). Programmatic support for HAL is through the HAL APIs, which are documented on the HAL web site. See `hal(5)`. The return value of this function is undefined.

The `volmgt_release()` routine releases the removable media device reservation specified as *dev*. See `volmgt_acquire(3VOLMGT)` for a description of *dev*.

If *dev* is reserved by the caller, `volmgt_release()` updates the internal device reservation database to indicate that the device is no longer reserved. If the requested device is reserved by another process, the release attempt fails and `errno` is set to `0`.

Return Values The return from this function is undefined.

Errors On failure, `volmgt_release()` returns `0`, and sets `errno` for one of the following conditions:

`EINVAL` *dev* was invalid or missing.

`EBUSY` *dev* was not reserved by the caller.

Examples **EXAMPLE 1** Using `volmgt_release()`

In the following example, volume management is running, and the first floppy drive is reserved, accessed and released.

```
#include <volmgt.h>
char *errp;
if (!volmgt_acquire("floppy0", "FileMgr", 0, &errp,
    NULL)) {
    /* handle error case */
    . . .
}
/* floppy acquired - now access it */
if (!volmgt_release("floppy0")) {
    /* handle error case */
    . . .
}
```

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe
Interface Stability	Obsolete

See Also volmgt_acquire(3VOLMGT), attributes(5), hal(5)

Name volmgt_root – return the volume management root directory

Synopsis

```
cc [ flag... ] file... -lvolmgt [ library ... ]
#include <volmgt.h>
const char *volmgt_root(void);
```

Description This function is obsolete. The management of removable media by the Volume Management feature, including `vold`, has been replaced by software that supports the Hardware Abstraction Layer (HAL). Programmatic support for HAL is through the HAL APIs, which are documented on the HAL web site. See `hal(5)`. The return value of this function is undefined.

The `volmgt_root()` function returns the current volume management root directory, which by default is `/vol` but can be configured to be in a different location.

Return Values The return from this function is undefined.

Errors This function may fail if an `open()` of `/dev/volctl` fails. If this occurs a pointer to the default Volume Management root directory is returned.

Examples **EXAMPLE 1** Finding the Volume Management Root directory.

To find out where the volume management root directory is:

```
if ((path = volmgt_root()) != NULL) {
        (void) printf("Volume Management root dir=%s\n", path);
} else {
        (void) printf("can't find Volume Management root dir\n");
}
```

Files `/dev` default location for the volume management root directory

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe
Interface Stability	Obsolete

See Also `open(2)`, `volmgt_check(3VOLMGT)`, `volmgt_inuse(3VOLMGT)`, `volmgt_running(3VOLMGT)`, `attributes(5)`, `hal(5)`

Notes This function returns the default root directory location even when volume management is not running.

Name volmgt_running – return whether or not volume management is running

Synopsis

```
cc [ flag... ] file... -lvolmgt [ library... ]
#include <volmgt.h>

int volmgt_running(void);
```

Description This function is obsolete. The management of removable media by the Volume Management feature, including `vold`, has been replaced by software that supports the Hardware Abstraction Layer (HAL). Programmatic support for HAL is through the HAL APIs, which are documented on the HAL web site. See `hal(5)`.

`volmgt_running()` tells whether or not Volume Management is running.

Return Values `volmgt_running()` always returns `0` indicating Volume Management (as implemented by `vold`) is not running.

Errors `volmgt_running()` will fail, returning `0`, if a `stat(2)` or `open(2)` of `/dev/volctl` fails, or if any of the following is true:

`ENXIO` Volume Management is not running.

`EINTR` An interrupt signal was detected while checking to see if Volume Management was running.

Examples **EXAMPLE 1** Using `volmgt_running()`

To see if Volume Management is running:

```
if (volmgt_running() != 0) {
        (void) printf("Volume Management is running\n");
} else {
        (void) printf("Volume Management is NOT running\n");
}
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe
Interface Stability	Obsolete

See Also `open(2)`, `stat(2)`, `volmgt_check(3VOLMGT)`, `volmgt_inuse(3VOLMGT)`, `attributes(5)`, `hal(5)`

Notes Volume Management must be running for many of the Volume Management library routines to work.

Name volmgt_symname, volmgt_symdev – convert between Volume Management symbolic names, and the devices that correspond to them

Synopsis

```
cc [ flag... ] file... -lvolmgt [ library... ]
#include <volmgt.h>

char *volmgt_symname(char *pathname);

char *volmgt_symdev(char *symname);
```

Description This function is obsolete. The management of removable media by the Volume Management feature, including `vold`, has been replaced by software that supports the Hardware Abstraction Layer (HAL). Programmatic support for HAL is through the HAL APIs, which are documented on the HAL web site. See `hal(5)`. The return value of this function is undefined.

These two routines compliment each other, translating between Volume Management's symbolic name for a device, called a *symname*, and the `/dev` *pathname* for that same device.

`volmgt_symname( )` converts a supplied `/dev` *pathname* to a `symname`, Volume Management's idea of that device's symbolic name.

`volmgt_symdev( )` does the opposite conversion, converting between a *symname*, Volume Management's idea of a device's symbolic name for a volume, to the `/dev` *pathname* for that device.

Return Values The return from this function is undefined.

Errors `volmgt_symname( )` can fail, returning a null string pointer, if a `stat(2)` of the supplied `pathname` fails, or if an `open(2)` of `/dev/volctl` fails, or if any of the following is true:

`ENXIO` Volume Management is not running.

`EINTR` An interrupt signal was detected while trying to convert the supplied *pathname* to a *symname*.

`volmgt_symdev( )` can fail if an `open(2)` of `/dev/volctl` fails, or if any of the following is true:

`ENXIO` Volume Management is not running.

`EINTR` An interrupt signal was detected while trying to convert the supplied *symname* to a `/dev` *pathname*.

Examples **EXAMPLE 1** Testing Floppies

The following tests how many floppies Volume Management currently sees in floppy drives (up to 10):

```
for (i=0; i < 10; i++) {
        (void) sprintf(path, "floppy%d", i);
        if (volmgt_symdev(path) != NULL) {
                (void) printf("volume %s is in drive %d\n",
```

EXAMPLE 1 Testing Floppies *(Continued)*

```
                    path, i);
        }
}
```

EXAMPLE 2 Finding The Symbolic Name

This code finds out what symbolic name (if any) Volume Management has for `/dev/rdsk/c0t6d0s2`:

```
if ((nm = volmgt_symname("/dev/rdsk/c0t6d0s2")) == NULL) {
        (void) printf("path not managed\n");
} else {
        (void) printf("path managed as %s\n", nm);
}
```

Attributes

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe
Interface Stability	Obsolete

See Also

`open(2)`, `stat(2)`, `free(3C)`, `malloc(3C)`, `volmgt_check(3VOLMGT)`, `volmgt_inuse(3VOLMGT)`, `volmgt_running(3VOLMGT)`, `attributes(5)`, `hal(5)`

Name wsreg_add_child_component, wsreg_remove_child_component, wsreg_get_child_components – add or remove a child component

Synopsis

```
cc [flag ...] file ...-lwsreg [library ...]
#include <wsreg.h>

int wsreg_add_child_component(Wsreg_component *comp,
     const Wsreg_component *childComp);

int wsreg_remove_child_component(Wsreg_component *comp,
     const Wsreg_component *childComp);

Wsreg_component **wsreg_get_child_components(const Wsreg_component *comp);
```

Description The `wsreg_add_child_component()` function adds the component specified by *childComp* to the list of child components contained in the component specified by *comp*.

The `wsreg_remove_child_component()` function removes the component specified by *childComp* from the list of child components contained in the component specified by *comp*.

The `wsreg_get_child_components()` function returns the list of child components contained in the component specified by *comp*.

Return Values The `wsreg_add_child_component()` function returns a non-zero value if the specified child component was successfully added; otherwise, 0 is returned.

The `wsreg_remove_child_component()` function returns a non-zero value if the specified child component was successfully removed; otherwise, 0 is returned.

The `wsreg_get_child_components()` function returns a null-terminated array of `Wsreg_component` pointers that represents the specified component's list of child components. If the specified component has no child components, `NULL` is returned. The resulting array must be released by the caller through a call to `wsreg_free_component_array()`. See `wsreg_create_component(3WSREG)`.

Usage The parent-child relationship between components in the product install registry is used to record a product's structure. Product structure is the arrangement of features and components that make up a product. The structure of installed products can be displayed with the prodreg GUI.

The child component must be installed and registered before the parent component can be. The registration of a parent component that has child components results in each of the child components being updated to reflect their parent component.

Read access to the product install registry is required in order to use these functions because these relationships are held with lightweight component references that can only be fully resolved using the registry contents.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also prodreg(1M), wsreg_can_access_registry(3WSREG), wsreg_create_component(3WSREG), wsreg_initialize(3WSREG), wsreg_register(3WSREG), wsreg_set_parent(3WSREG), attributes(5)

Name wsreg_add_compatible_version, wsreg_remove_compatible_version, wsreg_get_compatible_versions – add or remove a backward-compatible version

Synopsis

```
cc [flag ...] file ...-lwsreg [library ...]
#include <wsreg.h>

int wsreg_add_compatible_version(Wsreg_component *comp,
     const char *version);

int wsreg_remove_compatible_version(Wsreg_component *comp,
     const char *version);

char **wsreg_get_compatible_versions(const Wsreg_component *comp);
```

Description The `wsreg_add_compatible_version()` function adds the version string specified by *version* to the list of backward-compatible versions contained in the component specified by *comp*.

The `wsreg_remove_compatible_version()` function removes the version string specified by *version* from the list of backward-compatible versions contained in the component specified by *comp*.

The `wsreg_get_compatible_versions()` function returns the list of backward-compatible versions contained in the component specified by *comp*.

Return Values The `wsreg_add_compatible_version()` function returns a non-zero value if the specified backward-compatible version was successfully added; otherwise, 0 is returned.

The `wsreg_remove_compatible_version()` function returns a non-zero value if the specified backward-compatible version was successfully removed; otherwise, 0 is returned.

The `wsreg_get_compatible_versions()` function returns a null-terminated array of `char` pointers that represents the specified component's list of backward-compatible versions. If the specified component has no such versions, `NULL` is returned. The resulting array and its contents must be released by the caller.

Usage The list of backward compatible versions is used to allow components that are used by multiple products to upgrade successfully without compromising any of its dependent products. The installer that installs such an update can check the list of backward-compatible versions and look at what versions are required by all of the dependent components to ensure that the upgrade will not result in a broken product.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also prodreg(1M), wsreg_initialize(3WSREG), wsreg_register(3WSREG), wsreg_set_version(3WSREG), attributes(5)

Name wsreg_add_dependent_component, wsreg_remove_dependent_component, wsreg_get_dependent_components – add or remove a dependent component

Synopsis

```
cc [flag ...] file ...-lwsreg [library ...]
#include <wsreg.h>

int wsreg_add_dependent_component(Wsreg_component *comp,
     const Wsreg_component *dependentComp);

int wsreg_remove_dependent_component(Wsreg_component *comp,
     const Wsreg_component *dependentComp);

Wsreg_component **wsreg_get_dependent_components(const Wsreg_component *comp);
```

Description The `wsreg_add_dependent_component()` function adds the component specified by *dependentComp* to the list of dependent components contained in the component specified by *comp*.

The `wsreg_remove_dependent_component()` function removes the component specified by *dependentComp* from the list of dependent components contained in the component specified by *comp*.

The `wsreg_get_dependent_components()` function returns the list of dependent components contained in the component specified by *comp*.

Return Values The `wsreg_add_dependent_component()` function returns a non-zero value if the specified dependent component was successfully added; otherwise, 0 is returned.

The `wsreg_remove_dependent_component()` function returns a non-zero value if the specified dependent component was successfully removed; otherwise, 0 is returned.

The `wsreg_get_dependent_components()` function returns a null-terminated array of `Wsreg_component` pointers that represents the specified component's list of dependent components. If the specified component has no dependent components, `NULL` is returned. The resulting array must be released by the caller through a call to `wsreg_free_component_array()`. See `wsreg_create_component(3WSREG)`.

Usage The relationship between two components in which one must be installed for the other to be complete is a dependent/required relationship. The component that is required by the other component is the required component. The component that requires the other is the dependent component.

The required component must be installed and registered before the dependent component can be. Uninstaller applications should check the registry before uninstalling and unregistering components so a successful uninstallation of one product will not result in another product being compromised.

Read access to the product install registry is required to use these functions because these relationships are held with lightweight component references that can only be fully resolved using the registry contents.

The act of registering a component having required components results in the converse dependent relationships being established automatically.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also wsreg_add_required_component(3WSREG), wsreg_can_access_registry(3WSREG), wsreg_create_component(3WSREG), wsreg_initialize(3WSREG), wsreg_register(3WSREG), attributes(5)

Name wsreg_add_display_name, wsreg_remove_display_name, wsreg_get_display_name, wsreg_get_display_languages – add, remove, or return a localized display name

Synopsis

```
cc [flag ...] file ...-lwsreg [library ...]
#include <wsreg.h>

int wsreg_add_display_name(Wsreg_component *comp, const char *language,
     const char *display_name);

int wsreg_remove_display_name(Wsreg_component *comp, const char *language);

char *wsreg_get_display_name(const Wsreg_component *comp,
     const char *language);

char  **wsreg_get_display_languages(const Wsreg_component *comp);
```

Description For each of these functions, the *comp* argument specifies the component on which these functions operate. The *language* argument is the ISO 639 language code identifying a particular display name associated with the specified component.

The `wsreg_add_display_name()` function adds the display name specified by *display_name* to the component specified by *comp*.

The `wsreg_remove_display_name()` function removes a display name from the component specified by *comp*.

The `wsreg_get_display_name()` function returns a display name from the component specified by *comp*.

The `wsreg_get_display_languages()` returns the ISO 639 language codes for which display names are available from the component specified by *comp*.

Return Values The `wsreg_add_display_name()` function returns a non-zero value if the display name was set correctly; otherwise 0 is returned.

The `wsreg_remove_display_name()` function returns a non-zero value if the display name was removed; otherwise 0 is returned.

The `wsreg_get_display_name()` function returns the display name from the specified component if the component has a display name for the specified language code. Otherwise, `NULL` is returned. The caller must not free the resulting display name.

The `wsreg_get_display_languages()` function returns a null-terminated array of ISO 639 language codes for which display names have been set into the specified component. If no display names have been set, `NULL` is returned. It is the caller's responsibility to release the resulting array, but not the contents of the array.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also wsreg_initialize(3WSREG), attributes(5)

Name wsreg_add_required_component, wsreg_remove_required_component, wsreg_get_required_components – add or remove a required component

Synopsis

```
cc [flag ...] file ...-lwsreg [library ...]
#include <wsreg.h>

int wsreg_add_required_component(Wsreg_component *comp,
     const Wsreg_component *requiredComp);

int wsreg_remove_required_component(Wsreg_component *comp,
     const Wsreg_component *requiredComp);

Wsreg_component **wsreg_get_required_components
     (const Wsreg_component *comp);
```

Description The `wsreg_add_required_component()` function adds the component specified by *requiredComp* to the list of required components contained in the component specified by *comp*.

The `wsreg_remove_required_component()` function removes the component specified by *requiredComp* from the list of required components contained in the component specified by *comp*.

The `wsreg_get_required_components()` function returns the list of required components contained in the component specified by *comp*.

Return Values The `wsreg_add_required_component()` function returns a non-zero value if the specified required component was successfully added. Otherwise, 0 is returned.

The `wsreg_remove_required_component()` function returns a non-zero value if the specified required component was successfully removed. Otherwise, 0 is returned.

The `wsreg_get_required_components()` function returns a null-terminated array of `Wsreg_component` pointers that represents the specified component's list of required components. If the specified component has no required components, `NULL` is returned. The resulting array must be released by the caller through a call to `wsreg_free_component_array()`. See `wsreg_create_component(3WSREG)`.

Usage The relationship between two components in which one must be installed for the other to be complete is a dependent/required relationship. The component that is required by the other component is the required component. The component that requires the other is the dependent component.

The required component must be installed and registered before the dependent component can be. Uninstaller applications should check the registry before uninstalling and unregistering components so a successful uninstallation of one product will not result in another product being compromised.

Read access to the product install registry is required in order to use these functions because these relationships are held with lightweight component references that can only be fully resolved using the registry contents.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also wsreg_add_dependent_component(3WSREG), wsreg_can_access_registry(3WSREG), wsreg_create_component(3WSREG), wsreg_initialize(3WSREG), wsreg_register(3WSREG), attributes(5)

Name wsreg_can_access_registry – determine access to product install registry

Synopsis

```
cc [flag ...] file ...-lwsreg [library ...]
#include <fcntl.h>
#include <wsreg.h>

int wsreg_can_access_registry(int access_flag);
```

Description The `wsreg_can_access_registry()` function is used to determine what access, if any, an application has to the product install registry.

The *access_flag* argument can be one of the following:

`O_RDONLY` Inquire about read only access to the registry.

`O_RDWR` Inquire about modify (read and write) access to the registry.

Return Values The `wsreg_can_access_registry()` function returns non-zero if the specified access level is permitted. A return value of 0 indicates the specified access level is not permitted.

Examples **EXAMPLE 1** Initialize the registry and determine if access to the registry is permitted.

```
#include <fcntl.h>
#include <wsreg.h>

int main(int argc, char **argv)
{
    int result;
    if (wsreg_initialize(WSREG_INIT_NORMAL, NULL)) {
        printf("conversion recommended, sufficient access denied\n");
    }

    if (wsreg_can_access_registry(O_RDONLY)) {
        printf("registry read access granted\n");
    } else {
        printf("registry read access denied\n");
    }

    if (wsreg_can_access_registry(O_RDWR)) {
        printf("registry read/write access granted\n");
    } else {
        printf("registry read/write access denied\n");
    }
}
```

Usage The `wsreg_initialize(3WSREG)` function must be called before calls to `wsreg_can_access_registry()` can be made.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also wsreg_initialize(3WSREG), attributes(5)

Name wsreg_clone_component – clone a component

Synopsis

```
cc [flag ...]  file ...-lwsreg [library ...]
#include <wsreg.h>

Wsreg_component *wsreg_clone_component(const Wsreg_component *comp);
```

Description The `wsreg_clone_component()` function clones the component specified by *comp*.

Return Values The `wsreg_clone_component()` returns a pointer to a component that is configured exactly the same as the component specified by *comp*.

Usage The resulting component must be released through a call to `wsreg_free_component()` by the caller. See `wsreg_create_component(3WSREG)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `wsreg_create_component(3WSREG)`, `wsreg_initialize(3WSREG)`, `wsreg_get(3WSREG)`, `attributes(5)`

Name wsreg_components_equal – determine equality of two components

Synopsis

```
cc [flag ...]  file  ...-lwsreg [library ...]
#include <wsreg.h>

int wsreg_components_equal(const Wsreg_component *comp1,
     const Wsreg_component *comp2);
```

Description The `wsreg_components_equal()` function determines if the component specified by the *comp1* argument is equal to the component specified by the *comp2* argument. Equality is evaluated based only on the content of the two components, not the order in which data was set into the components.

Return Values The `wsreg_components_equal()` function returns a non-zero value if the component specified by the *comp1* argument is equal to the component specified by the *comp2* argument. Otherwise, 0 is returned.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `wsreg_clone_component(3WSREG)`, `wsreg_create_component(3WSREG)`, `wsreg_initialize(3WSREG)`, `attributes(5)`

Name wsreg_create_component, wsreg_free_component, wsreg_free_component_array – create or release a component

Synopsis

```
cc [flag ...]  file ...-lwsreg [library ...]
#include <wsreg.h>

Wsreg_component *wsreg_create_component(const char *uuid);

void wsreg_free_component(Wsreg_component *comp);

int wsreg_free_component_array(Wsreg_component **complist);
```

Description The `wsreg_create_component()` function allocates a new component and assigns the uuid (universal unique identifier) specified by *uuid* to the resulting component.

The `wsreg_free_component()` function releases the memory associated with the component specified by *comp*.

The `wsreg_free_component_array()` function frees the null-terminated array of component pointers specified by *complist*. This function can be used to free the results of a call to `wsreg_get_all()`. See `wsreg_get`(3WSREG).

Return Values The `wsreg_create_component()` function returns a pointer to the newly allocated `Wsreg_component` structure.

The `wsreg_free_component_array()` function returns a non-zero value if the specified `Wsreg_component` array was freed successfully. Otherwise, 0 is returned.

Usage A minimal registerable `Wsreg_component` configuration must include a version, unique name, display name, and an install location.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `wsreg_add_display_name`(3WSREG), `wsreg_get`(3WSREG), `wsreg_initialize`(3WSREG), `wsreg_register`(3WSREG), `wsreg_set_id`(3WSREG), `wsreg_set_location`(3WSREG), `wsreg_set_unique_name`(3WSREG), `wsreg_set_version`(3WSREG), `attributes`(5)

Name wsreg_get, wsreg_get_all – query product install registry

Synopsis

```
cc [flag ...] file ...-lwsreg [library ...]
#include <wsreg.h>

Wsreg_component *wsreg_get(const Wsreg_query *query);

Wsreg_component **wsreg_get_all(void);
```

Description The `wsreg_get()` function queries the product install registry for a component that matches the query specified by *query*.

The `wsreg_get_all()` function returns all components currently registered in the product install registry.

Return Values The `wsreg_get()` function returns a pointer to a `Wsreg_component` structure representing the registered component. If no component matching the specified query is currently registered, `wsreg_get()` returns `NULL`.

The `wsreg_get_all()` function returns a null-terminated array of `Wsreg_component` pointers. Each element in the resulting array represents one registered component.

Usage The `wsreg` library must be initialized by a call to `wsreg_initialize(3WSREG)` before any call to `wsreg_get()` or `wsreg_get_all()`.

The `Wsreg_component` pointer returned from `wsreg_get()` should be released through a call to `wsreg_free_component()`. See `wsreg_create_component(3WSREG)`.

The `Wsreg_component` pointer array returned from `wsreg_get_all()` should be released through a call to `wsreg_free_component_array()`. See `wsreg_create_component(3WSREG)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `wsreg_create_component(3WSREG)`, `wsreg_initialize(3WSREG)`, `wsreg_register(3WSREG)`, `attributes(5)`

Name wsreg_initialize – initialize wsreg library

Synopsis

```
cc [flag ...] file ...-lwsreg [library ...]
#include <wsreg.h>

int wsreg_initialize(Wsreg_init_level level, const char *alternate_root);
```

Description The `wsreg_initialize()` function initializes the `wsreg` library.

The *level* argument can be one of the following:

`WSREG_INIT_NORMAL`	If an old registry file is present, attempt to perform a conversion.
`WSREG_INIT_NO_CONVERSION`	If an old conversion file is present, do not perform the conversion, but indicate that the conversion is recommended.

The *alternate_root* argument can be used to specify a root prefix. If `NULL` is specified, no root prefix is used.

Return Values The `wsreg_initialize()` function can return one of the following:

`WSREG_SUCCESS`	The initialization was successful and no registry conversion is necessary.
`WSREG_CONVERSION_RECOMMENDED`	An old registry file exists and should be converted.

A conversion is attempted if the *init_level* argument is `WSREG_INIT_NORMAL` and a registry file from a previous version of the product install registry exists. If the `wsreg_initialize()` function returns `WSREG_CONVERSION_RECOMMENDED`, the user either does not have permission to update the product install registry or does not have read/write access to the previous registry file.

Usage The `wsreg_initialize()` function must be called before any other `wsreg` library functions.

The registry conversion can take some time to complete. The registry conversion can also be performed using the graphical registry viewer `/usr/bin/prodreg` or by the registry converter `/usr/bin/regconvert`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `prodreg(1M)`, `wsreg_can_access_registry(3WSREG)`, `attributes(5)`

Name wsreg_query_create, wsreg_query_free – create a new query

Synopsis

```
cc [flag ...]  file ...-lwsreg [library ...]
#include <wsreg.h>

Wsreg_query *wsreg_query_create(void);

void wsreg_query_free(Wsreg_query *query);
```

Description The `wsreg_query_create()` function allocates a new query that can retrieve components from the product install registry.

The `wsreg_query_free()` function releases the memory associated with the query specified by *query*.

Return Values The `wsreg_query_create()` function returns a pointer to the newly allocated query. The resulting query is completely empty and must be filled in to describe the desired component.

Usage The query identifies fields used to search for a specific component in the product install registry. The query must be configured and then passed to the `wsreg_get(3WSREG)` function to perform the registry query.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `wsreg_get(3WSREG)`, `wsreg_initialize(3WSREG)`, `wsreg_query_set_id(3WSREG)`, `wsreg_query_set_instance(3WSREG)`, `wsreg_query_set_location(3WSREG)`, `wsreg_query_set_unique_name(3WSREG)`, `wsreg_query_set_version(3WSREG)`, `wsreg_unregister(3WSREG)`, `attributes(5)`

Name wsreg_query_set_id, wsreg_query_get_id – set or get the uuid of a query

Synopsis

```
cc [flag ...] file ...-lwsreg [library ...]
#include <wsreg.h>

int wsreg_query_set_id(Wsreg_query *query, const char *uuid);

char *wsreg_query_get_id(const Wsreg_query *query);
```

Description The `wsreg_query_set_id()` function sets the uuid (universal unique identifier) specified by *uuid* in the query specified by *query*. If a uuid has already been set in the specified query, the resources associated with the previously set uuid are released.

The `wsreg_query_get_id()` function returns the uuid associated with the query specified by *query*. The resulting string is not a copy and must not be released by the caller.

Return Values The `wsreg_query_set_id()` function returns non-zero if the uuid was set correctly; otherwise 0 is returned.

The `wsreg_query_get_id()` function returns the uuid associated with the specified query.

Usage The query identifies fields used to search for a specific component in the product install registry. By specifying the uuid, the component search is narrowed to all components in the product install registry that have the specified uuid.

Other fields can be specified in the same query to further narrow the search.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `wsreg_get(3WSREG)`, `wsreg_initialize(3WSREG)`, `wsreg_query_create(3WSREG)`, `attributes(5)`

Name wsreg_query_set_instance, wsreg_query_get_instance – set or get the instance of a query

Synopsis

```
cc [flag ...]  file ...-lwsreg [library ...]
#include <wsreg.h>

 int wsreg_query_set_instance(Wsreg_query *query, int instance);

int wsreg_query_get_instance(Wsreg_query *comp);
```

Description The `wsreg_query_set_instance()` function sets the instance number specified by *instance* in the query specified by *query*.

The `wsreg_query_get_instance()` function retrieves the instance from the query specified by *query*.

Return Values The `wsreg_query_set_instance()` function returns a non-zero value if the instance was set correctly; otherwise 0 is returned.

The `wsreg_query_get_instance()` function returns the instance number from the specified query. It returns 0 if the instance number has not been set.

Usage The query identifies fields used to search for a specific component in the product install registry. By specifying the instance, the component search is narrowed to all components in the product install registry that have the specified instance.

Other fields can be specified in the same query to further narrow down the search.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `wsreg_get(3WSREG)`, `wsreg_initialize(3WSREG)`, `wsreg_query_create(3WSREG)`, `attributes(5)`

Name wsreg_query_set_location, wsreg_query_get_location – set or get the location of a query

Synopsis

```
cc [flag ...] file ...-lwsreg [library ...]
#include <wsreg.h>

int wsreg_query_set_location(Wsreg_query *query, const char *location);

char *wsreg_query_get_location(Wsreg_query *query);
```

Description The `wsreg_query_set_location()` function sets the location specified by *location* in the query specified by *query*. If a location has already been set in the specified query, the resources associated with the previously set location are released.

The `wsreg_query_get_location()` function gets the location string from the query specified by *query*.

Return Values The `wsreg_query_set_location()` function returns a non-zero value if the location was set correctly; otherwise 0 is returned.

The `wsreg_query_get_location()` function returns the location from the specified query structure. The resulting location string is not a copy, so it must not be released by the caller.

Usage The query identifies fields used to search for a specific component in the product install registry. By specifying the install location, the component search is narrowed to all components in the product install registry that are installed in the same location.

Other fields can be specified in the same query to further narrow the search.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `wsreg_get(3WSREG)`, `wsreg_initialize(3WSREG)`, `wsreg_query_create(3WSREG)`, `attributes(5)`

Name wsreg_query_set_unique_name, wsreg_query_get_unique_name – set or get the unique name of a query

Synopsis

```
cc [flag ...]  file  ...-lwsreg [library ...]
#include <wsreg.h>

int wsreg_query_set_unique_name(Wsreg_query *query,
     const char *unique_name);

char *wsreg_query_get_unique_name(const Wsreg_query *query);
```

Description The `wsreg_query_set_unique_name()` function sets the unique name specified by *unique_name* in the query specified by *query*. If a unique name has already been set in the specified query, the resources associated with the previously set unique name are released.

The `wsreg_query_get_unique_name()` function gets the unique name string from the query specified by *query*. The resulting string is not a copy and must not be released by the caller.

Return Values The `wsreg_query_set_unique_name()` function returns a non-zero value if the unique_name was set correctly; otherwise 0 is returned.

The `wsreg_query_get_unique_name()` function returns a copy of the *unique_name* from the specified query.

Usage The query identifies fields used to search for a specific component in the product install registry. By specifying the unique name, the component search is narrowed to all components in the product install registry that have the specified unique name.

Other fields can be specified in the same query to further narrow the search.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `wsreg_get(3WSREG)`, `wsreg_initialize(3WSREG)`, `wsreg_query_create(3WSREG)`, `attributes(5)`

Name wsreg_query_set_version, wsreg_query_get_version – set or get the version of a query

Synopsis

```
cc [flag ...] file ... -lwsreg [library ...]
#include <wsreg.h>

int wsreg_query_set_version(Wsreg_query *query, const char *version);

char *wsreg_query_get_version(const Wsreg_query *query);
```

Description The `wsreg_query_set_version()` function sets the version specified by *version* in the query specified by *query*. If a version has already been set in the specified query, the resources associated with the previously set version are released.

The `wsreg_query_get_version()` function gets the version string from the query specified by *query*. The resulting string is not a copy and must not be released by the caller.

Return Values The `wsreg_query_set_version()` function returns a non-zero value if the version was set correctly; otherwise 0 is returned.

The `wsreg_query_get_version()` function returns the version from the specified query. If no version has been set, `NULLt` is returned. The resulting version string is not a copy and must not be released by the caller.

Usage The query identifies fields used to search for a specific component in the product install registry. By specifying the version, the component search is narrowed to all components in the product install registry that have the specified version.

Other fields can be specified in the same query to further narrow the search.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `wsreg_get(3WSREG)`, `wsreg_initialize(3WSREG)`, `wsreg_query_create(3WSREG)`, `attributes(5)`

Name wsreg_register – register a component in the product install registry

Synopsis

```
cc [flag ...]  file ...-lwsreg [library ...]
#include <wsreg.h>

int wsreg_register(Wsreg_component *comp);
```

Description The `wsreg_register()` function updates a component in the product install registry.

If *comp* is already in the product install registry, the call to `wsreg_register()` results in the currently registered component being updated. Otherwise, *comp* is added to the product install registry.

An instance is assigned to the component upon registration. Subsequent component updates retain the same component instance.

If *comp* has required components, each required component is updated to reflect the required component relationship.

If *comp* has child components, each child component that does not already have a parent is updated to reflect specified component as its parent.

Return Values Upon successful completion, a non-zero value is returned. If the component could not be updated in the product install registry, 0 is returned.

Examples **EXAMPLE 1** Create and register a component.

The following example creates and registers a component.

```
#include <wsreg.h>

int main (int argc, char **argv)
{
    char *uuid = "d6cf2869-1dd1-11b2-9fcb-080020b69971";
         Wsreg_component *comp = NULL;

         /* Initialize the registry */
         wsreg_initialize(WSREG_INIT_NORMAL, NULL);

         /* Create the component */
         comp = wsreg_create_component(uuid);
         wsreg_set_unique_name(comp, "wsreg_example_1");
         wsreg_set_version(comp, "1.0");
         wsreg_add_display_name(comp, "en", "Example 1 component");
         wsreg_set_type(comp, WSREG_COMPONENT);
         wsreg_set_location(comp, "/usr/local/example1_component");

         /* Register the component */
         wsreg_register(comp);
```

EXAMPLE 1 Create and register a component. *(Continued)*

```
        wsreg_free_component(comp);
        return 0;
}
```

Usage A product's structure can be recorded in the product install registry by registering a component for each element and container in the product definition. The product and each of its features would be registered in the same way as a package that represents installed files.

Components should be registered only after they are successfully installed. If an entire product is being registered, the product should be registered after all components and features are installed and registered.

In order to register correctly, the component must be given a uuid, unique name, version, display name, and a location. The location assgined to product structure components should generally be the location in which the user chose to install the product.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `wsreg_get(3WSREG)`, `wsreg_initialize(3WSREG)`, `wsreg_create_component(3WSREG)`, `wsreg_unregister(3WSREG)`, `attributes(5)`

Name wsreg_set_data, wsreg_get_data, wsreg_get_data_pairs – add or retrieve a key-value pair

Synopsis

```
cc [flag ...]  file  ...-lwsreg [library ...]
#include <wsreg.h>

int wsreg_set_data(Wsreg_component *comp, const char *key,
     const char *value);

char *wsreg_get_data(const Wsreg_component *comp, const char *key);

char *wsreg_get_data_pairs(const Wsreg_component *comp);
```

Description The `wsreg_set_data()` function adds the key-value pair specified by *key* and *value* to the component specified by *comp*. If *value* is `NULL`, the key and current value is removed from the specified component.

The `wsreg_get_data()` function retrieves the value associated with the key specified by *key* from the component specified by *comp*.

The `wsreg_get_data_pairs()` function returns the list of key-value pairs from the component specified by *comp*.

Return Values The `wsreg_set_data()` function returns a non-zero value if the specified key-value pair was successfully added. It returns 0 if the addition failed. If `NULL` is passed as the value, the current key-value pair are removed from the specified component.

The `wsreg_get_data()` function returns the value associated with the specified key. It returns `NULL` if there is no value associated with the specified key. The `char` pointer that is returned is not a clone, so it must not be freed by the caller.

The `wsreg_get_data_pairs()` function returns a null-terminated array of `char` pointers that represents the specified component's list of data pairs. The even indexes of the resulting array represent the key names. The odd indexes of the array represent the values. If the specified component has no data pairs, `NULL` is returned. The resulting array (not its contents) must be released by the caller.

Usage Any string data can be associated with a component. Because this information can be viewed in the prodreg registry viewer, it is a good place to store support contact information.

After the data pairs are added or removed, the component must be updated with a call to `wsreg_register(3WSREG)` for the modifications to be persistent.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also prodreg(1M), wsreg_initialize(3WSREG), wsreg_register(3WSREG), attributes(5)

Name wsreg_set_id, wsreg_get_id – set or get the uuid of a component

Synopsis

```
cc [flag ...]  file ...-lwsreg [library ...]
#include <wsreg.h>

int wsreg_set_id(Wsreg_component *comp, const char *uuid);

char *wsreg_get_id(const Wsreg_component *comp);
```

Description The `wsreg_set_id()` function sets the uuid (universal unique identifier) specified by *uuid* into the component specified by *comp*. If a uuid has already been set into the specified component, the resources associated with the previously set uuid are released.

The `wsreg_get_id()` function returns a copy of the uuid of the component specified by *comp*. The resulting string must be released by the caller.

Return Values The `wsreg_set_id()` function returns non-zero if the uuid was set correctly; otherwise 0 is returned.

The `wsreg_get_id()` function returns a copy of the specified component's uuid.

Usage Generally, the uuid will be set into a component by the `wsreg_create_component(3WSREG)` function, so a call to the `wsreg_set_id()` is not necessary.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `wsreg_create_component(3WSREG)`, `wsreg_initialize(3WSREG)`, `attributes(5)attributes(5)`

Name wsreg_set_instance, wsreg_get_instance – set or get the instance of a component

Synopsis

```
cc [flag ...] file ...-lwsreg [library ...]
#include <wsreg.h>

int wsreg_set_instance(Wsreg_component *comp, int instance);

int wsreg_get_instance(Wsreg_component *comp);
```

Description The `wsreg_set_instance()` function sets the instance number specified by *instance* of the component specified by *comp*. The instance number and uuid are used to uniquely identify any component in the product install registry.

The `wsreg_get_instance()` function determines the instance number associated with the component specified by *comp*.

Return Values The `wsreg_set_instance()` function returns a non-zero value if the instance was set correctly; otherwise 0 is returned.

The `wsreg_get_instance()` function returns the instance number associated with the specified component.

Examples **EXAMPLE 1** Get the instance value of a registered component.

The following example demonstrates how how to get the instance value of a registered component.

```
#include <fcntl.h>
#include <wsreg.h>

int main (int argc, char **argv)
{
    char *uuid = "d6cf2869-1dd1-11b2-9fcb-080020b69971";
    Wsreg_component *comp = NULL;

    /* Initialize the registry */
    wsreg_initialize(WSREG_INIT_NORMAL, NULL);
    if (!wsreg_can_access_registry(O_RDWR)) {
        printf("No permission to modify the registry.\n");
        return 1;
    }

    /* Create a component */
    comp = wsreg_create_component(uuid);
    wsreg_set_unique_name(comp, "wsreg_example_1");
    wsreg_set_version(comp, "1.0");
    wsreg_add_display_name(comp, "en", "Example 1 component");
    wsreg_set_type(comp, WSREG_COMPONENT);
    wsreg_set_location(comp, "/usr/local/example1_component");
```

EXAMPLE 1 Get the instance value of a registered component. *(Continued)*

```
    /* Register */
    wsreg_register(comp);

    printf("Instance %d was assigned\n", wsreg_get_instance(comp));

    wsreg_free_component(comp);
    return 0;
}
```

Usage Upon component registration with the `wsreg_register(3WSREG)` function, the instance number is set automatically. The instance number of 0 (the default) indicates to the `wsreg_register()` function that an instance number should be looked up and assigned during registration. If a component with the same uuid and location is already registered in the product install registry, that component's instance number will be used during registration.

After registration of a component, the `wsreg_get_instance()` function can be used to determine what instance value was assigned.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `wsreg_create_component(3WSREG)`, `wsreg_register(3WSREG)`, `attributes(5)`

Name wsreg_set_location, wsreg_get_location – set or get the location of a component

Synopsis

```
cc [flag ...]  file ...-lwsreg [library ...]
#include <wsreg.h>

int wsreg_set_location(Wsreg_component *comp, const char *location);

char *wsreg_get_location(const Wsreg_component *comp);
```

Description The `wsreg_set_location()` function sets the location specified by *location* into the component specified by *comp*. Every component must have a location before being registered. If a location has already been set into the specified component, the resources associated with the previously set location are released.

The `wsreg_get_location()` function gets the location string from the component specified by *comp*. The resulting string must be released by the caller.

Return Values The `wsreg_set_location()` function returns a non-zero value if the location was set correctly; otherwise 0 is returned.

The `wsreg_get_location()` function returns a copy of the location from the specified component.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `wsreg_initialize(3WSREG)`, `attributes(5)`

Name wsreg_set_parent, wsreg_get_parent – set or get the parent of a component

Synopsis

```
cc [flag ...]  file  ...-lwsreg [library ...]
#include <wsreg.h>

void wsreg_set_parent(Wsreg_component *comp,
     const Wsreg_component *parent);

Wsreg_component *wsreg_get_parent(const Wsreg_component *comp);
```

Description The `wsreg_set_parent()` function sets the parent specified by *parent* of the component specified by *comp*.

The `wsreg_get_parent()` function gets the parent of the component specified by *comp*.

Return Values The `wsreg_get_parent()` function returns a pointer to a `Wsreg_component` structure that represents the parent of the specified component. If the specified component does not have a parent, `NULL` is returned. If a non-null value is returned, it the caller's responsibility to release the memory associated with the resulting `Wsreg_component` pointer with a call to `wsreg_free_component()`. See `wsreg_create_component(3WSREG)`.

Usage The parent of a component is set as a result of registering the parent component. When a component that has children is registered, all of the child components are updated to reflect the newly registered component as their parent. This update only occurs if the child component does not already have a parent component set.

The specified parent component is reduced to a lightweight component reference that uniquely identifies the parent in the product install registry. This lightweight reference includes the parent's uuid and instance number.

The parent must be registered before a call to `wsreg_set_parent()` can be made, since the parent's instance number must be known at the time the `wsreg_set_parent()` function is called.

A process needing to call `wsreg_set_parent()` or `wsreg_get_parent()` must have read access to the product install registry.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `wsreg_can_access_registry(3WSREG)`, `wsreg_create_component(3WSREG)`, `wsreg_initialize(3WSREG)`, `wsreg_register(3WSREG)`, `wsreg_set_instance(3WSREG)`, `attributes(5)`

Name wsreg_set_type, wsreg_get_type – set or get the type of a component

Synopsis

```
cc [ flag... ] file ... -lwsreg [ library ... ]
#include <wsreg.h>

int wsreg_set_type(Wsreg_component *comp, Wsreg_component_type type);

Wsreg_component_type wsreg_get_type(const Wsreg_component *comp);
```

Description The `wsreg_set_type()` function sets the type specified by *type* in the component specified by *comp*.

The `wsreg_get_type()` function retrieves the type from the component specified by *comp*.

Return Values The `wsreg_set_type()` function returns a non-zero value if the type is set successfully; otherwise 0 is returned.

The `wsreg_get_type()` function returns the type currently set in the component specified by *comp*.

Usage The component type is used to indicate whether a `Wsreg_component` structure represents a product, feature, or component. The *type* argument can be one of the following:

`WSREG_PRODUCT` Indicates the `Wsreg_component` represents a product. A product is a collection of features and/or components.

`WSREG_FEATURE` Indicates the `Wsreg_component` represents a feature. A feature is a collection of components.

`WSREG_COMPONENT` Indicates the `Wsreg_component` represents a component. A component is a collection of files that may be installed.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `wsreg_create_component(3WSREG)`, `wsreg_initialize(3WSREG)`, `wsreg_register(3WSREG)`, `wsreg_set_instance(3WSREG)`, `attributes(5)`

Name wsreg_set_uninstaller, wsreg_get_uninstaller – set or get the uninstaller of a component

Synopsis

```
cc [flag ...]  file ...-lwsreg [library ...]
#include <wsreg.h>

int wsreg_set_uninstaller(Wsreg_component *comp, const char *uninstaller);

char *wsreg_set_uninstaller(const Wsreg_component *comp);
```

Description The `wsreg_set_uninstaller()` function sets the uninstaller specified by *uninstaller* in the component specified by *comp*. If an uninstaller has already been set in the specified component, the resources associated with the previously set uninstaller are released.

The `wsreg_get_uninstaller()` function gets the uninstaller string from the component specified by *comp*. The resulting string must be released by the caller.

Return Values The `wsreg_set_uninstaller()` function returns a non-zero value if the uninstaller was set correctly; otherwise 0 is returned.

The `wsreg_get_uninstaller()` function returns a copy of the uninstaller from the specified component.

Usage An uninstaller is usually only associated with a product, not with every component that comprises a product. The uninstaller string is a command that can be passed to the shell to launch the uninstaller.

If an uninstaller is set in a registered component, the `prodreg(1M)` registry viewer will provide an uninstall button that will invoke the uninstaller.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `prodreg(1M)`, `wsreg_initialize(3WSREG)`, `attributes(5)`

Name wsreg_set_unique_name, wsreg_get_unique_name – set or get the unique name of a component

Synopsis

```
cc [flag ...] file ...-lwsreg [library ...]
#include <wsreg.h>

int wsreg_set_unique_name(Wsreg_component *comp, const char *unique_name);

char *wsreg_get_unique_name(const Wsreg_component *comp);
```

Description The `wsreg_set_unique_name()` function sets the unique name specified by *unique_name* in the component specified by *comp*. Every component must have a unique name before being registered. If a unique name has already been set in the specified component, the resources associated with the previously set unique name are released.

The `wsreg_get_unique_name()` function gets the unique name string from the component specified by *comp*. The resulting string must be released by the caller.

Return Values The `wsreg_set_unique_name()` function returns a non-zero value if the unique name was set correctly; otherwise it returns 0.

The `wsreg_get_unique_name()` function returns a copy of the unique name from the specified component.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `wsreg_initialize(3WSREG)`, `attributes(5)`

Name wsreg_set_vendor, wsreg_get_vendor – set or get the vendor of a componentt

Synopsis

```
cc [flag ...]  file ...-lwsreg [library ...]
#include <wsreg.h>

int wsreg_set_vendor(Wsreg_component *comp, const char *vendor);

char *wsreg_get_vendor(const Wsreg_component *comp);
```

Description The `wsreg_set_vendor()` function sets the vendor specified by *vendor* in the component specified by *comp*. The *vendor* argument is a string that identifies the vendor of the component. If a vendor has already been set in the specified component, the resources associated with the previously set vendor are released.

The `wsreg_get_vendor()` function gets the vendor string from the component specified by *comp*. The resulting string must be released by the caller.

Return Values The `wsreg_set_vendor()` function returns a non-zero value if the vendor was set correctly; otherwise it returns 0.

The `wsreg_get_vendor()` function returns a copy of the vendor from the specified component.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `wsreg_initialize(3WSREG)`, `attributes(5)`

Name wsreg_set_version, wsreg_get_version – set or get the version of a component

Synopsis

```
cc [flag ...]  file ...-lwsreg [library ...]
#include <wsreg.h>

int wsreg_set_version(Wsreg_component *comp,  const char *version);

char *wsreg_get_version(const Wsreg_component *comp);
```

Description The `wsreg_set_version()` function sets the version specified by *version* in the component specified by *comp*. The *version* argument is a string that represents the version of the component. Every component must have a version before being registered. If a version has already been set in the specified component, the resources associated with the previously set version are released.

The `wsreg_get_version()` function gets the version string from the component specified by *comp*. The resulting string must be released by the caller.

Return Values The `wsreg_set_version()` function returns a non-zero value if the version was set correctly; otherwise it returns 0.

The `wsreg_get_version()` function returns a copy of the version from the specified component.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also `wsreg_initialize(3WSREG)`, `attributes(5)`

Name wsreg_unregister – remove a component from the product install registry

Synopsis

```
cc [flag ...]  file ...-lwsreg [library ...]
#include <wsreg.h>

int wsreg_unregister(const Wsreg_component *comp);
```

Description The `wsreg_unregister()` function removes the component specified by *comp* from the product install registry. The component will only be removed if the *comp* argument has a matching uuid, instance, and version.

Usually, the component retrieved through a call to `wsreg_get(3WSREG)` before being passed to the `wsreg_unregister()` function.

If the component has required components, the respective dependent components will be updated to reflect the change.

A component that has dependent components cannot be unregistered until the dependent components are uninstalled and unregistered.

Return Values Upon successful completion, a non-zero return value is returned. If the component could not be unregistered, 0 is returned.

Examples **EXAMPLE 1** Unregister a component.

The following example demonstrates how to unregister a component.

```
#include <stdio.h>
#include <wsreg.h>

int main(int argc, char **argv)
{
    char *uuid = "d6cf2869-1dd1-11b2-9fcb-080020b69971";
         char *location = "/usr/local/example1_component";
         Wsreg_query *query = NULL;
         Wsreg_component *comp = NULL;

         /* Initialize the registry */
         wsreg_initialize(WSREG_INIT_NORMAL, NULL);

         /* Query for the component */
         query = wsreg_query_create();
         wsreg_query_set_id(query, uuid);
         wsreg_query_set_location(query, location);
         comp = wsreg_get(query);

         if (comp != NULL) {
             /* The query succeeded.  The component has been found. */
             Wsreg_component **dependent_comps;
```

EXAMPLE 1 Unregister a component. *(Continued)*

```
            dependent_comps = wsreg_get_dependent_components(comp);
            if (dependent_comps != NULL) {
            /*
             * The component has dependent components.  The
             * component cannot be unregistered.
             */
            wsreg_free_component_array(dependent_comps);
            printf("The component cannot be uninstalled because "
                "it has dependent components\n");
            } else {
                /*
                 * The component does not have dependent components.
                 * It can be unregistered.
                 */
                if (wsreg_unregister(comp) != 0) {
                    printf("wsreg_unregister succeeded\n");
                } else {
                    printf("unregister failed\n");
                }
            }
            /* Be sure to free the component */
            wsreg_free_component(comp);
        } else {
            /* The component is not currently registered. */
            printf("The component was not found in the registry\n");
        }
        wsreg_query_free(query);
}
```

Usage Components should be unregistered before uninstallation. If the component cannot be unregistered, uninstallation should not be performed.

A component cannot be unregistered if other registered components require it. A call to `wsreg_get_dependent_components()` can be used to determine if this situation exists. See `wsreg_add_dependent_component(3WSREG)`.

A successful unregistration of a component will result in all components required by the unregistered component being updated in the product install registry to remove the dependency. Also, child components will be updated so the unregistered component is no longer registered as their parent.

When unregistering a product, the product should first be unregistered, followed by the unregistration of its first feature and then the unregistration and uninstallation of the components that comprise that feature. Be sure to use this top-down approach to avoid removing a component that belongs to a product or feature that is required by a separate product.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

See Also wsreg_add_dependent_component(3WSREG), wsreg_get(3WSREG), wsreg_initialize(3WSREG), wsreg_register(3WSREG), attributes(5)

Name XTSOLgetClientAttributes – get all label attributes associated with a client

Synopsis

```
cc [flag...] file... -lX11 -lXtsol [library...]

#include <X11/extensions/Xtsol.h>

Status XTSOLgetClientAttributes(display, windowid, clientattr);

Display *display;
XID windowid;
XTsolClientAttributes *clientattrp;
```

Parameters

display Specifies a pointer to the `Display` structure. Is returned from `XOpenDisplay()`.

windowid Specifies window ID of X client.

clientattrp Client must provide a pointer to an `XTsolClientAttributes` structure.

Description The `XTSOLgetClientAttributes()` function retrieves all label attributes that are associated with a client in a single call. The attributes include process ID, user ID, IP address, audit flags and session ID.

Return Values None.

Errors

`BadAccess` Lack of privilege.

`BadValue` Not a valid client.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe

See Also `libXtsol(3LIB)`, `XTSOLgetPropAttributes(3XTSOL)`, `XTSOLgetResAttributes(3XTSOL)`, `attributes(5)`

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name XTSOLgetPropAttributes – get the label attributes associated with a property hanging on a window

Synopsis

```
cc [flag...] file... -lX11 -lXtsol [library...]
#include <X11/extensions/Xtsol.h>

Status XTSOLgetPropAttributes(display, window, property, propattrp);

Display *display;
Window window;
Atom property;
XTSOLPropAttributes *propattrp;
```

Parameters

display Specifies a pointer to the `Display` structure; returned from `XOpenDisplay()`.

window Specifies the ID of a window system object.

property Specifies the property atom.

propattrp Client must provide a pointer to `XTSOLPropAttributes`.

Description The client requires the `PRIV_WIN_DAC_READ` and `PRIV_WIN_MAC_READ` privileges. The `XTSOLgetPropAttributes()` function retrieves the label attributes that are associated with a property hanging out of a window in a single call. The attributes include UID and sensitivity label.

Return Values None

Errors

`BadAccess` Lack of privilege

`BadWindow` Not a valid window

`BadAtom` Not a valid atom

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Unsafe

See Also `libXtsol(3LIB)`, `XTSOLgetClientAttributes(3XTSOL)`, `XTSOLgetResAttributes(3XTSOL)`, `attributes(5)`

"Setting Window Polyinstantiation Information" in *Solaris Trusted Extensions Developer's Guide*

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name XTSOLgetPropLabel – get the label associated with a property hanging on a window

Synopsis

```
cc [flag...] file... -lX11 -lXtsol [library...]
#include <X11/extensions/Xtsol.h>

Status XTSOLgetPropLabel(display, window, property, sl);

Display *display;
Window window;
Atom property;
m_label_t *sl;
```

Parameters

display Specifies a pointer to the `Display` structure; returned from `XOpenDisplay()`.

window Specifies the ID of the window whose property's label you want to get.

property Specifies the property atom.

sl Returns a sensitivity label that is the current label of the specified property.

Description Client requires the `PRIV_WIN_DAC_READ` and `PRIV_WIN_MAC_READ` privileges. The `XTSOLgetPropLabel()` function retrieves the sensitivity label that is associated with a property hanging on a window.

Return Values None.

Errors

`BadAccess` Lack of privilege.

`BadWindow` Not a valid window.

`BadAtom` Not a valid atom.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe

See Also `libXtsol(3LIB)`, `XTSOLgetPropAttributes(3XTSOL)`, `XTSOLsetPropLabel(3XTSOL)`, `attributes(5)`

"Setting Window Polyinstantiation Information" in *Solaris Trusted Extensions Developer's Guide*

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name XTSOLgetPropUID – get the UID associated with a property hanging on a window

Synopsis

```
cc [flag...] file... -lX11 -lXtsol [library...]
#include <X11/extensions/Xtsol.h>

Status XTSOLgetPropUID (display, window, property, uidp);

Display *display;
Window window;
Atom property;
uid_t *uidp;
```

Parameters

display Specifies a pointer to the `Display` structure; returned from `XOpenDisplay()`.

window Specifies the ID of the window whose property's UID you want to get.

property Specifies the property atom.

uidp Returns a UID which is the current UID of the specified property. Client needs to provide a `uid_t` type storage and passes the address of this storage as the function argument. Client must provide a pointer to `uid_t`.

Description The client requires the `PRIV_WIN_DAC_READ` and `PRIV_WIN_MAC_READ` privileges. The `XTSOLgetPropUID()` function retrieves the ownership of a window's property. This allows a client to get the ownership of an object it did not create.

Return Values None.

Errors

`BadAccess` Lack of privilege.

`BadWindow` Not a valid window.

`BadAtom` Not a valid atom.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe

See Also `libXtsol(3LIB)`, `XTSOLgetPropAttributes(3XTSOL)`, `XTSOLsetPropUID(3XTSOL)`, `attributes(5)`

"Setting Window Polyinstantiation Information" in *Solaris Trusted Extensions Developer's Guide*

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name XTSOLgetResAttributes – get all label attributes associated with a window or a pixmap

Synopsis

```
cc [flag...] file... -lX11 -lXtsol [library...]
#include <X11/extensions/Xtsol.h>

Status XTSOLgetResAttributes(display, object, type, winattrp);

Display *display;
XID object;
ResourceType type;
XTSOLResAttributes *winattrp;
```

Parameters

display Specifies a pointer to the `Display` structure; returned from `XOpenDisplay()`.

object Specifies the ID of a window system object. Possible window system objects are windows and pixmaps.

type Specifies what type of resource is being accessed. Possible values are `IsWindow` and `IsPixmap`.

winattrp Client must provide a pointer to `XTSOLResAttributes`.

Description The client requires the `PRIV_WIN_DAC_READ` and `PRIV_WIN_MAC_READ` privileges. The `XTSOLgetResAttributes()` function retrieves all label attributes that are associated with a window or a pixmap in a single call. The attributes include UID, sensitivity label, and workstation owner.

Return Values None.

Errors

`BadAccess` Lack of privilege.

`BadWindow` Not a valid window.

`BadPixmap` Not a valid pixmap.

`BadValue` Not a valid type.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe

See Also `libXtsol(3LIB)`, `XTSOLgetClientAttributes(3XTSOL)`, `XTSOLgetPropAttributes(3XTSOL)`, `attributes(5)`

"Obtaining Window Attributes" in *Solaris Trusted Extensions Developer's Guide*

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name XTSOLgetResLabel – get the label associated with a window, a pixmap, or a colormap

Synopsis

```
cc [flag...] file... -lX11 -lXtsol [library...]
#include <X11/extensions/Xtsol.h>

Status XTSOLgetResLabel(display, object, type, sl);

Display *display;
XID object;
ResourceType type;
m_label_t *sl;
```

Parameters

display Specifies a pointer to the `Display` structure; returned from `XOpenDisplay()`.

object Specifies the ID of a window system object whose label you want to get. Possible window system objects are windows, pixmaps, and colormaps.

type Specifies what type of resource is being accessed. Possible values are `IsWindow`, `IsPixmap` or `IsColormap`.

sl Returns a sensitivity label which is the current label of the specified object.

Description The client requires the `PRIV_WIN_DAC_READ` and `PRIV_WIN_MAC_READ` privileges. The `XTSOLgetResLabel()` function retrieves the label that is associated with a window or a pixmap or a colormap.

Return Values None.

Errors

`BadAccess` Lack of privilege.

`BadPixmap` Not a valid pixmap.

`BadValue` Not a valid type.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe

See Also `libXtsol(3LIB)`, `XTSOLgetClientAttributes(3XTSOL)`, `XTSOLsetResLabel(3XTSOL)`, `attributes(5)`

"Obtaining a Window Label" in *Solaris Trusted Extensions Developer's Guide*

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name XTSOLgetResUID – get the UID associated with a window, a pixmap

Synopsis

```
cc [flag...] file... -lX11 -lXtsol [library...]
#include <X11/extensions/Xtsol.h>

Status XTSOLgetResUID(display, object, type, uidp);

Display *display;
XID object;
ResourceType type;
uid_t *uidp;
```

Parameters

display Specifies a pointer to the `Display` structure; returned from `XOpenDisplay()`.

object Specifies the ID of a window system object whose UID you want to get. Possible window system objects are windows or pixmaps.

type Specifies what type of resource is being accessed. Possible values are `IsWindow` and `IsPixmap`.

uidp Returns a UID which is the current UID of the specified object. Client must provide a pointer to `uid_t`.

Description The client requires the `PRIV_WIN_DAC_READ` and `PRIV_WIN_MAC_READ` privileges. The `XTSOLgetResUID()` function retrieves the ownership of a window system object. This allows a client to get the ownership of an object that the client did not create.

Return Values None.

Errors

`BadAccess` Lack of privilege.

`BadWindow` Not a valid window.

`BadPixmap` Not a valid pixmap.

`BadValue` Not a valid type.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe

See Also `libXtsol(3LIB)`, `XTSOLgetClientAttributes(3XTSOL)`, `XTSOLgetResAttributes(3XTSOL)`, `XTSOLgetResLabel(3XTSOL)`, `attributes(5)`

"Obtaining the Window User ID" in *Solaris Trusted Extensions Developer's Guide*

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name XTSOLgetSSHeight – get the height of screen stripe

Synopsis

```
cc [flag...] file... -lX11 -lXtsol [library...]
#include <X11/extensions/Xtsol.h>

Status XTSOLgetSSHeight(display, screen_num, newheight);

Display *display;
int screen_num;
int *newheight;
```

Parameters

display Specifies a pointer to the `Display` structure; returned from `XOpenDisplay()`.

screen_num Specifies the screen number.

newheight Specifies the storage area where the height of the stripe in pixels is returned.

Description The `XTSOLgetSSHeight()` function gets the height of trusted screen stripe at the bottom of the screen. Currently the screen stripe is only present on the default screen. Client must have the Trusted Path process attribute.

Return Values None.

Errors

`BadAccess` Lack of privilege.

`BadValue` Not a valid *screen_num or newheight*.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe

See Also `libXtsol(3LIB)`, `XTSOLsetSSHeight(3XTSOL)`, `attributes(5)`

"Accessing and Setting the Screen Stripe Height" in *Solaris Trusted Extensions Developer's Guide*

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name XTSOLgetWorkstationOwner – get the ownership of the workstation

Synopsis

```
cc [flag...] file... -lX11 -lXtsol [library...]
#include <X11/extensions/Xtsol.h>

Status XTSOLgetWorkstationOwner(display, uidp);

Display *display;
uid_t *uidp;
```

Parameters

display Specifies a pointer to the `Display` structure; returned from `XOpenDisplay()`.

uidp Returns a UID which is the current UID of the specified `Display` workstation server. Client must provide a pointer to `uid_t`.

Description The `XTSOLgetWorkstationOwner()` function retrieves the ownership of the workstation.

Return Values None.

Errors BadAccess Lack of privilege.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe

See Also `libXtsol(3LIB)`, `XTSOLsetWorkstationOwner(3XTSOL)`, `attributes(5)`

"Obtaining the X Window Server Workstation Owner ID" in *Solaris Trusted Extensions Developer's Guide*

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name XTSOLIsWindowTrusted – test if a window is created by a trusted client

Synopsis

```
cc [flag...] file... -lX11 -lXtsol [library...]
#include <X11/extensions/Xtsol.h>

Bool XTSOLIsWindowTrusted(display, window);

Display *display;
Window window;
```

Description The `XTSOLIsWindowTrusted()` function tests if a window is created by a trusted client. The window created by a trusted client has a special bit turned on. The client does not require any privilege to perform this operation.

Parameters

display Specifies a pointer to the `Display` structure; returned from `XOpenDisplay()`.

window Specifies the ID of the window to be tested.

Return Values

`True` If the window is created by a trusted client.

Errors

`BadWindow` Not a valid window.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe

See Also `libXtsol(3LIB)`, `attributes(5)`

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name XTSOLMakeTPWindow – make this window a Trusted Path window

Synopsis

```
cc [flag...] file... -lX11 -lXtsol [library...]
#include <X11/extensions/Xtsol.h>

Status XTSOLMakeTPWindow(display, w);

Display *display;
Window w;
```

Parameters

display Specifies a pointer to the `Display` structure; returned from `XOpenDisplay()`.

w Specifies the ID of a window.

Description The `XTSOLMakeTPWindow()` function makes a window a trusted path window. Trusted Path windows always remain on top of other windows. The client must have the Trusted Path process attribute set.

Return Values None.

Errors

`BadAccess` Lack of privilege.

`BadWindow` Not a valid window.

`BadValue` Not a valid type.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe

See Also `libXtsol(3LIB)`, `attributes(5)`

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name XTSOLsetPolyInstInfo – set polyinstantiation information

Synopsis

```
cc [flag...] file... -lX11 -lXtsol [library...]
#include <X11/extensions/Xtsol.h>

Status XTSOLsetPolyInstInfo(display, sl, uidp, enabled);

Display *display;
m_label_t sl;
uid_t *uidp;
int enabled;
```

Parameters

display Specifies a pointer to the `Display` structure; returned from `XOpenDisplay()`.

sl Specifies the sensitivity label.

uidp Specifies the pointer to UID.

enabled Specifies whether client can set the property information retrieved.

Description The `XTSOLsetPolyInstInfo()` function sets the polyinstantiated information to get property resources. By default, when a client requests property data for a polyinstantiated property, the data returned corresponds to the SL and UID of the requesting client. To get the property data associated with a property with specific *sl* and *uid*, a client can use this call to set the SL and UID with *enabled* flag to `TRUE`. The client should also restore the *enabled* flag to `FALSE` after retrieving the property value. Client must have the `PRIV_WIN_MAC_WRITE` and `PRIV_WIN_DAC_WRITE` privileges.

Return Values None.

Errors

`BadAccess` Lack of privilege.

`BadValue` Not a valid *display* or *sl*.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe

See Also `libXtsol(3LIB)`, `attributes(5)`

"Setting Window Polyinstantiation Information" in *Solaris Trusted Extensions Developer's Guide*

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name XTSOLsetPropLabel – set the label associated with a property hanging on a window

Synopsis

```
cc [flag...] file... -lX11 -lXtsol [library...]
#include <X11/extensions/Xtsol.h>

Status XTSOLsetPropLabel(*display, window, property, *sl);

Display *display;
Window window;
Atom property;
m_label_t *sl;
```

Parameters

display Specifies a pointer to the `Display` structure; returned from `XOpenDisplay()`.

window Specifies the ID of the window whose property's label you want to change.

property Specifies the property atom.

sl Specifies a pointer to a sensitivity label.

Description The `XTSOLsetPropLabel()` function changes the sensitivity label that is associated with a property hanging on a window. The client must have the `PRIV_WIN_DAC_WRITE`, `PRIV_WIN_MAC_WRITE`, and `PRIV_WIN_UPGRADE_SL` privileges.

Return Values None.

Errors

`BadAccess` Lack of privilege.

`BadWindow` Not a valid window.

`BadAtom` Not a valid atom.

`BadValue` Not a valid *sl*.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe

See Also `libXtsol(3LIB)`, `XTSOLgetPropAttributes(3XTSOL)`, `XTSOLgetPropLabel(3XTSOL)`, `attributes(5)`

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name XTSOLsetPropUID – set the UID associated with a property hanging on a window

Synopsis

```
cc [flag...] file... -lX11 -lXtsol [library...]
#include <X11/extensions/Xtsol.h>

Status XTSOLsetPropUID(display, window, property, uidp);

Display *display;
Window window;
Atom property;
uid_t *uidp;
```

Parameters

display Specifies a pointer to the `Display` structure; returned from `XOpenDisplay()`.

window Specifies the ID of the window whose property's UID you want to change.

property Specifies the property atom.

uidp Specifies a pointer to a `uid_t` that contains a UID.

Description The `XTSOLsetPropUID()` function changes the ownership of a window's property. This allows another client to modify a property of a window that it did not create. The client must have the `PRIV_WIN_DAC_WRITE` and `PRIV_WIN_MAC_WRITE` privileges.

Return Values None.

Errors

`BadAccess` Lack of privilege.

`BadWindow` Not a valid window.

`BadAtom` Not a valid atom.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe

See Also `libXtsol(3LIB)`, `XTSOLgetPropAttributes(3XTSOL)`, `XTSOLgetPropUID(3XTSOL)`, `attributes(5)`

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name XTSOLsetResLabel – set the label associated with a window or a pixmap

Synopsis

```
cc [flag...] file... -lX11 -lXtsol [library...]
#include <X11/extensions/Xtsol.h>

Status XTSOLsetResLabel(display, object, type, sl);

Display *display;
XID object;
ResourceType type;
m_label_t *sl;
```

Parameters

display Specifies a pointer to the `Display` structure; returned from `XOpenDisplay()`.

object Specifies the ID of a window system object whose label you want to change. Possible window system objects are windows and pixmaps.

type Specifies what type of resource is being accessed. Possible values are `IsWindow` and `IsPixmap`.

sl Specifies a pointer to a sensitivity label.

Description The client must have the `PRIV_WIN_DAC_WRITE`, `PRIV_WIN_MAC_WRITE`, `PRIV_WIN_UPGRADE_SL`, and `PRIV_WIN_DOWNGRADE_SL` privileges. The `XTSOLsetResLabel()` function changes the label that is associated with a window or a pixmap.

Return Values None.

Errors

`BadAccess` Lack of privilege.

`BadPixmap` Not a valid pixmap.

`BadValue` Not a valid *type* or *sl*.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe

See Also `libXtsol(3LIB)`, `XTSOLgetResAttributes(3XTSOL)`, `XTSOLgetResLabel(3XTSOL)`, `attributes(5)`

"Setting a Window Label" in *Solaris Trusted Extensions Developer's Guide*

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name XTSOLsetResUID – set the UID associated with a window, a pixmap, or a colormap

Synopsis

```
cc [flag...] file... -lX11 -lXtsol [library...]
#include <X11/extensions/Xtsol.h>

Status XTSOLsetResUID(display, object, type, uidp);

Display *display;
XID object;
ResourceType type;
uid_t *uidp;
```

Parameters

display Specifies a pointer to the `Display` structure; returned from `XOpenDisplay()`.

object Specifies the ID of a window system object whose UID you want to change. Possible window system objects are windows and pixmaps.

type Specifies what type of resource is being accessed. Possible values are: `IsWindow` and `IsPixmap`.

uidp Specifies a pointer to a `uid_t` structure that contains a UID.

Description The client must have the `PRIV_WIN_DAC_WRITE` and `PRIV_WIN_MAC_WRITE` privileges. The `XTSOLsetResUID()` function changes the ownership of a window system object. This allows a client to create an object and then change its ownership. The new owner can then make modifications on this object as this object being created by itself.

Return Values None.

Errors

`BadAccess` Lack of privilege.

`BadWindow` Not a valid window.

`BadPixmap` Not a valid pixmap.

`BadValue` Not a valid type.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Unsafe

See Also `libXtsol(3LIB)`, `XTSOLgetResUID(3XTSOL)`, `attributes(5)`

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name XTSOLsetSessionHI – set the session high sensitivity label to the window server

Synopsis

```
cc [flag...] file... -lX11 -lXtsol [library...]
#include <X11/extensions/Xtsol.h>

Status XTSOLsetSessionHI(display, sl);

Display *display;
m_label_t *sl;
```

Parameters

display Specifies a pointer to the `Display` structure; returned from `XOpenDisplay()`.

sl Specifies a pointer to a sensitivity label to be used as the session high label.

Description The `XTSOLsetSessionHI()` function sets the session high sensitivity label. After the session high label has been set by a Trusted Extensions window system TCB component, `logintool`, X server will reject connection request from clients running at higher sensitivity labels than the session high label. The client must have the `PRIV_WIN_CONFIG` privilege.

Return Values None.

Errors `BadAccess` Lack of privilege.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe

See Also `libXtsol(3LIB)`, `XTSOLsetSessionLO(3XTSOL)`, `attributes(5)`

"Setting the X Window Server Clearance and Minimum Label" in *Solaris Trusted Extensions Developer's Guide*

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name XTSOLsetSessionLO – set the session low sensitivity label to the window server

Synopsis

```
cc [flag...] file... -lX11 -lXtsol [library...]
#include <X11/extensions/Xtsol.h>

Status XTSOLsetSessionLO(display, sl);

Display *display;
m_label_t *sl;
```

Parameters

display Specifies a pointer to the `Display` structure; returned from `XOpenDisplay()`.

sl Specifies a pointer to a sensitivity label to be used as the session low label.

Description The `XTSOLsetSessionLO()` function sets the session low sensitivity label. After the session low label has been set by a Trusted Extensions window system TCB component, `logintool`, X server will reject a connection request from a client running at a lower sensitivity label than the session low label. The client must have the `PRIV_WIN_CONFIG` privilege.

Return Values None.

Errors

`BadAccess` Lack of privilege.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe

See Also `libXtsol(3LIB)`, `XTSOLsetSessionHI(3XTSOL)`, `attributes(5)`

"Setting the X Window Server Clearance and Minimum Label" in *Solaris Trusted Extensions Developer's Guide*

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name XTSOLsetSSHeight – set the height of screen stripe

Synopsis

```
cc [flag...] file... -lX11 -lXtsol [library...]
#include <X11/extensions/Xtsol.h>

Status XTSOLsetSSHeight(display, screen_num, newheight);

Display *display;
int screen_num;
int newheight;
```

Parameters

display Specifies a pointer to the `Display` structure; returned from `XOpenDisplay`.

screen_num Specifies the screen number.

newheight Specifies the height of the stripe in pixels.

Description The `XTSOLsetSSHeight()` function sets the height of the trusted screen stripe at the bottom of the screen. Currently the screen stripe is present only on the default screen. The client must have the Trusted Path process attribute.

Return Values None.

Errors

`BadAccess` Lack of privilege.

`BadValue` Not a valid *screen_num* or *newheight*.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe

See Also `libXtsol(3LIB)`, `XTSOLgetSSHeight(3XTSOL)`, `attributes(5)`

"Accessing and Setting the Screen Stripe Height" in *Solaris Trusted Extensions Developer's Guide*

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name XTSOLsetWorkstationOwner – set the ownership of the workstation

Synopsis

```
cc [flag...] file... -lX11 -lXtsol [library...]
#include <X11/extensions/Xtsol.h>

Status XTSOLsetWorkstationOwner(display, uidp);

Display *display;
uid_t *uidp;
XTSOLClientAttributes *clientattrp;
```

Parameters

display Specifies a pointer to the `Display` structure; returned from `XOpenDisplay()`.

uidp Specifies a pointer to a `uid_t` structure that contains a UID.

Description The `XTSOLsetWorkstationOwner()` function is used by the Solaris Trusted Extensions `logintool` to assign a user ID to be identified as the owner of the workstation server. The client running under this user ID can set the server's device objects, such as keyboard mapping, mouse mapping, and modifier mapping. The client must have the Trusted Path process attribute.

Return Values None.

Errors `BadAccess` Lack of privilege.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Unsafe

See Also `libXtsol(3LIB)`, `XTSOLgetWorkstationOwner(3XTSOL)`, `attributes(5)`

"Accessing and Setting a Workstation Owner ID" in *Solaris Trusted Extensions Developer's Guide*

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.